SE

- Application
- System level
- Service (SaaS)

Software
- program (installation, dependencies)
- Config
- Documentation (user, system)
- Support
- ...

Eigenschaften
- Size
- no physical borders
- Exposure
- Distribution (cheap, updates not enforced)
- Changing Environment
- Long Lifetime
- hard to measure

=> many Projects fail

Software Engineering through Requirement Analysis → Domain driven Design

systematic
disciplined
quantifiable
=> development
Operation
Maintainence

Design Principles, Patterns

Architecture

SE

Tools
+ productivity
- getting used to

IDE - Integrated Development Enviroments
- File Explorer
- Syntax Highlighting
- Code completion
- Problem Indicators
- Code Navigation
- Refactoring
- Testing
- Debugger
- Version Control
- Terminal

Command Shell e.g. Bash
- interactive programming language
- Commands = executable programs
- Command_Name [Flags] Arguments
  - -k
  - --recursive=arg
- man Command_Name

C1 | C2 redirects std out from C1 in C2
C1 1> FILE redirects out
2 error
* both

exit code 0 success
-255 error code
C1 || C2 executes C2 if C1 fails
C1 && C2 succedes

- touch
- ls, tree
- mv
- cp
- rm -r for directories
- chown, chmod
- cat concatenate files
- wc word count
- sort
- grep search through file
- find files
- pwd
- mkdir
- cd
- rmdir

```
FOO=1
BAR=$(( $FOO*2 ))
if [ $FOO -lt 123 ];
  then echo "hi";
  else ... ;
fi
for file in $FILES; do process $FILE; done
while [ $N -lt 100 ]; do N=$(($N+1)); ...; done

function name(){
  locale N=$1; # first Param
  echo ...;
}
n=$(command argument) executes function
```

Build Systems - automate repetitive tasks on code
- compile, run, clean, interpreter, generate docs, style, deploy
- config build scripts
- incremental: only update changed files

Version Control Systems
- history
- multiple developers

VCSs
- Centralized - all changes in a single repository (Subversion)
- Decentralized - multiple independent repositorys (Git, Mercurial) (widely used)

Commit - atomic change (organizes)
- hash
- author, date
- message
- Line-wise additions deletions

Branches - organize commits (organize)
- different line of work
- commits partially-ordered
- merge ⚠ conflicts manual solving

Repository
- push, pull

workflow e.g.
- blessed repository
- integration manager
- public
- private developer

Software Engineering Process

responsibilities and interactions of development teams

for Stakeholders
- External
  - Users
  - Customers
  - Government
- Internal
  - Developer
  - Project manager
  - Company developing

Quality assurance
Consistency, Repeatebility
Collaboration, coordination
Risk Management anticipate problems
Scalability, efficiency
Continuous Improvement

Requirement Analysis
↓
Design
↓
Development
↓
Validation
↓
(Maintenance)
↓
Operation
↓
Customer feedback

Waterfall Method

one Phase nach andere
error ⇒ repeat

- late verification
- no changes easy later

z.B. dependencies new hardware Competitors,

Shorter cycles
more successful

Agile (many diffent)

iteration / sprint - result executable
fixed time, immutable
- backlog → next iteration
- changes only at end

Requirements
continous discovery

Planning
Stakeholder
- Scope - How much until value
- Priority
- Composition of Reteam
- Date

maxed by Capacity = Σ to implement Story Points
- ~ Velocity prior iterations (Σ completed)
- ~ available developers

User Stories
- Description
- Story points - z.B. Planning Poker
  - simulten voting until consens

Developers
- technical consequences
- detailed scheduling

↓ break up in 4-16h tasks ← developers sign up for

Retrospective
often between iterations
⇒ find actions to improve process

Kanban board ← team intern → **Daily Standups**
- column = steps
- colors = type of task
- who's working on what

- report progress, synchronise
- redistribut if problems
- assign new tasks

Individuals & Interactions > Process & Tools
Working Software > Comprehensive Documentation
Customer Collaboration > Contract Negotiation
Respond to change

Requirements Analysis – identifying documenting validating managing

software: supposed to do properties

often informal
⚠ implicit assumptions

Who?

Stakeholder ⇔ System Requirements
z.B. ... | # needs database

What?

- functional
  - User stories
  - Use cases
- non-functional
  - Product
    - Portability
    - Reliability
    - Performance
      - Space
      - Efficiency
    - Usability
  - Organizational
    - Delivery
    - Implementation
    - Standards
  - External
    - Interoperability
    - Ethical
    - Legislative
      - Safety
      - Privacy

HIPAA – Health Insurance Portability and Accountability Act
GDPR: General Data Protection Regulation

Elicitation

Stakeholder identification → viewpoints → Interviews, ~~Survey~~

identification: operates, uses, benefits, purchasing, regulating, opposes

viewpoints:
- Interactor
- Indirect
- Domain
  - z.B. Legal

Interviews, Survey
Closed ⇔ Open
- cooperation?
- implizit assumptions?

Gap analysis
Prototyping
Feasibility Study
- Technical
- Operational
- Economical
- Legal
- schedule

→ even start?

Documentation

Standard Dokument

- Introduction
  - Purpose of Document
  - Scope of Product
  - Definitions, acronyms
  - References
  - Overview
- General description
  - Product Perspective
  - Functions
  - User characteristics
  - Limitations
  - Assumptions, dependencies
- Specific requirements
- Appendices, Index, etc.

Requirements
- non-functional: free form
- functional
  - User Stories ←Informal→ Use cases
    - Who, What, Why
    - As a ..., I want .. so that ...
    - Acceptance Criteria
    - ~~Informal~~
    - Brief → Brief to long
      - Understandable
      - Valuable
      - Sized Appropriately – Underspecified ⇔ Overspecified
      - Estimatable Effort
      - Independent
      - Testable

Mnemonic: INVEST negotiable

  - Structured sections
    - Title
    - Primary Actor
    - Goal
    - Scope
    - Stakeholders for this
    - Precondition
    - Main Success Scenario
    - Acceptance Criteria

Illustrations

Flow chart
- Nodes
  - round → initial or terminal
  - rectangular → intermediate
  - rhombus → decisions
- Arrows → transitions

states

↓ extends

BPMN – Business Process Model and Notation
- rows for departments
- → concurrent processes

BRUNNEN

Validation

Acceptance tests: written before/concurrently to impl. if user story/case done → never allowed to fail again (avoid regressions)

Metrics for non-functional requirements → continuously monitored

Code review (by 3th Party) z.B. before investment by governments
Legal

Domain-Driven Design

DDD – goal reduce incidental complexity
better communication to stakeholders

Language of domain
not static view, broken abstraction of code
need to be kept up to date (when code changes)

domain – subject area of software
" models – abstract concepts
in Language of domain
visualized with UML-Diagrams

solution space – implementation
close to model ⇒ craft

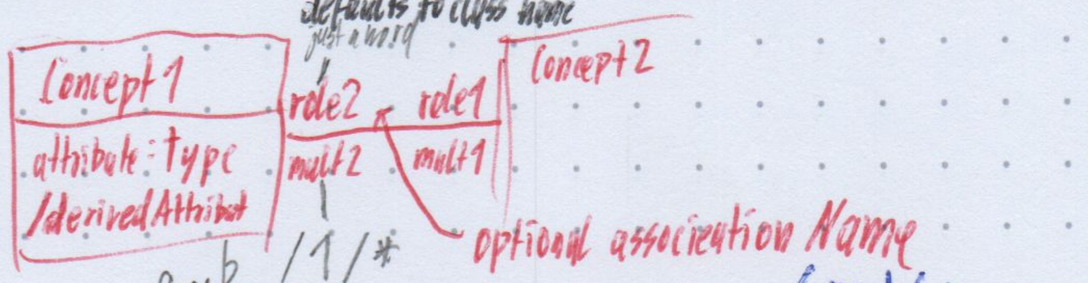
Modelling (use Vokabular of Domain)
1. identify concepts
2. generalizations between concepts
3. properties
attributes — Seperate sections for primitive type / compound type (include when need to be preserved)
associations — A part of B, line item of transaction B, related transaction, product of transaction B
4. derived properties
Can be computed from other properties

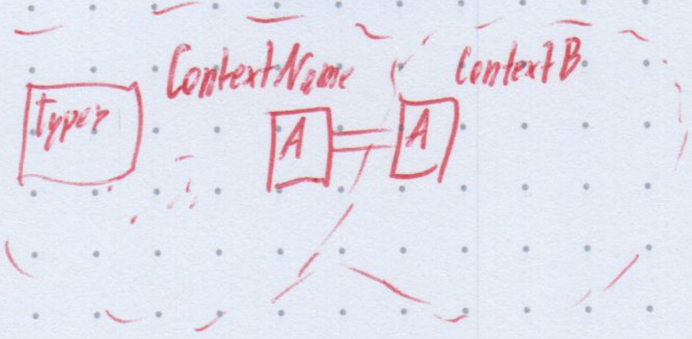
talk to domain experts (multiple years making experience)
noun phrases (unstructured Interviews)
category list (candidates for concepts, list categories first z.B. Physical objects, Specifications, Places, Events, Transactions [Items], Organisations, Roles, Containers)
+specific meaning in domain
-synonyms -generic -impl-details

B specific case of A
B can be used in context expecting A
B ▷ A

UML Box

Implementation
Concepts ⇒ traits or classes
generalizations ⇒ traits, extended by classes
attributes ⇒ " with accessors
association ⇒ depending on access direction and mult
[List of] accessor in one ~~concept~~type
derived properties ⇒ accessor functions?





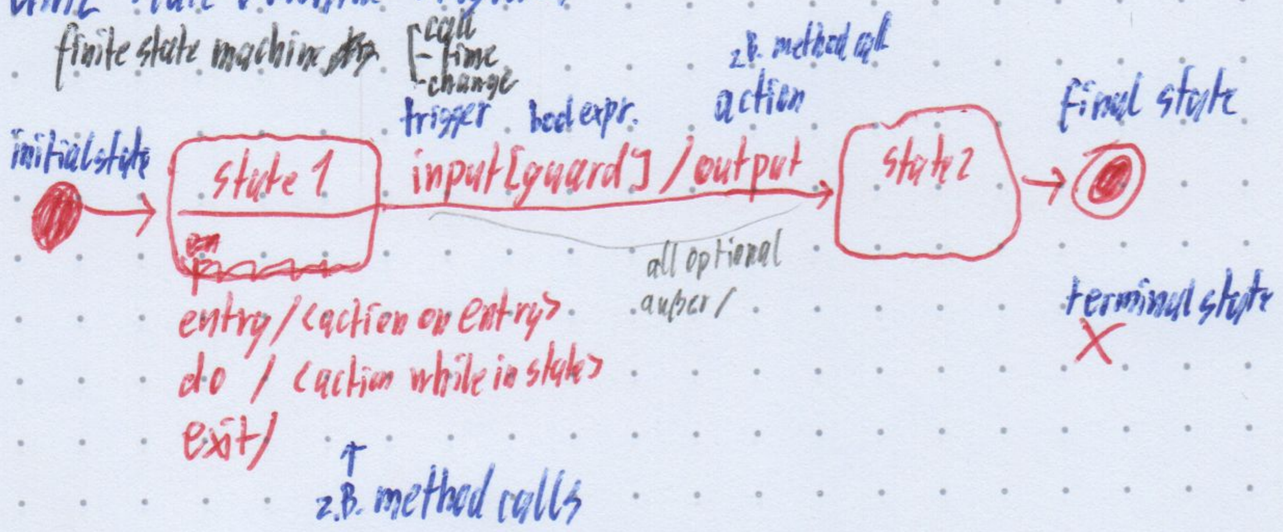
Context bounds
1 uncept - n contexts

↓ only types

Behavior Domain Models

UML State Machine Diagram

finite state machine



Implementation
States ⇒ enum
transitions as func.

```
class ...
var currentState: States = Initial

def transition: Unit =
  currentState match
    case Initial =>
      val guard = ...
      if (guard) {
        output.action();
        currentState = ...
      }
    case ...
```

BRUNNEN

Software Quality

extern Stakeholder (message: How positive it is perceived) – intern dev

Factors:

		stakeholder relevant
Validity	Efficiency	Extensibility
Responsiveness	Reusability	Maintainability
Ease of use		Understandability
Functionality		Enjoyability
Compability		
Portability		

low-quality ⇒ fail

Assure Quality

- external: continues Feedback stakeh., Domain Modeling, Testing, Static Analysis
- internal quality: Design Principles and code metrics, Design patterns, Architecture

Testing

+ assure quality
– can only proof existence of bugs
– costly 15-25% dev time
– need maintance

confirm requirements, find vulnerabilities, detect regressions

SUT – System under test

Who

- automated
- manual

Techniques

- Example-Based: example inputs, expected (+ easy, – often not exhaustive)
- Property-Based: mathematical properties, random inputs (+ exhaustive, – difficult)
- Fuzz Testing: test if crash, discover vulnerabilities
- ...

~~What~~ Purpose

- acceptance (Requirements)
- smoke
- usability
- accessability
- security
- performance
- ...

fuzzer can ~~use~~ be based on
- input structure (Grammar z.B. SQL, JSON, File formats, protocols)
- dynamic analysis information z.B. (Coverage of code, Concolic Execution (conditions in code))

Which Part

rule of thumb: more ... more likely to break, harder to identify problem

- Unit Tests: test Classes, Methods
  - n Test cases
  - assertions to verify
- Integration: n units and interaction
  - ~~n units~~
- System (End-to-End) – entire system often Acceptance Tests

for code behind simple checks → Custom generator to avoid useless test cases (still also tests other)

shrink found fail to minimal counterexample
return seed of failing

common properties
- unary func
  - Totality – ∀x no exception
  - Injectivity
  - Surjectivity
  - Idempotent – f(f(x)) = f(x), z.B. sort
- inverse func.
  - Isomorphism – z.B. deserialize(serialize(x)) = x, well understood

Binary func
- Identity – x+0=x
- Commutativity
- Associativity

Binary relations
- Reflexivity
- Symmetry
- Transitivity

happy path – how it was meant to be used, ↯ trigger errors
corner, exceptional cases – z.B. null, negative, empty param
non-trivial behaviour – test them, not getter/setter

```
test ("name"):
  ... should be ...
  forAll { (param) =>
    ... should be ...
  }
```

AAA pattern – Arrange, Act, Assert
DRY – don't repeat yourself ⇒ helper func.
KISS – Keep it simple, stupid so tests don't need testing
Title → problem should be clear
Isolation → where bug? ⇒ reduce dependencies if possible

```
val m = mock[Class]
m expects 'func with args
(*...)*
```

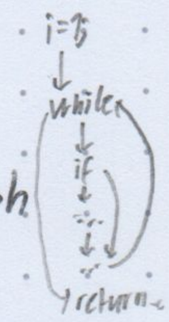
Test Fakes, Mock through Dependency Injection
- check if called
- imitate functionality

avoid:
- Side Effects ⇒ before, after function to exec. effects for test case
  - ↓ can cause beforeAll, afterAll
- Flaky Tests – sometimes fail, sometimes pass

Test Coverage

Structural

- Statement Coverage = Executed stmt / All stmt
- Edge Coverage: percentage of edges in control-flow graph
- Path Coverage: % paths from start to end
  - – unachievable in Praxis
  - can be infinite or unfeasible paths



Logical

- Condition ⚠ – short circuit evaluation
  - % conditions at least once evaluated true and false
  - conditions = bedean Expressions without &&, ||, !, &, |
- Decision
  - "decisions" alternative Def. branchpoint but tmp vars would change Decisions
  - Decisions = all bedeam Expressions (each parts) ⊆ Conditions

MCDC – Modified Condition/Decision Coverage
- ∀ entry and exit points min 1 invoked
- ∀ decisions ∀ outcomes min on 1
- ∀ conditions in a decision shown to independently affect decisions outcome
- + for critical software

BRUNNEN

Static Analysis

Tests can't proof absence of bugs

- Formal Verification
  + can prove absence 100%
  - time, specific, which property?
- Dynamic Analysis
  analyse trace data of exec.
  - Limited inputs, which to use?
- Static Analysis - extract info without executing
  + automated
  + strong gurantees
  + $\forall$ possible exec. paths
  - undecidable

Usage
- IDE
  - Recomendations
  - Refactorings
- Compilers
  - check
  - optimize
- Formal Verification
  - gather info
  - ensure valid spec
- Security Analysis
  - find vulnerabilities
  - malware
  - (data leaks)

Intra- ↔ Inter procedural
- (method at a time
- whole programm

Flow - sensitive ⇔ insensitive
compute result @ statements - globally valid

Context - sensitive ⇔ insensitive
- use multiple contexts
  e.g. calling method
- consider each piece only once

Completness - no results runtime impossible
Precision = $\frac{\#\text{true positives}}{\#\text{results}}$

often conflicting

Soundness - produce $\forall$ results possible @ runtime
Recall = $\frac{\#\text{true positives}}{\#\text{expected results}}$

often not achievable ⇒ Soundiness
- deliberate, document unsound choices

Scalability ⇒
large softwares, limited time memory

Control Flow Analysis
- order & dependencies of stmts

Data Flow Analysis
- where value
  - used
  - defined

Abstract Interpretation + for soundness
- over-approximate semantics z.B. abstract values $]-\infty, \infty[$

Symbolic Execution - path explosion
- interprete with symbolic values z.B.
  ~~for branches~~ Fork on branches
  if(x>0){ i = λ
  } else { i = λ+2, λ>0 }

Constraint Based Analysis - may be hard to solve
extract constraints on solution
solve for most precise solution

Type and Effect Systems
- Type System assign Type to $\forall$ Expr. z.B. @ compile time already
- Effect System assign effect - seltener

compositional (follows from subexpressions)

Common Analysis

CFG - Control Flow Graph
Dominator tree - Which stmt exec before another
Post-dominator tree - " after

Monotone Data Flow
- Liveness - value readable after certain point z.B. - dead code removal
- Reaching definitions - where can values come from - Loop invariant code motion
- Available expressions - value allready computed - common subexpr. elimination

Graph-Based Interprocedural Data-Flow
turn into graph reachability problems
efficient for suitable problems (important: distributivity)
e.g. for taint analyses (track sensitive/dangerous data)

Call Graph
prerequisite interprocedural analysis
diff Algorithms precision & scalability

Points-To Analysis e.g. for Call Graphs
Which objects can var refer to
- Type Abstraction
- Allocation-site Abstraction
- Shape Abstraction

precise ↓

Escape & Alias Analysis
- dynamic scope of objects
  e.g. if's limits for optimizing
- relationship of vars
  may / must 2 vars same object
  often & flow(in) sensitive

Immutability
Purity
- deterministic,
- no side effects

+ safe, maintainable

Slicing - slicing criterion
- Forward - $\forall$ stmt affected by
- Backward - $\forall$ stmt affecting

+ debugging, clone detection, parallelization

Design Principals

KISS - Keep it simple, stupid!

complexity ⇒ harder to understand, test, maintain — metrics

- accidental
  - from implementation
  - ⇒ refactor
- essential
  - from domain
  - unavoidable

- Lack of documentation
  - understand ⇒ reuse
- Bad architectural decision
- Premature performance optimizations
  - only if problem affecting stakeholder
- Uncontrolled side-effects
  - non-deterministic
- Developer inexperience
  - eg. don't use patterns ⇒ pair with experienced
- Software evolution
  - pressure from stakeholders

- fan in - # functions calling measured func.) static analyses
- fan out - " called by) overapproximates practice
- length of Function - # lines
- Cyclomatic complexity - # independent paths through control
  - linear
  - diff set of edges ⇒ for single method

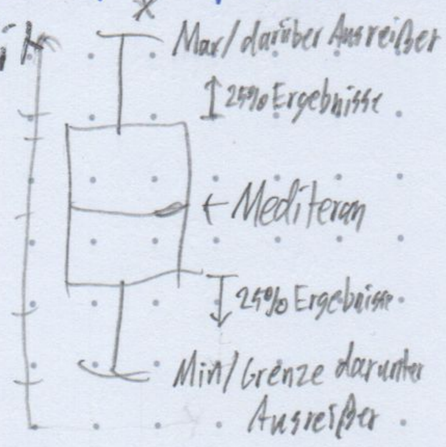
$$= |Edges(G)| - |Nodes(G)| + 2 \cdot |components(G)|$$

- components(G) = connected subgraph not part of larger
- ∀ pairs undirected path

- Depth of conditional nesting (if, loops)
- Coupling - interdependence of classes

⇒ detect complex components in package/project / longitudinal analyses
- trends over time

Grafik



Separating Side-Effects from Domain Logic

- non-deterministic, implicit ⇒ harder to understand

pure - code without Side-Effects

- Side causes: not constants
- Side effects: mutating var, time, network, databases, ...

Arguments → Function → Return Value

functional statt imperative

extract Pure Core

Immutable Parameter — return new copy instead

immutable allows destructive updates

Avoid Using Ambiguous Data Types

- no info about meaning

static type system prevents errors

enum

klassen

LSP - Liskov Substitution Principle / strong behavioral subtyping

subtype may replace type without breaking programm

formal: $\varphi(x)$ provable property

$$\forall x : T.\ \varphi(x) \Rightarrow \forall y : S <: T.\ \varphi(x)$$

- S subtype of T

parameters	contravariant in subtype (praiser)
return types	covariant + no new exceptions
Preconditions	~~strengthened~~ weakened
Post-	strengthened
Invariants	

Limit extend of changes, Wiederverwendbarkeit

DRY - Don't Repeat Yourself
may forgen repetion when changing
variables for common values
expressions ⚠ seiteneffekte

Code Similarity Metrics
- Token-based
- tree-based
- Semantik-based
  - dependence graph
    - control
    - duty

new function for repetition inside functions
- auch höherer Ordnung

new or super class for repetition inside classes

SRP - Single Responsibility Principle
- Doing - Calculate, Create, Controlling
- Knowing - Provide access, Keep references

refactor into multiple classes

High Cohesion, Low Coupling

- class/interface A coupled to B <= A requires B (auch indirekt)
  - has attribute
  - contains (sub-) expression
  - calls methods of
  - has method that references (as param, local var, return type)
  - is subtype

metric CBO-coupling between objects for class
CBO(C) = |{D | C coupled to D ∧ (C ≠ D)}|
- auch Standard bib.
- nicht rekursiv

- hard to understand in isolation

- relation between methods of a class
  - low ⇒ little in common ⇒ hard to understand accidental coupling ⇒ split class
  - metric LCOM - Lack of Cohesion of Methods
    - LCOM(C) = # connected components of G(M, E)
      - ∈ [0, |M|]
      - = |{⟨m,n⟩ ∈ M×M | ∃ f ∈ F: m,n access f, m ≠ n}|
    - M = methods of C
    - F = instance fields of C

ISP - Interface Segregation Principle
~~impleme~~ users only need what they want ⇒ avoids coupling, shorter impl.

Violation in Java standard library
- Iterator<E>{
  - boolean hasNext();
  - E next();
  - void remove(); - nicht für alle UnsupportedOperationException
- }

⇒ Iterator<E> ← RemovingIterator<E>

Design Patterns - proven solutions

+ not reinvent wheel
+ hide complexity
- boilerplate (not immediately for domain)
- Unjustified → Use don't smaller design
- Dogmatic → apart pattern

languages still often used

indication for
missing language features
Monad Pattern => Built-in effects
Singleton " => Scala's objects
Prototype " => Scala's ~~obj~~case classes
Visitor " => Pattern matching
Strategy " => Anonymous functions
easily misused features
decorater pattern delegation over inheritance
RAII pattern manage memory management

Object-Oriented Design Patterns by a book 1994

Creational - How objects are created, Structural - class composition, Behavioral - class interaction responsibility distribution

Abstract Factory +portability through diffent factories

Participants
Abstract Factory - interface
Concrete Factories
Abstract Product
Concrete Product
Client - calls ConcreteFactorie through interface

+ abstract concrete products as product families
  - consistent family
  - changing easy through diff. factory
- hard to add unforeseen products
  change all factories
- non-standard learning required

Structural - class decomposition

Decorator - ~~extend fun~~

structure
Component - interface
Decorators - extend referenced Component
Basic Components - core functionality

+ extend class functionality without inheritance z.B. avoids multiple inheritance
+ dynamic ↔ static at runtime
- hard to make order independent (decorator stack)
- boilerplate for not extended methods

Scala builtin syp
trait I extends C (class)
class D extends C with I with ... (I done last, ~~decorated~~ done first)

Behavioral - class interaction responsibility distribution

Iterator

```
trait Iterator[A]:
  def next: Option[A]
```

+ hides complexity of data structure - can be complicated multiple versions
  => less error prone, efficient
  z.B. depth first, breadth-first tree
- imperative => hard to parallelize
  Folds are functional version of it

~~Front~~ Observer

Structure
Subject (attach, detach, notifyObservers()) → * Observer - interface update(...)
z.B. Subject, Delta
ConcreteSubject (getState(): T, modifyState(...)) ← ConcreteObserver update(...)
only Placeholders

+ abstracts coupling
+ allows broadcasts
- ⚠ update cascades
- no differentiating update types

(many instances)

Functional Design Patterns — inspired from abstract algebra, category theory

Monoids (A, combine, empty)

combine : A×A → A — associativ
empty : A — left & right identity

implemented as scala type class

+ abstract accumulation

scala stuff
- type class - instance inferred given object by compiler
- type constructor - func : Type → Type F[_]

Foldable

└ types defining Fold function
└ accumulate Velements of data structure with Monoid

fold : f structure of As × Monoid of A → A

foldMap : structure of As × Monoid of A × Monoid of B × f : A → B → B

+ abstracts accumulation hiding structure implementation

- can't profitate
z.B. search in sorted structure

Categorys (C, Obj, id, ○)

- ○ : C(Y,Z) × C(X,Y) → C(X,Z) Assoziativ
- id : C(X,X) Left & Right identity
- Menge Objekte X, Y, Z ∈ Obj
  - typischerweise Typen ∈ Obj ⇒ C(X,Y) = Funktion X → Y) Category of types and functions
- set of Arrows C(X,Y) ⇒ nicht zwischen allen Paaren existiert Pfeil

+ generalizes Monoid
gibt nicht zwischen allen Elementen Pfeile

Functors F : C → D

(C, D: categorys)

- $F_{obj} : Obj_C \to Obj_D$ preserves identity
- $F_{Arr} : C(X,Y) \to D(F_{obj}(X), F_{obj}(Y))$ preserves Composition

Monads (F, pure, join)

combined Arrow to normal arrow
- C(C(F(F(A)), F(A))
- C(C(A), F(A))
- Functor F : C → C

+ abstracts side effects

Associativity: join ○ F(join) = join ○ join
Left identity: join ○ F(pure) = id
Right identity: join ○ pure = id

Traversables

+ generalizes Folds collecting in Monad

BRUNNEN

Software Architecture - high level design, organisation

Recent changes
open source => cheap
standards => easier swaping Komponents
Container (lightweight virtualization)
=> easily added/removed => microservices architecture

Dimensions

Architecture Characteristics

Operational about execution
- Availability incl. failure recovery
  - incl. failure recovery
- Performance
  - latency, throughput, resources
- Scalability
  - threads, servers, database

Structural about code
- Extensible
  - new functionality
- Maintainability
  - adapting, enhancing
- Reuseability
  - requires low coupling
- Configurability
  - for user

Cross-Cutting - affect all parts
- Localization
  - diff languages, units
  - requires good abstraction
- Accessability
  - users with disability?
- Privacy
- Security

Architecture Styles - components
organisation, dependencies
communication between

Software often uses multiple

Layered
- open ↔ closed
  - requests must pass layers
  - allow bypassing

+ Seperation of concerns
+ easy
- sinkhole anti-pattern (overhead/cruft for request through layers with little processing)
- vertical => hard to scale

Service Oriented
- independent services on bus
- shared ↔ own database
  - don't need to know each other
  - Design
  - independence
  - ↓ consistency

+ encapsulating (allows third party service)
+ scale by new instances of service
- hard to debug
- network overhead
- large scale - unclear dependencies

Pipes and Filters
- pipes connecting
- Filters - transforms datatype
  - independent
  - stateless
- branching pipes allowed
  - stream/chunks of data

+ Filters concurrent (z.B. parallel on diff machines)
+ Seperation of concerns
+ may small data chunks
- pipes overhead (buffer, serialization)
- some Filter need all data => not only chunks
- error handling (keep original data until success)

Broker, Microservices

Development process:

requirement analyses → Choosing Architecture + structure into components → Development Team
- Influences
  - Domain
  - Requirements
  - Architecture characteristics
  - Organizational factors
  - Development process
- Hard to change, enforce retroactively
  - but often delayed (fear of wrong)
  - => experts/prototype
- Hard to enforce
  - => clear communication
- Document decision
  - z.B. one file in repository + sync with code
    - structured text, diagrams

BRUNNEN

Design Principles

Performance Engineering

! premature optimization => complexity
only critical

ressources
response time
useful work

Application ↔ System

Metrics

Running Time

- variating
  - system load
  - disk usage
  - thread sync
  - JIT - Just In Time optimization
  - Garbage collection => warm up Phase to allow deallocation

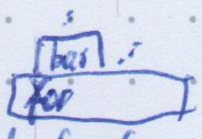
=> repeat

⚠ dead code elimination

Wall time

User CPU time - excl kernel
Σ threads

System CPU time - excl user

Flamegraphs


of time per method und caller-callee

Benchmark

JMH - Java Microbenchmark Harness

```
@Param(Array("100", "1000"))
var size$Int = _
var array: Array$Seq[Int] = _

@Setup
def method

@Benchmark
def method
```

CPU Usage

Profiler z.B. visualvm

programm used cycles
max. cycles

Memory Usage

identify big objects

in Bytes over time
Garbage collection => spikes
↗ memory => costly allocation

Throughput

operations #op / time

Peak - max measured
Bandwith - theoretical max

FLOPS - #Floating-point ops / s
IOPS - disk I/O ops / s
#Bytes send/received /s
#Requests handeld /s

KB = 10^3 B	KiB = 2^{10} B
MB = 10^6 B	MiB = 2^{20} B
GB = 10^9 B	GiB = 2^{30} B
TB = 10^{12} B	TiB = 2^{40} B
PB = 10^{15} B	PiB = 2^{50} B

JIT - Just In Time Optimizations

Optimize
- Null check eliminated
- Branch reorder
- Methods inlined
- Loop-unrolling
- invariante rausziehen

if getting invalid => deoptimize

=> unpredictable

Tool: jitwatch
↳ shows when method optimized

Theoretical - scaling for >> n

Time Complexity
- Worst Case - Big O
- Average Case - Theta Θ

z.B. Quicksort Θ(log n) O(n²)