ACHTUNG
SWT-2 KAUDERWELSCH !

1. Intro

Topics according to development process:

- Requirements engineering } Requirements
- Usecases
- Object-oriented analysis } Design, handling complexity, modularization
- Software architeakture
- Software components
- Enterprise application: persistence patterns } Detailed design
- Web applications
- Security / Real-time design
- Clean Code } Implementation
- Dependability

Brook's law: "Adding manpower to a late software projekt makes it later."

Boehm's law: "Errors are more frequent during requirements and design."

Dijkstra: "Testing shows the presence of bugs, not their absence"

(one of) Lehman's Law (s): "A system that is used will be changed."

Parna's law: "Only what is hidden can be changed without risk."

Another Law of Lehman: "An evolving system increases its complexity unless work is done to reduce it"

2. Software Development Processes

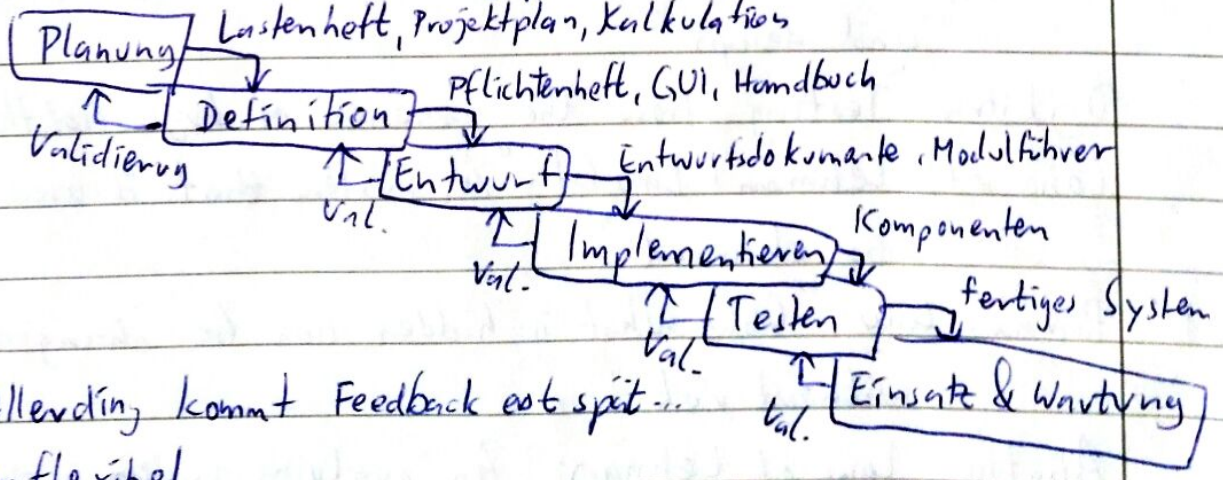
Software Vorgehensmodell ≙ abstrakte Repräsentation eines Softwareentwicklungsprozesses

Bestandteile: Aktivitäten / Rollen / Produkte (Dokumente / Artefakte)
optional: Techniken / Tools

Ohne diesen neigt man zum "Code & Fix" approach =>
- schlecht strukturierter Code
- Fehlende Dokumentation und fehlendes Design
- Unsystematische Erweiterungen / Implementierungen
- Fehlende Planung im Team => kein Teamwork
- keine Code-Wiederverwendung
- erschwerte Wartung / Fehlersuche

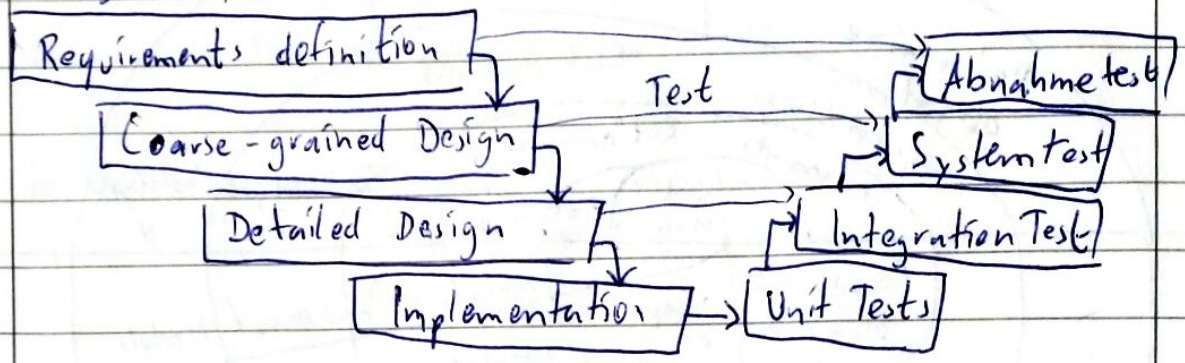
Was bringen Prozessmodelle?
- Verständnis für Code (da Übersicht & Struktur)
- erlauben Skalierbarkeit
- Code-Wiederverwendung
- Risikoabschätzungen für Kosten im frühen Stadium

#1 Wassenfallmodell:
+ teilt den Entwicklungsprozess in Phasen ein:



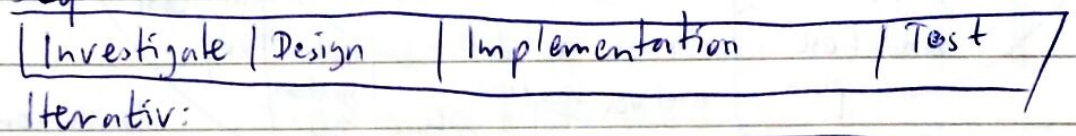
- Allerdings kommt Feedback oft spät...
- Unflexibel
- Wann sollte man zur nächsten Phase übergehen?
- gerade über lange Zeit schwer planbar

V modell: zeigt, wie Aktivitäten mit Artefakten in Relation stehen, ist aber kein Prozessmodell

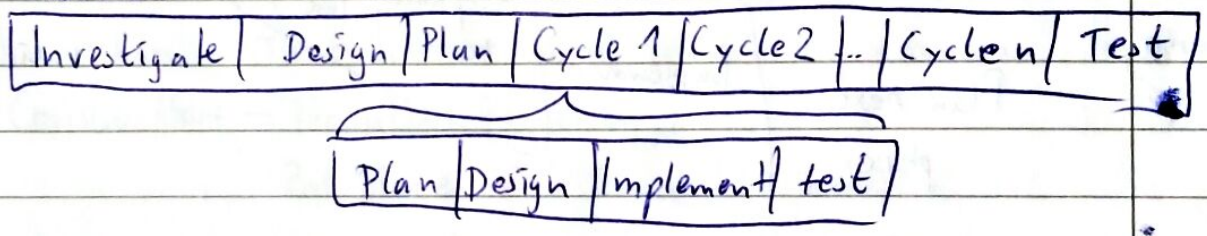


#2 Alternative Lebenszyklen: Iterative & inkrementelle Entwicklung

Sequentiell:



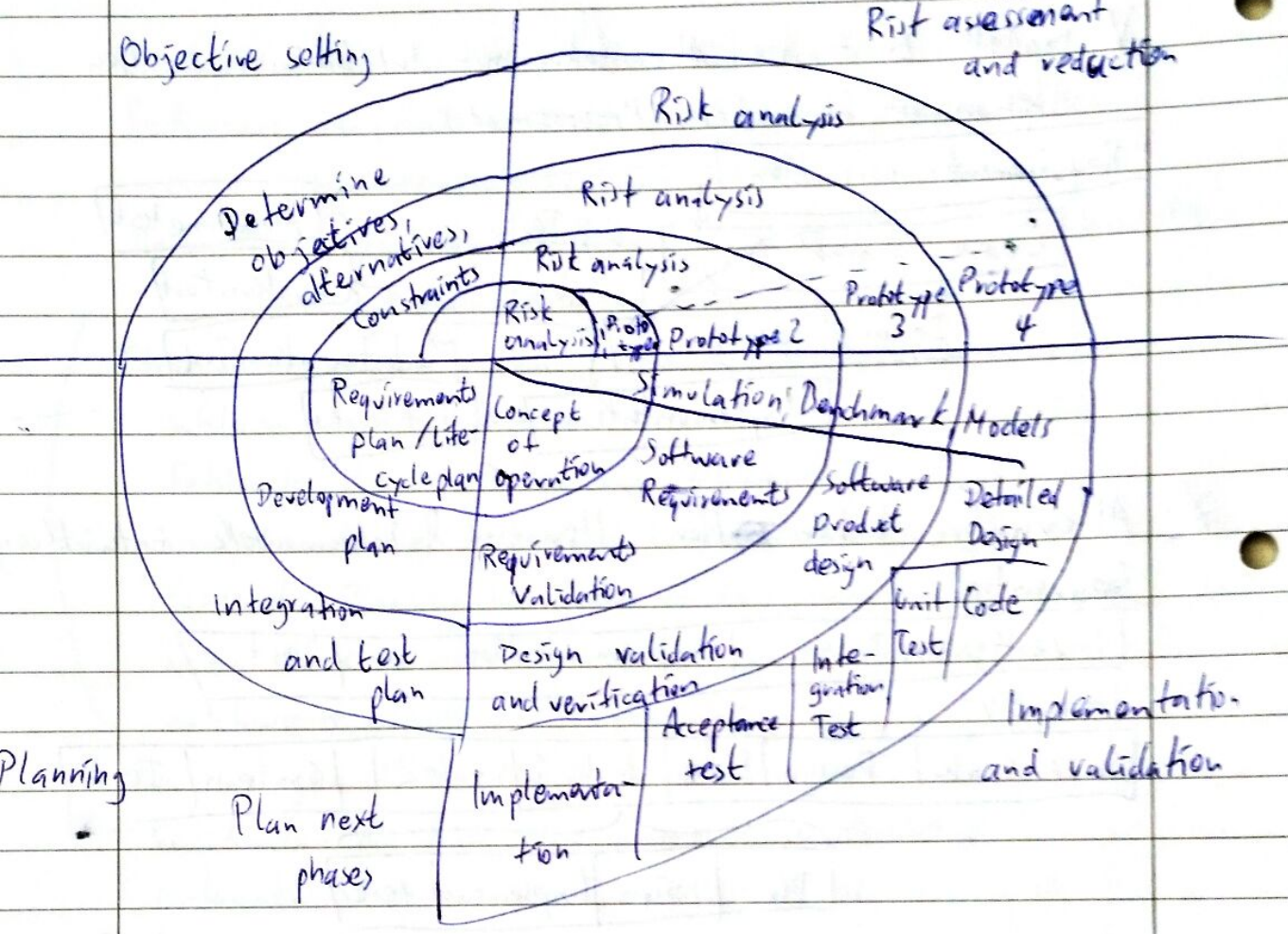
Iterativ:



Mehrere Wasserfälle hintereinander, kürzere Feedbackzeiten
⇒ weniger Risiko ("continuous delivery")
Prototypen können erzeugt werden

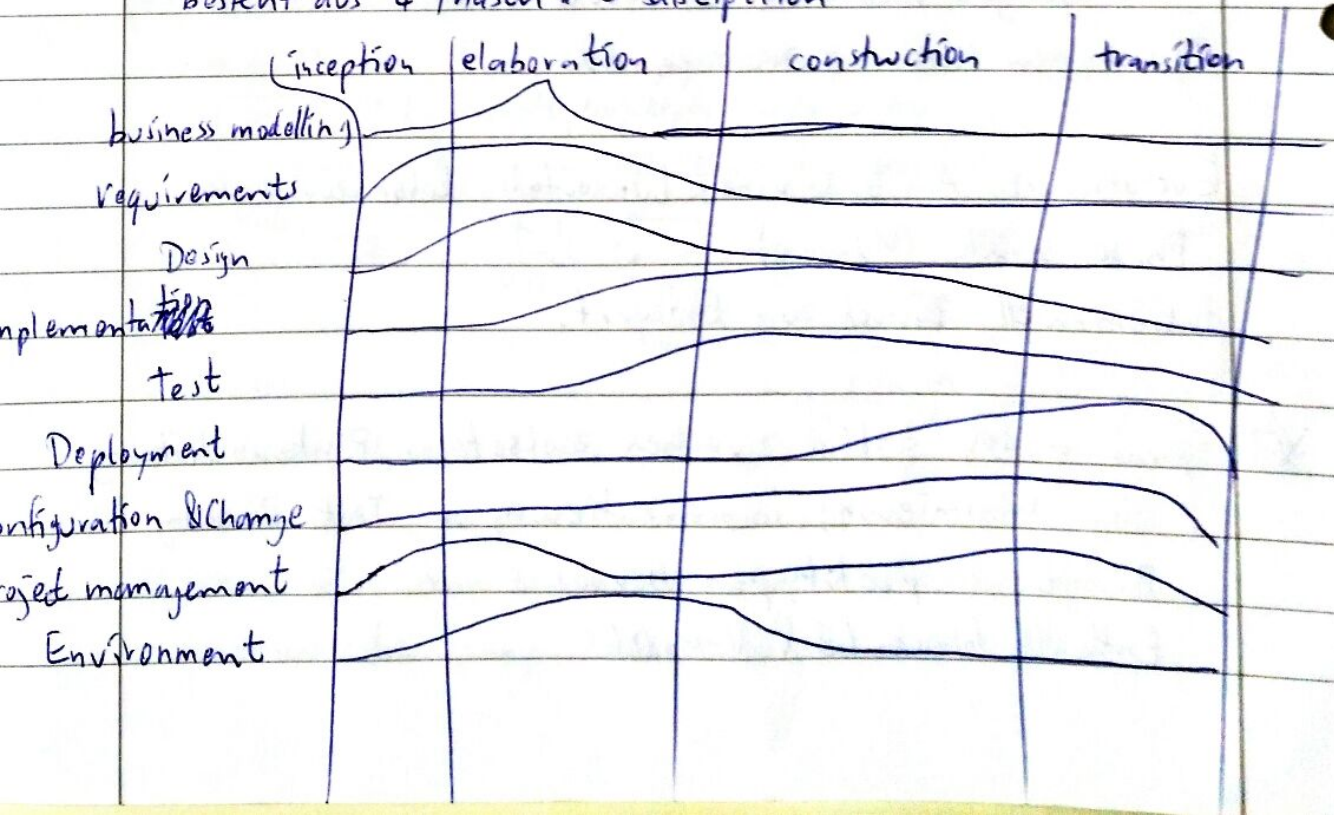
Evolutionär: Am Ende eines Wasserfalls, kehre zur ersten Phase zurück (Planung).
Inkrementell: zurück zum Entwurf.

#3 Spiral model: Zyklus zwischen Zielsetzung, Risikoabschätzung und Minimierung, Implementierung und Test, Planung.
Bringt n Prototypen heraus
Enthält Wasserfall & V-Modell



#4 Unified Process

besteht aus 4 Phasen & 9 Disziplinen:



UP ist iterativ & inkrementell, risk-driven, client-driven & architeture centric
multiple roles per person

Inception phase: - Vision des Businesscases
- Umsetzbarkeit (Feasability)
- Selbst bauen oder Kaufen?
- Kostenabschätzung
- Projekt stoppen?

Elaboration: - Core aufbauen (Architektur)
- resolve high risk elements
- define most requirements
- estimate overall schedule & resources

Construction: - iteratives Vorgehen
- Reduzieren der Entwicklungskosten
- implementieren kleinerer Elemente
- Vorbereitung zur Entwicklung

Transition: - beta Test & deployment

Disziplinen:
- Business Modelling: domain concepts, business processes
- Requirements: elicitation, analysis, documentation, validation, management
- Design: object-oriented analysis, model-software-classes

Die anderen haben keine strikten Regeln, was zu tun ist.

#5 Rational Unified Process (RUP) (wie?)
spezielle Implementierung des UP, fügt Aktivitäten,
(was?) Artefakte und Rollen hinzu.
Auch „best-practices" (was?): 1. Develop software iteratively
2. Manage requirements
3. Use component-based-architectures
4. Model visually (UML)
5. Verify software quality
6. Control changes done to Software

3. Agile

Herkömmliche Prozesse sind zu schwergewichtig, Zeit- und Geldruck verlangen ein schnelles und flexibles Vorgehen
Die Softwareprodukte sind bereits kompliziert genug, als dass man noch ein kompliziertes Vorgehen dabei nutzen möchte.
Agile Manifesto:

Individuals & Interactions	>	Processes & tools
Working Software	>	Comprehensive documentation
Customer collaboration	>	Contract negotiation
Responding to change	>	Following a plan

Can't plan ahead anymore, because Requirements might change anytime, unpredictable complexity
Continuous monitoring of customer needs & product quality

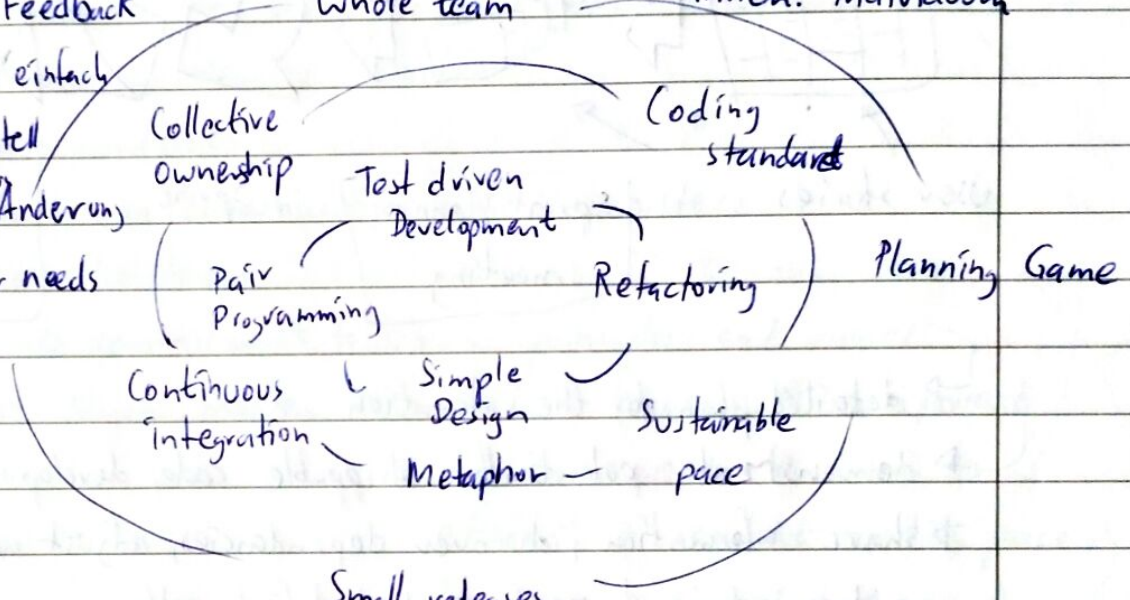
#1. Xtreme Programming (XP)
set of values, principles & practices focusing on implementation

doesn't deal with architecture design
Main values: communication, simplicity, feedback, courage

Prinzipien:
schnelle Lieferung
rapides Feedback
halte es einfach
inkrementell
offen für Änderung
Customer needs

XP Practices

Außen: Projektwelt
Mitte: Teamspezifisch
Innen: Individuum

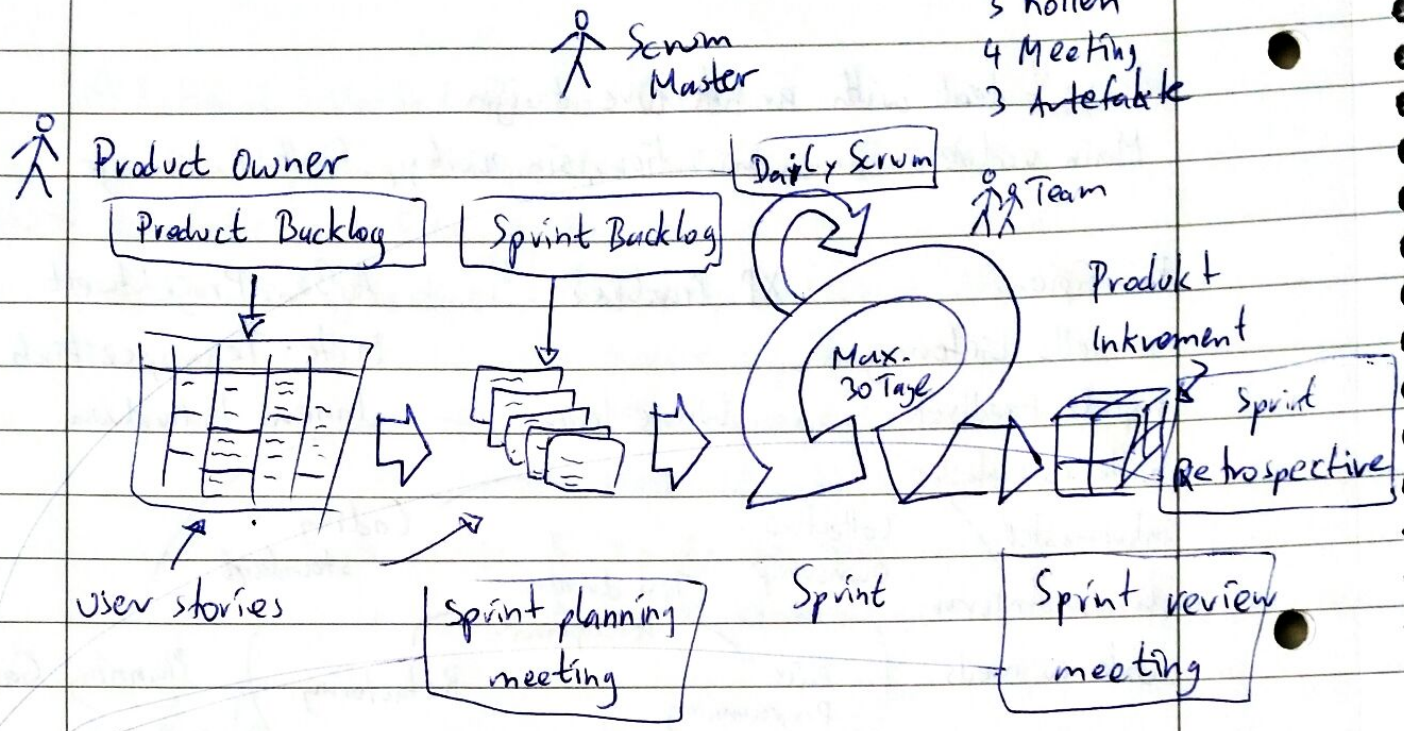


Kritik: skaliert schlecht, da keine Architektur & Spezifikation
Ad-hoc Prozess ⇒ nicht replizierbar
Fehlende Dokumentation
Nutzer muss kooperieren
Einige Praktiken (pair-programming, test-driven-development) sind nicht genügen getestet worden

#2 Scrum

agile management framework
Deliver Software in short sprints
Values of agile manifesto
No difference between development of new software or altering an existing one.

3 Rollen
4 Meeting
3 ~~Artefakte~~



- detailed plan for the iteration
- demonstrate potentially shippable code, developed during sprint
- share information, discover dependencies, adjust work plan, address individual needs and identified problems
- assesses the work in sprint, identifies good and bad practices

Sprint: implementation focused
short & consistent time frame ⇒ fast feedback,
split work into small tasks
no changes on product backlog

Roles: Pig roles: 1. Product owner:
responsible for product, decides what will be delivered
represents the customers interest
helps team to clarify questions
changes features and priorities before each sprint
accepts / declines the results
release management

common mistakes:
often unavailable
not enough power / knowledge
more than 1 product owners

2. Scrum Master
responsible for the process, solves problems
resolves problems / obstacles
buffers conflicts
supports communication with stakeholders
protects the team
secures commitment to scrum principles and values
Good scrum master <=> working process transparent & continuous ∧ team is efficient ∧ constant working pace ∧ teams support product owner ∧ team reports problems & obstacles

3. Team
small group of programmers, implement product increment, self-organized, help each other, have the same goal, have no predefined roles (difficult)

Chicken roles: {Stakeholders}, managers, customers, ...

Product Backlog
Collection of requirements priorized by busines-value, risk + impact.
~~can be changed anytime~~
Requirements are captured as user stories.

no activities. They belong to sprint-backlog
they can be changed anytime and are completed when the software project is finished
Story points measure expected effort
start with 2-3 requirements and refine later

Project planning
Scrum doesn't utilize a release or project plan, unrealistic to plan ahead a long time
estimations for costs and duration are necessary
Scrum release planning is based on the status of the product backlog & the team's development velocity
finer: sprint planning
finest: workday planning

Sprint backlog
all user stories the team has committed to in sprint meeting
taken from the product backlog
updated daily
- to do
- in progress
- done

new arising user stories have to be added to the product backlog, not to the sprint backlog
filled in sprint planning meeting
to do that: understand next steps in software development and choose wisely.

don't overload the team. ~85% is a good way to go.
Velocity ≙ average of done user stories
can be negative, if new user stories arise

Critical Evaluation
people / roles: need to be commited
Artefacts: code & tests
Documentation: needs to be demanded
Activities: no architectural design
Scalability: promblem with communication
Architeture: via refactorings, a practice: sprint 0 for arch
Quality critical software: no seperate developer / tester team, no stress-tests, only unit-tests
Initial Adaption: difficult, need discipline ⇒ might be changed inappropriately

Hints for LARGE projects
- start small & grow organically
- minimiz team dependency
- meet for daily scrum of scrums

Brook's law: adding people makes things slower!

distributed projects
- never seperate team and scrum master
- distribute team stepwise
- exchange delegates (to meet the other teams)

No silver bullet
initial adaption: participants need to learn new rules and behavior, often scrum is changed instead of the habits of the person.
traditional behaviors are conserved

4 Requirements Engineering

Requirements are something that is wanted or needed, formalized in natural language.
They need to be: adequate (hinreichend, angemessen)
complete (vollständig)
consistent (widerspruchsfrei)
understandable
unambiguous (nicht falsch interpretierbar)
verifiable (überprüfbar, ob eingehalten)
suitable for the risk

three kinds of requirements:
1. Functional requirements
2. Quality "
3. Constraints

Requirements are start & end point of a software project
Wenn hier etwas nicht stimmt, kann es teuer werden oder es geht sogar alles in die Hose

Requirements engineering process
Cooperative, incremental, iterative process of:
Elicitation, Documentation, Agreement, Validation & Management
↳ Seeking, capturing, consolidating requirements from available sources
↳ Persist the elicitated ~, create software ~ specification (SRS)
↳ resolve conflicts, find ~ that are acceptable for most stakeholders
determine and comprehend all relevant ~ in necessary degree of detail
achieve acknoledgement of ~ by involved stakeholder

Stakeholder ≙ a person / organisation that (in-) directly influences the r of a system

i.e.: users / customers
operators
purchaser / sponsor / controller
developper
software Architect
tester

missing stakeholder may lead to missing requirement!

Requirements Engineer (business analyst)
translates between users and developpers
need methodological skillz:
thinks analytically,
has empathy
has communication skillz
is conflict solving
" self-confident
" convincing
has discussion / moderation skillzz
is responsible for elicitation, documentation, agreement and maintainance

Techniques for Requirement elicitation
Questioning: Interview, Questionnaires, On-site-customer
Creativity: Brainstorming, changing perspective, analogy
Retrospective: System archaeology, reuse, competing systems
Observation: field observation, apprenticing

Supporting actions for elicitation:
mind-maps, Workshops, CRC-Cards, Audio & Picture recording, Use-Case-Modelling, User stories, personas, prototypes

Classification of Requirements
helps to identify " :
was this R stated, because we need to specify...

... some of the system's behavior, data, input, or reaction to stimuli, regardless of the way this is done?	functional
... a quality that the system or a component shall have?	quality
... any other restriction about what the system shall do, how it shall do it, or any prescribed solution or solution element?	constraint

More facets: satisfaction: hard / soft
role: prescriptive / normative / assumptive
representation: operational / quantitive / qualitative / declarative

prescriptive → system
normative → environment
assumptive → user

operational → operation or data specification
quantitative → measurable size "
qualitative → " of goals
declarative → description of a required feature

Basic writing recommandations:
- short sentence with one clear requirement
- make clear who is responsible for what
- avoid weak words (like "good", "effective"...)
- maintain glossary of items
- use templates

1. The system — must / shall / will — be able — whom? give the possibility to... — objects & extensions — process word

OR

2. (Trigger) [Actor] [Action] [Object] (Condition)

⇒ usually results in low-level feature list

Modern Requirements capture
user stories (agile) or use cases (model-based)
focus on interaction of user and software (put requirements in context of goals)
In non-user related requirements use feature driven approach

Requirement validation (← expensive if not done properly)
output of requirement artefacts: ambiguity, incompleteness, inconsistencies
validate input appropiate to context
are all stakeholders involved?
no formal verification possible! require models, reviews, prototypes...!

Validation techniques
Inspection, Walkthrough, Review
Simulation

Prototyping
Creation of system cases
Model checking

Cost & Benefit Considerations
Cost for removing errors depends on how long it stays in the software
Specification of requirements costs (time and money)
find most economic tradeoff.
(Boehm's law)

5 Use Cases

In general: textual notation to capture an interaction of (typically) two parties
→ a story of how a system is used by a user which can be another system
therefore focus on operational requirements
Best practice for requirement elicitation
Text first, diagrams only for illustration
Just a notation, can be used for other flows
UML diagrams show relations between them

Scope & Context
Goals and actors depend on the scope
Scope ≙ border between system & environment
Context ≙ part of environment relevant for a use case
Context Boundary ≙ seperation from system and

irrelevant environment

Black box scope ≙ don't know about the inside functions of a system

White box ≠ black box scope

Business use case has enterprise as its design scope

System use " " computer system as its " "

Component " " subsystem " " " "

Good use case ≙ adequate level of abstraction. Emphasis on goal to be achieved

EBP ≙ elementary business process: <[a task], [performed by one person/system], [in response to a business event], [which adds a measurable business value], [and leaves data in a consistent state]>

How to determine: boss test / coffe break test / size test

Use Case levels:

White → blue → indigo → system operation

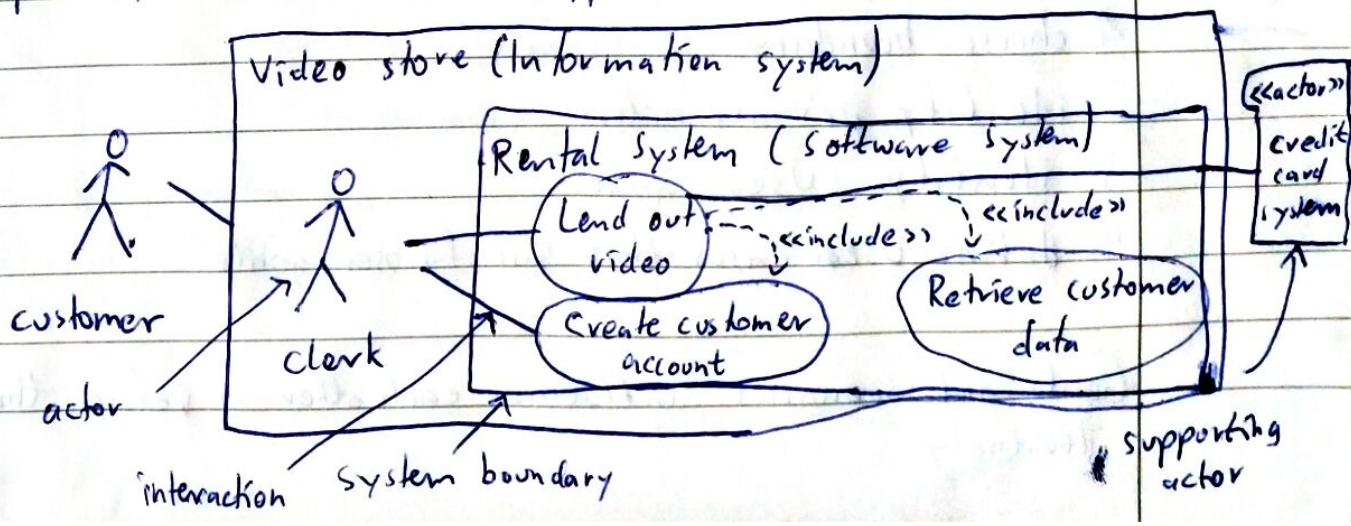
Summary goal EBP Subgoals

«include» ≙ sub use case call

«extend» ≙ callback of an interrupt

Use Case diagrams

primary actors on the left, support actors on the right



textual description is more important than diagrams, only create diagrams for user goal level use cases
differentiate between computer/human interactions

Terminology

Stakeholder ≙ person/organisation with interest in system
actor ≙ entity (person/system) with behavior outside the SuD (System under discussion)
primary actor initiates interaction (with software system)

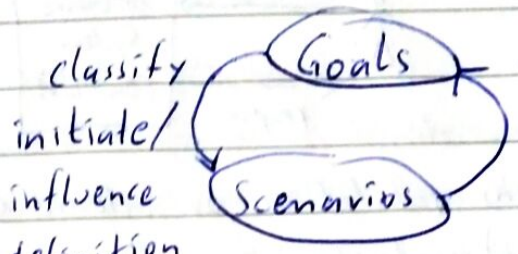
Use case model ≙ set of all use cases and related diagrams (may also include activity diagrams to illustrate flows)

Scenario ≙ specific sequence of (inter)actions between actor and SuD
aka use case instance, one particular story of using the system

How to find use cases

1. choose boundary
2. identify primary actors
3. identify user goals
4. define usem cases that satisfy user goals

Goals and scenarios influence each other => refine them iteratively



classify initiate/influence definition

concretise lead to refinement/identification/revision

"Living" until approved by stakeholders changed and emerged permanently

Breadth first!

Feasible precision levels:

1. primary actor's goal and name
2. use case briefs // main success scenario
3. add extention conditions (failure conditions)
4. add " handling steps (" handling) + more

Example:

Actor and Goal: As a cashier I would like to process sales in order to earn money for the store

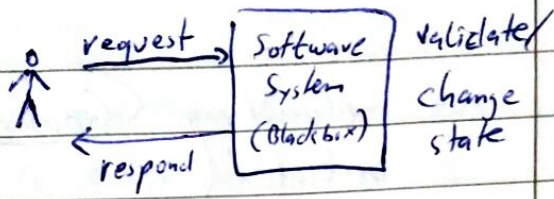
Main success scenario: The cashier enters item identifiers into the system. It shows and calculates running total. On finish, taxes are calculated, bill is printed. Payment by card or physical. System stores data of sale and updates inventory, prints receipt.

Failure conditions: - item identifier may be incorrectly recognized
- cashier removes item
- " canceles process
- customer choose to pay with credit card

Fairure handling ...

USE TEMPLATES!

How to write use cases



Steps describe interactions, validation or internal change
each step shows a (sub) goal succeeding => process moves forward
" " captures actor's intend
Exclude UI interaction details
Data description precision levels:
1. Data nickname
2. Data fields (with type) associated with nickname
3. Field types, lengths, validations
No conditions (if-statements) => use multiple scenarios or define in extensions section (3a. System signals invalid identifier
→ 1. Signal error and reject)

Fully dressed
1. Einleitung (preface) elements : scope, goal level, primary actor
2. Stakeholder & interest list
3. Preconditions
4. Postconditions
5. Main success scenario (typical success path)
6. Extensions (describe all other branches to success or failure)
7. special requirements (non-functional requirements, quality attributes & constraints)
8. Technology & data variations (describes how things get done)

Requirement Specification
- complete description of the external behavior of the Software
- documentation of all interfaces between software & environment
- Structure of ": Introduction: purpose, scope, stakeholders, definitions, acronyms, abbreviations, references, overview
Description: system environment, architectural descr., system functionality, users and audience, constraints, assumptions
Requirements: Functional, quality, ...
Appendix
Index

Management
requirements hosted in special repository (software also special) are linked and contain metadata, extended attributes like author, date, version, ID, name, criticality, priority, stability, status, type, infos, efforts, release → traceability

6 Requirement analysis

Analysis ≙ investigation of a problem, not a solution
Design ≙ conceptual solution, not an implementation

Operation Contracts

starting point for system design, since they provide a

more detailed description of it.
they are expressed in a declarative state change fashion
"what" > "how"
helpful for system design that has to deal with the "how"
free use cases from becoming overly verbose
Once system operations are identified, you might want to
~~the~~ describe them more in detail
=> use cases + detail => operation contracts
describe the outcome of the state change in the domain model

Aussehen (Schema eines OC):

Operation: Name, Parameterliste
Cross-References: (optional) andere use cases, die in dieser Operation auftreten können
Preconditions: erwähnenswerte Annahmen über den Zustand des Systems, bevor die Operation ausgeführt werden kann
Postconditions: state of the objects in the domain model after the operation is completed
past tense!

Contracts vs. use cases

Use cases ≙ main repository for requirements, details
contracts ⇑ useful when complexity underlying, make them difficult to read
or unexperienced people onboard
post condition syntax allows good analysis

they help to understand potential relationships in a domain
initial best guess, will never complete, enhance the domain model

Contract Creation summary

1. Identify system operations from use cases / sequence diagrams
2. construct a contract for system operations that are complex or unclear in the use case
3. Use the following categories to describe postconditions:
   - instance creation / deletion
   - attribute modification
   - associations formed / broken (eg. sales/Item to a sale process)

! model system just good enough

When are Contracts useful?
→ when use cases can't capture details & complexity of required state change
don't write them, if a use case itself is easy enought to understand

Contracts & UML

UML specifies operations having a contract. Use natural language or OCL (Object Constraint Language) for description.

7 Software Architecture

Advantages of explicit architecture
- Stakeholder communication
- System analysis
- Large scale reuse
- Project planning

What is a software architecture?
- A set of decisions, which are hard to revert & have to be made in the early development process
- Link between specification & design
- record why a decision has been made
- DB for example belong to the deployment architecture, not to the logical architecture

Reussner: "Software architecture ≙ result of a set of design decisions comprising (beinhalten) the structure of the system
  - with components and
  - their relationships and
  - their mapping to the environment"

Views, view points

View ≙ representation of a coherent set of architectural elements. ⇒ subset of an architecture
Structure ≙ set of architectural elements itself ⇒ architecture

view point ≙ group of views to the same concern
Every software has a structure, that might not have been derived from an arch.
Factors influencint the architecture:
Requirements (quality & constraints)
Re-usability (systems in the same domain, subsystems, components, style, patterns, guidelines...)
Organisation (Conway's law "A system (usually) reflects the organizational structure that built it."
Team size, team number, experience, organisation structure (distributed...))
The most important quality requirements might influence the architecture (like availability ⇒ redundant components...)

Design Principles

Seperation of concerns: minimize coupling, maximize cohesion
Single responsibility principle: one resp per module/component
Information hiding: Parnas "only what is hidden can be changed without risk"
Principle of least knowledge: Don't talk to strangers (Law of Demeter)
Don't repeat yourself!
Minimize upfront design: YAGNI, refactor if needed

Terminology
Architectural pattern: solution to recurring problem @ arch. level
" Style: solution principles (modular / OO) independent from application. Should be used systemwide, or at least among these areas: Communication, Deployment, Structure
Reference Architecture: Defines domain concepts, components,

subsystems that can be used by specific instances (templates)

Architectural Styles:

Layered: highe layers call those below.
reduces accidental complexity, implements separation of concerns, improves modifiability, simplified testing, independent exchengeability
Drawbacks: increase amount of classes, need façades or data transfer objects

Architectural Patterns:
Domain Model: an object model of the domain that incorporates both behavior and data.
+ Facilitates OO thinking, organizes complex domain functionality in a natural way
- Data persistence more komplex

DTO ≙ data transfer object, which carries data between processes or architectural elements.
+ reduces method calls, if data is complex & distributed
may contain a method for serialization
+ reduces coupling

MVC ≙ model view controller
Model @ domain layer
contains domain model & business logic
very application specific

problem: DB not OO

Modelling alternatives:

1. one controller per use case (for systems with many use cases)
2. " " an class per app / system (for small systems)
3. direct access to appropiate domain objects.

   reduces passing through parameters but brings control flow logic easily into domain model

View @ UI Layer (presentation)

presentation of data

managing interaction with user

event handlers forward UI events to application façade

Controller @ application layer

distributes the incoming requests

remembers session state

controls workflow

implements system operations

Façade

@ head of a subsystem. wrap up its functionality and offer few simple methods to upper layers

reduce coupling. Java packages have no interface by default. build dedicated class for it ≙ ∩

=> information hiding, manage complexity

Observer Pattern

notifies all objects that are from an object, if it changes its values.

8 Software Components

- are building blocks for software
- reusability through modularity and replaceability
- loosely coupled, allow abstraction, expose interfaces
- can be contractually (vertraglich) specified and deployed without understanding its internals
- effort for building should be as low as possible

Objects != Components (due to inheritance...)
Components in Java? → use façades (interfaces)

Location of Components
- Feature oriented, usually @ application & domain layer
- infrastructure may also form a component (JDBC)

Pattern: Service Layer
- offer a set of available operations to the underlying Layers (domain model, data Source Layer) and coordinate responses. ⇒ encapsulates other components

Component model, a standard for cross-platform components
- defines: what is a component?
  - how does it offer services?
  - how are they composed/connected?
  - how do they communicate?
  - Where can they be found?
- often offering a component framework, where they can be executed

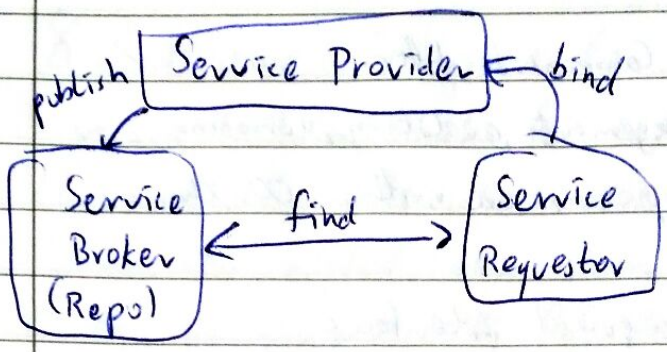
Technical realization of Components differ
differences: memory management, evolution, versioning
⇒ problems: many approaches are rather OO, than component based
no proper explicit required interfaces
similarities: late binding, encapsulation, interface inheritance

Open Service Gateway Initiative (OSGi)
OO ⇒ many fine granular objects but no real concept above packages
OSGi offers bundles as a solution
(= JAR-File defining a public interface via manifest)
OSGi provides a library/registry for alive services/bundles

Web Services ≙ a self-contained, self describing & modular application that can be reached via and is located in the web
self-contained ≙ exposes functionality & attributes in a public interface, if implementation is hidden
self-describing ≙ machines can read a provided description
modular ≙ reusable and can be composed to sth bigger
published ≙ can be registered to the "yellow pages of the web"
located ≙ has a fixed URL
invoked ≙ reachable by standard web protocol

Service oriented Architecture (SOA)
for ~~services~~ built from web services
systems

CBSE = Component Based Software Engineering!



publish: advertise service description to repo
find: receive them
bind: locate & invoke a service

Palladio Component Model (PCM)

Designed to enable early performance predictions of software architectures.
- Domain specific modeling Languages (DSL)
- prediction of quality properties on model base → systematic design of systems
- faster than real execution
- keeps back black box principle of components
- derive performance metrics from the models using analytical techniques and simulation

Influencing factors: user demand / input
hardware
code performance
assembly with other components
} explicit in PCM

Input for Palladio performance prediction:
- Component models (how the used components are written)
- Composition model (how they are composed)
- Deployment model (hardware running the system)
- Usage model (#users, #demands ...)

Outputs: Response Time, Resource utilization, throughput

Models and their creators:
Component model: Component developers
System design model: Software architekt
Deployment: System Deployer
Usage model: Domain Expert

Component Developer Tasks:
- specifies components & interfaces
- specifies data types
- builds composite components
- describes component behavior in parametrised service effect specification (SEFF)
- resource demand expressed in abstract units (file size...)
- explicit modelling of external service calls, binding to interfaces only

Software architect's tasks:
- specifies an architecture (system model) from existing components and interfaces
- specifies new components & interfaces
- uses architectural styles and patterns
- analyses architectural specification and makes design decisions
- guides performence prediction on architectural specification
- delegates implementation tasks to component developer
- guides whole development (process)

System deployer's Tasks
- models resource environment, abstract specification of resources (SEFF)
- models allocation of components to resources

Domain Expert's Tasks
- model user behavior on the outermost API (#users, #requests, input parameters ...)

Composing the models

Architect + Developer	System Deployer	Domain Expert
⇒ Assembly Context	⇒ Allocation Context	⇒ Usage Context
Horizontal Composition: binding other components	Component, Container	Usage @ system boundaries:
Vertical Composition: Encapsulation in Composite Components	Communication	#users, #inputs, request probabilities, parameter value
	Security, ~~Safety~~, Concurrency	
Tool output: Behavior of the whole system: overall SEFF	Tool output: Allocation dependent QoS characteristics: Timing values for resource demands, failure probability	Tool output: Branch probabilities, #loop iterations, output parameters, usage dependent resource demands

Patterns of enterprise application architecture & microservices

What is an enterprise application?
- application that supports business processes and that processes business transactions such as web shop, leasing management, expense tracking, shipping tracking, online shop, patient record management

Properties of EA
- persistent data
- large amount of data
- concurrent data access (# users high)
- different user interface screens
- interface to other systems
- complex business (often illogical)

Layers of EA

Front end, Presentation	Domain, Middle Business	Data Source
UI	calculation, load store	DB access

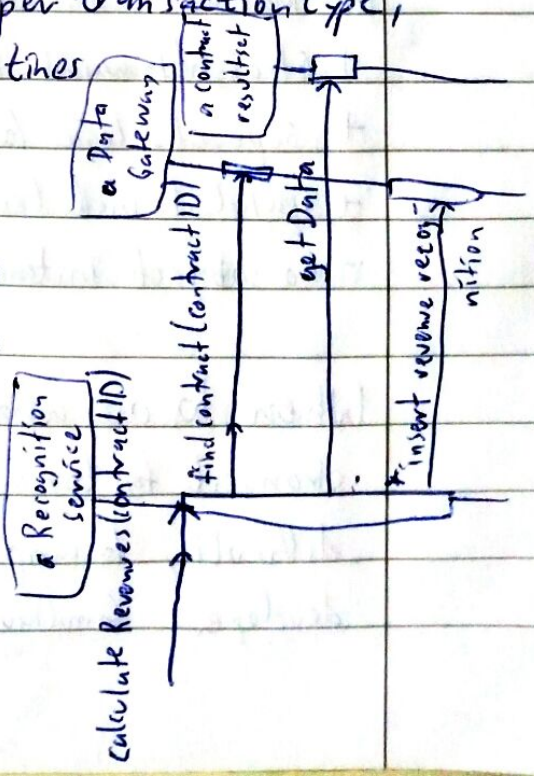
Domain logic Patterns

Challenge: representation of business logic, manage high complexity and connect UI with database

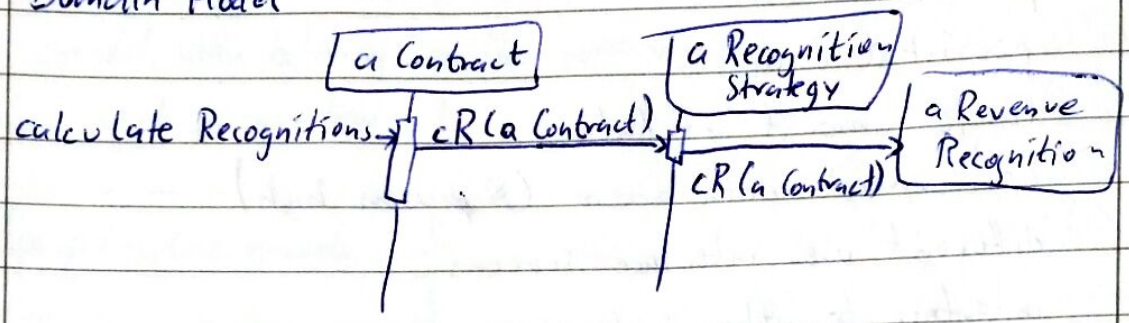
The choice of a pattern always comes with the requirements

Patterns are just a starting point, no complete solution.

Transaction script: single procedure per transaction type, factor common behavior into subroutines
- not OO
- + simple
- + easy to connect to simple data sources
- + transaction boundaries " to determine
- − doesn't scale well w/ complex logic
- − tends to have duplicate code then



Domain Model



An object model of the domain that incorporates both behavior and data

- objects collaborate to do transition
- OO thinking

+ organizes (complex) domain logic better
+ no code duplications
- mapping data to source more complex (DB)
- steep learning curve if not familiar with OO

Table Module

A single instance of a class that handles the business logic for all rows in a database table or view

+ Straightforward mapping to data
+ seperates logic for different concepts
+ useful if used technology supports it (COM, .NET)
- no object instances: can be hard for complex logic

When to use what?

strongest factor: domain logic
difficulty to map to data source
developers familiar with domain models?

What tools are used?
anyway, it is possible to combine all the three

DATA SOURCE ARCHITECTURAL PATTERNS

Main question: how to map Domain Logic (Objects like Person, attributes age, size, name...) to data base (with rows and columns)?

OO ≠ DB (inheritance, aggregations, references...)

Record set

In-memory object, that looks like a result of a DB query, but can be easily generated and manipulated by all other parts of the system

- usually not self-built, but created by framework/platform
- OO representation of DB content

Table data gateway

An object acts as a gateway to the DB table. One instance handles all rows in the table

- simplest access to DB
- Aim: separation of SQL statements and code that deals with data

  ⇒ not very useful for domain model implementation, rather for transaction script

implements CRUD

Gateway or façade?
- Façade: written by service developer to simplify complex API
- Gateway: " " client to encapsulate access to external system

Active Record
An object that wraps one row in a database table or view, encapsulates the database access, and adds domain logic on to that data
OO approach: bring data & functionality together.
provides an intuitive concept to represent data
⇒ make all person objects know how to read and write their data from and to the database
Good choice for very simple domain logic
DB schema and active record need to map 1:1
Complex business logic with object relationships & inheritance does not work well with Active Records
use Data Mapper instead

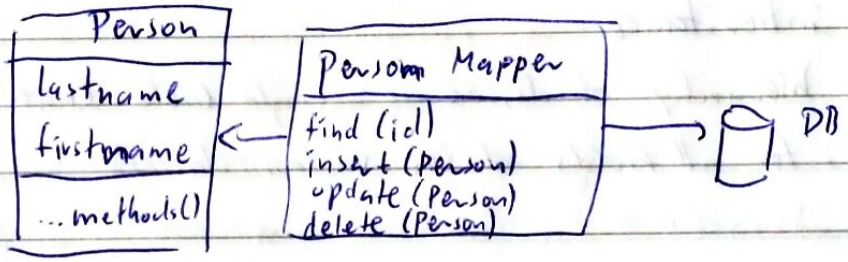
ROW DATA GATEWAY
Like data gateway, but there is one instance per row
- in memory object tied to DB
- DB usually has "finder" - ~~method~~ class
- can be automatically generated ⇒ whole DB access can be built "
- separates DB access and domain logic ⇒ DB change possible

IDENTITY MAP

ensure that each object gets loaded only once by keeping every loaded object in a map.
Look up objects using the map when referring to them

DATA MAPPER

A layer of mappers that moves data between objects and DB while keeping them independent from each other and their mapper
Isolates in-memory objects from DB



Also called DAO (Data access object)
Mapping via explicit code, can also be generated for complex logic
When domain model is used, ignore DB schema and try generating code for DB accesses

When to use what?

Transaction script → Row data Gateway. Explicit Interface, better to evolve
Table data Gateway, if record set Framework
Domain Model → Simple: active Record
Complex: data Mapper
not Gateways. They couple model too tightly to DB Schema

Table Module → Table Data Gateway. Especially if using record set framework. Possible to combine all 3.

Object-Relational structural Patterns

Idea: map OO structure (with inheritance, ...) to DB relational structure

Single table inheritance
Inheritance hierarchy of classes as a single table that has columns for all fields of the various classes.
+ easy!
+ no joins
+ only 1 table
+ moving fields in hierarchy doesn't change DB
- many unused fields in DB
- big table
- bottleneck

Class table inheritance
Inheritance hierarchy of class with one table for each class
+ all colums relevant for every row
+ easy to map legacy schemas
+ straightforward relation between domain model & DB scheme
- one object needs multiple tables
- moving fields change DB
- supertypes are bottlenecks
- hard to understand

Concrete table inheritance

Inheritance hierarchy of classes with one table per concrete class.

+ Tables self-contained, no irrelevant fields
+ concrete subclasses, no joins
+ each table accessed by one domain object
- refactoring harder
- operations on superclasses influence all subtypes
- find on superclass needs to check all subclasses

When to use what?

Tradeoff: data-duplication vs. speed

⇒ Mix ~~hierarchy~~ patterns in a hierarchy depending on how usual access is. Consider using commercial tool.

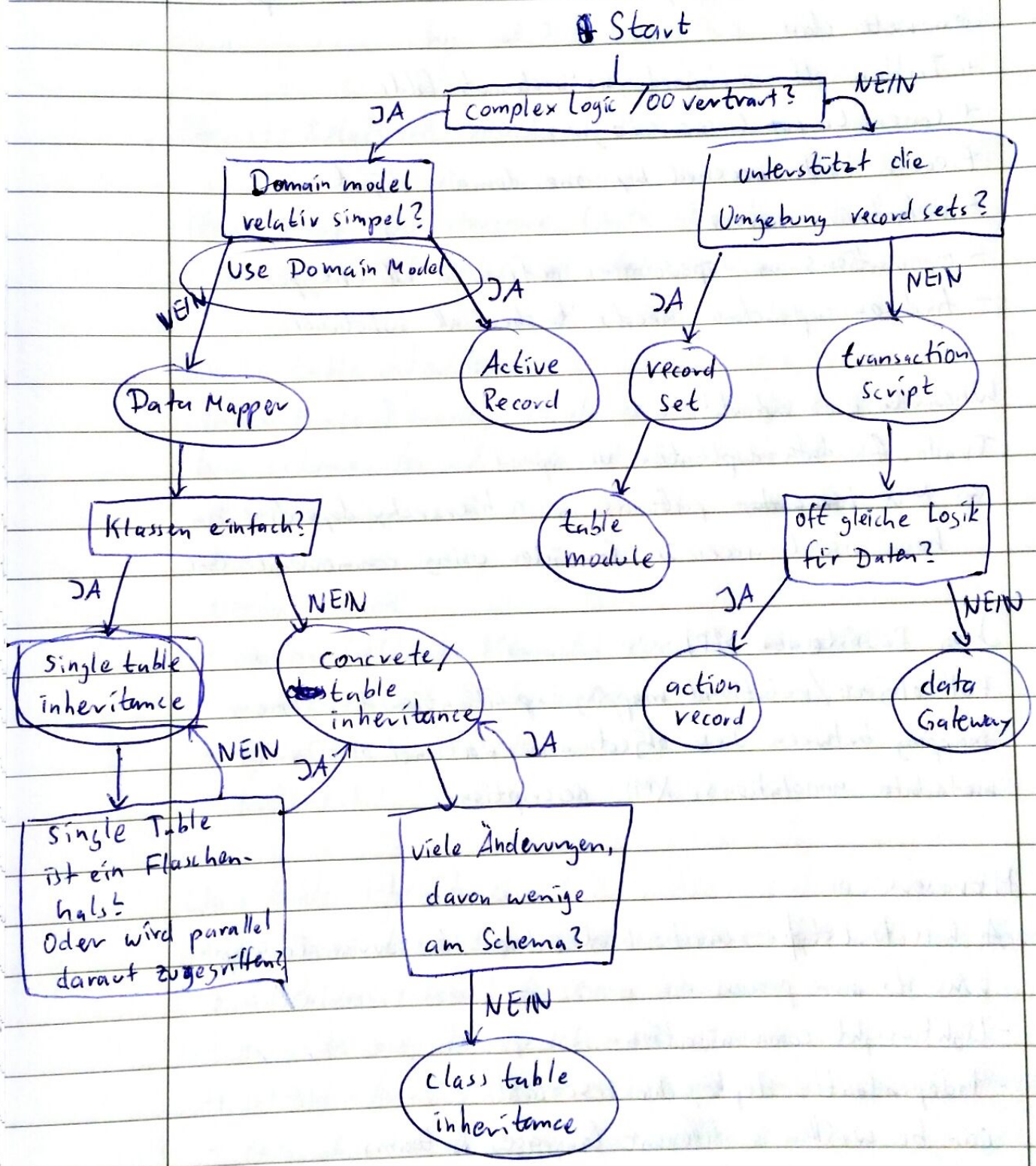
Java Persistance API

Full object/relational mapping specification to define mapping between Java objects and relational DB. Java metadata annotations, XML descriptions

Microservices

Architectural style, can be broken down into services. Each app runs its own process

- Lightweight communication
- Independently deployable + scalable
- can be written in different languages & teams
- design for failure
- decentralized data management & governance
- infrastructure automation
- evolutionary design


Start
complex Logic / OO vertraut?
JA
NEIN
Domain model relativ simpel?
unterstützt die Umgebung record sets?
Use Domain Model
JA
NEIN
JA
NEIN
Active Record
Data Mapper
record set
transaction script
Klassen einfach?
table module
oft gleiche Logik für Daten?
JA
NEIN
JA
NEIN
Single table inheritance
concrete table inheritance
action record
data Gateway
NEIN
JA
JA
Single Table ist ein Flaschenhals? Oder wird parallel darauf zugegriffen?
viele Änderungen, davon wenige am Schema?
NEIN
class table inheritance

10 Software Design

Domain ⟶ implementation ... but how?

Design model
A blueprint for implementation
Domain model is an artifact in Unified Process
Domain layer, where object design happens, is the layer with business logic in your application

Responsibility-driven designs
~ assign " to software objects
Type of ":
Doing ≙ initiating another objects doing calculation, coordinating activities
Knowing ≙ knowing about privat data, related objects, knowing about things that it can derive or calculate
They are implemented by methods, but are not necessarily the same as a method

Assigning responsibilities
useful to look @ operation contracts. Especially the post-conditions
find object interactions ↳ (to see what happens in the implementation)
document them in interaction diagrams
stake responsibilities clearly
guideline/pattern: information expert ⇒ find objects that have the required information

Design Diagrams

Interaction Diagram

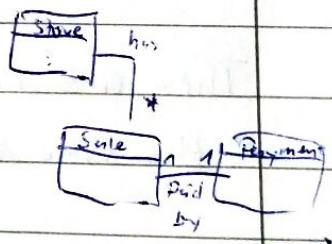
Valuable medium for exploring object designs, more important than class diagrams. Needed for creation:

1. System operations and their contracts
2. Domain model
3. architectural guidelines, design patterns

Design Class Diagram

Illustrates specifications for classes and interfaces.
In practice developed parallely to Interaction Diagram
Needed for creation:

1. Identif classes that participate in solution. NO 1:1 mapping of classes in design class Diagram to Domain Model
2. Draw class diagram for those classes
3. Identif class methods, distinguish with names.
   - create operation (constructor)
   - accessor / mutator (get/set)
   - multi-objects (collections)



GRAS Patterns (General Responsibility Assignment Software)

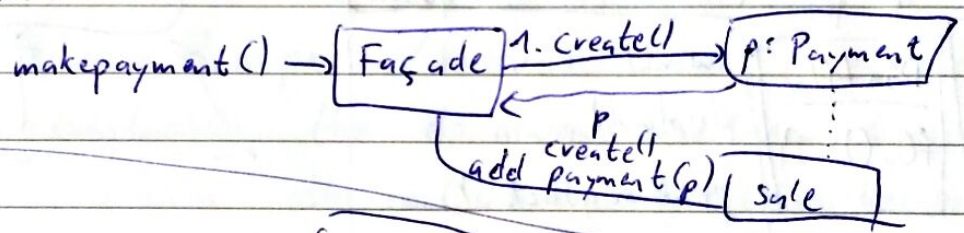
How to create interaction diagrams systematically
Creator ≙ What object is responsible for creation of X?
choose an object C such that:

- C contains or aggregates X
- C closely uses X
- C has the initialization data for X

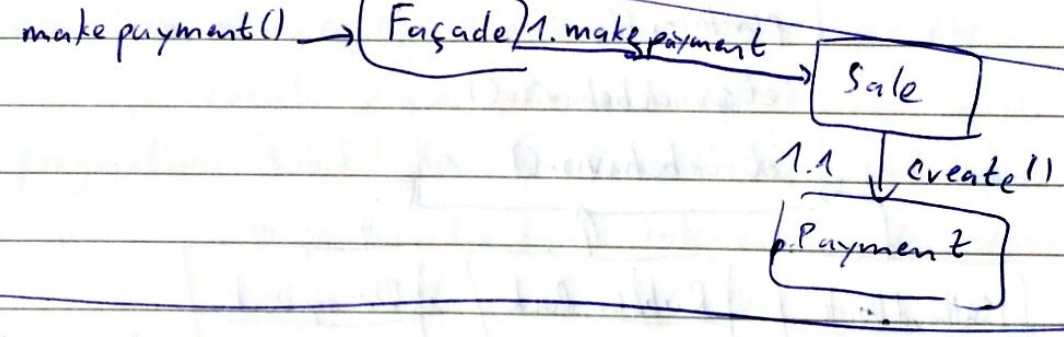
Controller ≙ have responsibility for receiving / handling system event messages
- Responsible for controlling the flow within a system operation
- One system façade controller or one per use case
- not MVC

Low coupling & high cohesion
to reduce the impact of changes.
low dependencies and distribute responsibilities
reduce simultaneous calls. sequential calls via other classes better:

BAD



GOOD



Polymorphism
How to handle alternatives based on type?
Use polymorphic method call to select behavior, instead of if (instanceof...)
makes it easier to add other behaviors later on
Use inheritance, abstract base class

Favour Composition over Inheritance

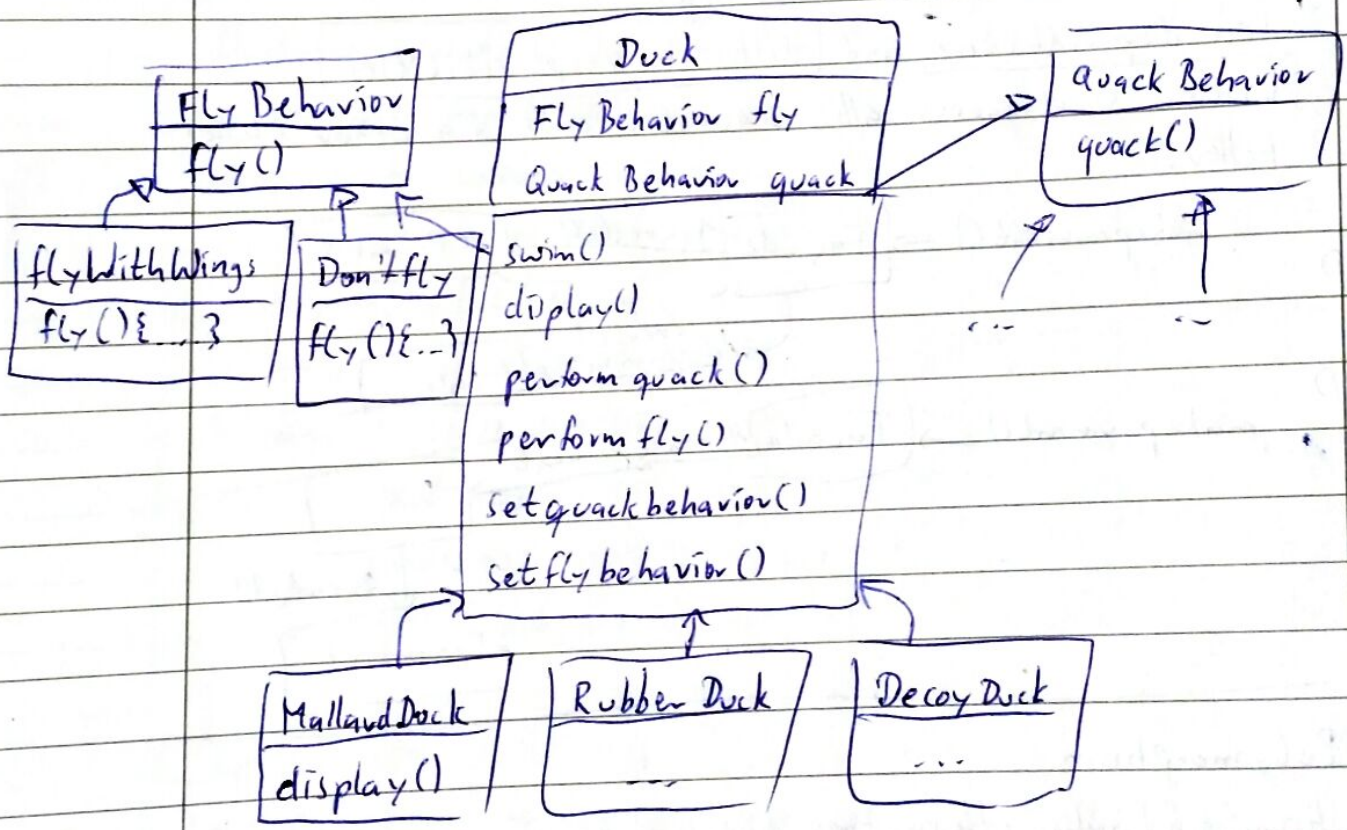
Inheritance exposes a subclass the details of its parent's implementation. It's often said, that inheritance breaks the concept of encapsulation.

Therefore: Composition!

Inheritance is white-box reuse (tighter coupling)

Composition is black-box "

You can use strategy pattern for composition on class level



Pure Fabrication

If no appropiate domain object for responsibility exists in the domain model, just invent it :)

Indirection

Create adapter object, if assigning responsibility brought high coupling / low cohesion.

Protected Variations

General principle for mechanisms to preserve objects and (sub)systems from varintions, e.g. information hiding, interfaces, polymorphism

- Law of Demeter (Don't talk to strangers!)
- Based on Parna's Law

11 Model Driven Development

Programming paradigms only reduce accidential complexity (caused by inappropiate development methods) not inherent ".

Models have 3 aspects / features:

- representation: they do always model something from the real world. might be another model
- reduction: only capture the relevant aspects for the creator / user of the model
- pragmatism: substitution function for particular subjects in particular time intervals, under restriction to particular mental / actual ops

Metamodel describe the structure of models (model a modell must have: Abstract syntax: elements, properties, relations are independent from context / representation

Concrete syntax: describes constructs, properties and relations that are specified in the abstract syntax

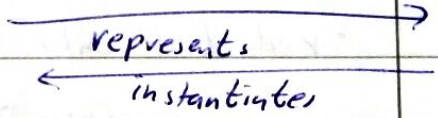
At least one concrete syntax must be specified for the abstracts syntax

Static semantics
modeling rules and restrictions which cannot be expressed in the abstract syntax. (OCL invariants)

```
context Locomotive
inv: self.series >= 100 and self.series <= 999
```

Dynamic ~~synt~~ semantics
describe the meaning of the constructs
often specified by natural language

Modeling layers
- Models represent originals
- if a model is prescriptive (created before original), the original instantiates the model
- includes (commonly) 4 layers (metamodel → model → model → orig)



Self-descriptive Models,
model their own syntax
} Top-Layer {meta}^{n} model

Model driven software development
MDE (Engineering)
- use domain specific modelling languages (DSML)
- transformation engines and generators
- reduction of platform complexity
- Code is also a model. Model of behavior and structure of the system. Code doesn't describe its deployment

Model driven vs. Model based

↳ model is primary artefact, explicitly specified, developed, versioned, etc.. analysis through model transformations

↳ secondary artefacts for documentation, communication

expected benefits from MDSD: cost reduction, short time to market, usage of domain knowledge in models

Model driven architecture (MDA)

Initiative of Open ~~Made~~ Management Group (OMG) to define non-proprietary standards for MDD with automated transformations. Goals:

Portability, Interoperability, Reusability through architectural separation of concerns

Computation-independent Model (CIM)

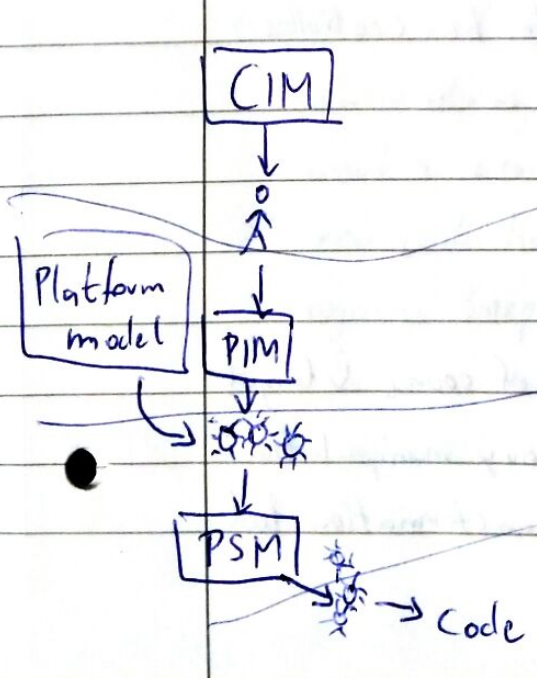
requirements: structure + behavior hidden and environment described, but

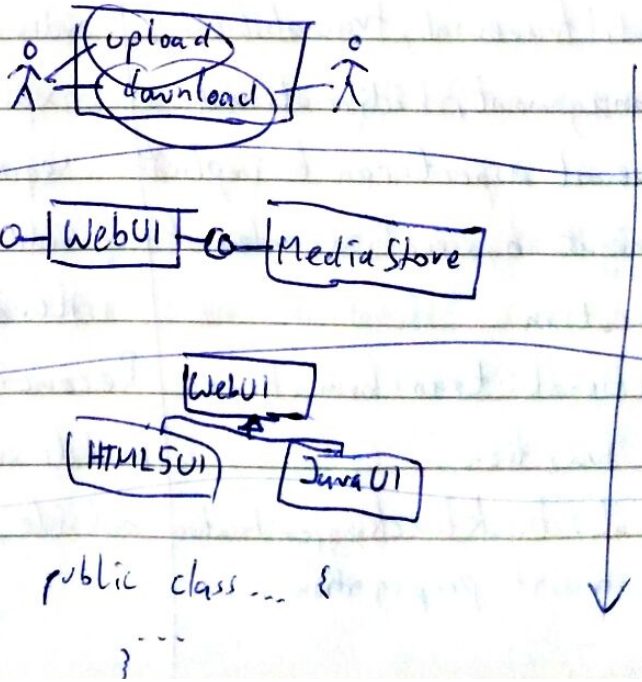
Platform independent model

focuses on the operation of a system, platform information hidden. Shows the complete part of spec, that doesn't change by platform

Platform-specific-model

extends PIM by platform info





Labour division (idealized)

Programmers: not necessary, automated as far as possible
Domain expert: creates models from which software is generated
Technology expert: preparing environment, develops modeling platforms, transformations, metamodels

Model Transformations

Automated generation of target model from source model according to a transformation definition. That's a set of transformation rules and descriptions how to transform language constructs.

Transformation ≙ applying transformation description
transformation description ≙ set of transformation rules
" rules ≙ description of how to transform one or more constructs from a source language into " " " in a target language.

This can happen declarative (what) or imperative (how)

Attractive:
model traversal, traceability management, bidirectional
several aspects can be implicit
navigation, creation, order of execution
Functional: transformation as function
Logical: backtracking, unification, constraint propagation

If declarative fails or if order needs to be controlled explicitly.
sequence
selection
iteration
Reconciliation of source & target models after heavy manipulation outside the transformation tool

Textual modelling

disadvantages of diagrams:
- graphical concrete syntaxes only cover a subset of the abstract syntax. Additional dialogues necessary
- layout information is necessary for understanding the models
- versioning problems after concurrent edits
- navigation hard for big models

advantages of textual modeling
+ reuse of textual editing tools (copy/paste, diff/merge, patches)
+ auto-completion
+ syntax highlighting
+ error markers

12 Clean Code

Unreadable code decreases productivity exponentially over time.
Changes @ code base occur. Lehman's first law:
"a system that is used, will be changed"
together w/ 2nd law:
"an evolving system increases its complexity, unless work is done to reduce it"
⇒ you will have to read your own code over and over again. Keep it clean and understandable!

SOLID Principles

S Single responsibility principle

1 Class has 1 concern

bad smell: big class ⇒ more reasons to change that particular class

benefits: easier understandable code

adding / modifying functionality affects fewer classes

risk for breaking code minimized

queries should not change the state of an object

easier to test and to understand

O Open-closed-principle

open for extension, closed for modification

write code in a way it is not necessary to change it each time you want to add functionalities

L Liskow Substitution-Principle

Functions that use pointers or references to base classes need to be able to use objects of derived classes without knowing it.

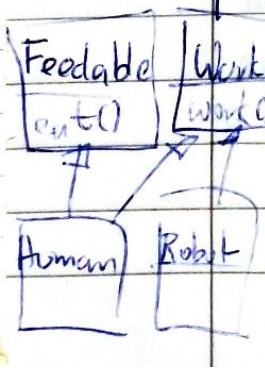
- square cannot substitute rectangle, square has stronger restrictions than rectangle.

I Interface Segregation principle

Clients should not be forced to use interfaces they don't need

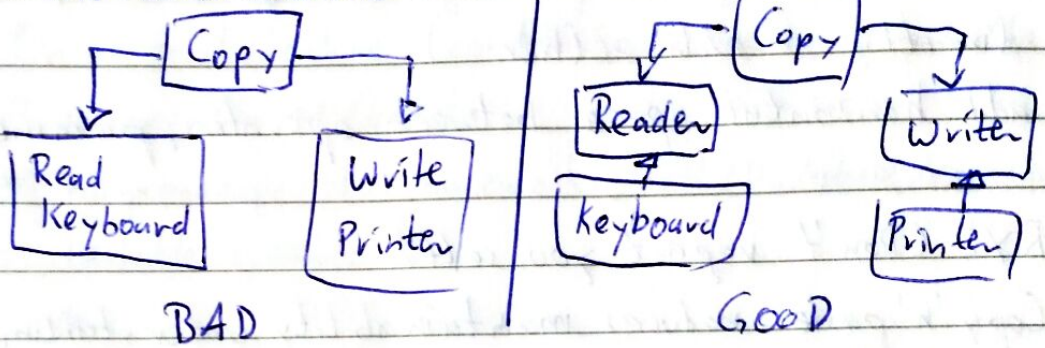
They should be kept as lean as possible (only for 1 concept)

" " " separated if used by different clients



① Dependency Inversion priciple

Put abstraction object between high-level and low level module:



BAD

Copy is affected when changing other components
cannot reuse copy easily

GOOD

depends on abstraction only. Readers & writers can be substituted

Law of Demeter "Don't talk to strangers!"
no message chains, keep effect of changes locally!

Boy scout rule: "Leave the playground cleaner as you found it!"
Code degrades over time, refactor before checking it in.

Principle of least surprise
program only what other programmers would expect
* when hearing method/class name

Code Conventions
Naming: Standardized (in project), hint on context, intention, type, sense, use, prefix. Avoid noninformation
Commenting: Don't comment bad code. Rewrite it.
whenever possible: use well named code

instead of comments

Formatting
- visually representing levels of cohesion
- add free lines between things that don't relate directly to each other
- add horizontal space between operators, parameters, ...

DRY! (Don't repeat yourself!)
- Copy 'n' paste reduces maintainability, understanding, evolvability and seeds errors over code

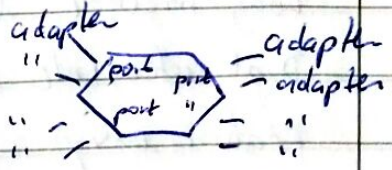
KISS (Keep it simple, stupid)
- Make it as simple as possible, not simpler
- Good code is easy to understand by everybody and addresses the problem adequately
- use reviews / pair programming

YAGNI (You ain't gonna need it)
- Featurism is costly
- don't optimize prematurely
- only implement required features

Single Level of abstraction (SLA)
- Statements within a function should be at the same abstraction level
- abstraction should decrease when reading top → bottom

Clean architecture Patterns

Separation of concerns using layers



Hexagonal (aka ports 'n' adapters)

Onion architecture (controls coupling since towards the center)

Boundary / Control / Entity

⇒ independence of frameworks / UI / Database / external agency

testable systems

"Clean Architecture"

Concentric circles represent different areas of software

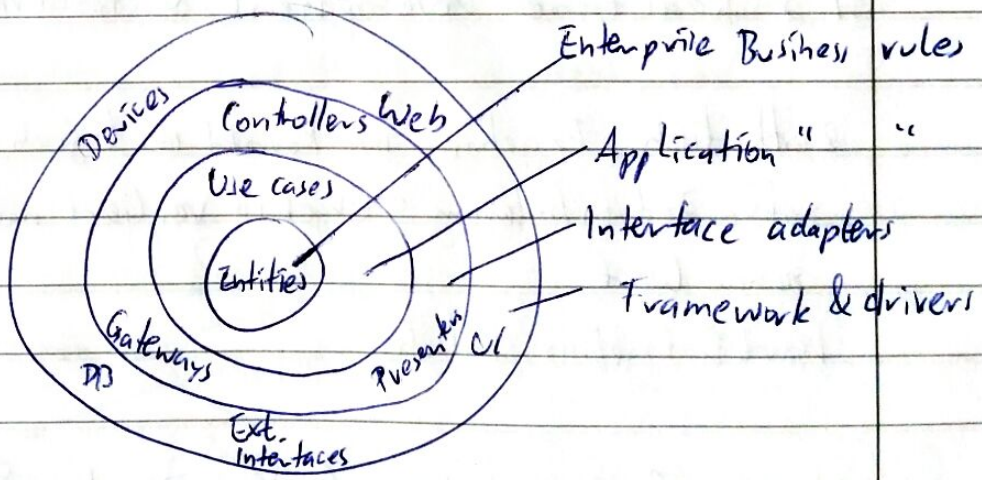
Further in ≙ higher level the software becomes

outer circles ≙ mechanisms

Dependency rule ≙ always towards point in the center

that also applies for methods, Functions, classes, data formats

⇒ no impacts from outer circles to inner circles



Refactoring

if it stinks ⇒ refactor

only with solid set of tests

Bad Smells:
- Long method (w/ comments ...) ⇒ extract
- Duplicated Code
- Feature envy ⇒ move code
- Data class (w/o methods)
- Large / God class (tries to do too much)
- Inappropriate intimacy (depends on implementation of other class)

Refactoring may reduce performance
Hard to change persistence layer of already published interfaces

When to refactor?
- When you look up details frequently
- when " feel like writing a new comment

13 Realtime Development & Patterns

zeitkritisch, Ergebnis nur korrekt ⇔ logisch ∧ zeitlich korrekt
sotte Zeitanforderung ≙ Ergebnis verliert nach Deadline seinen Wert
hart ≙ Ergebnis falsch

Stimuli ≙ Signal von Außen an das System, stoßen Prozesse an, die innerhalb ihrer Zeitschranken abgearbeitet werden sollen

Arten von RT Systemen.

Monitoring: prüfe Eingabewerte (von anderem System oder Sensor) und reagiere auf spezifische Werte

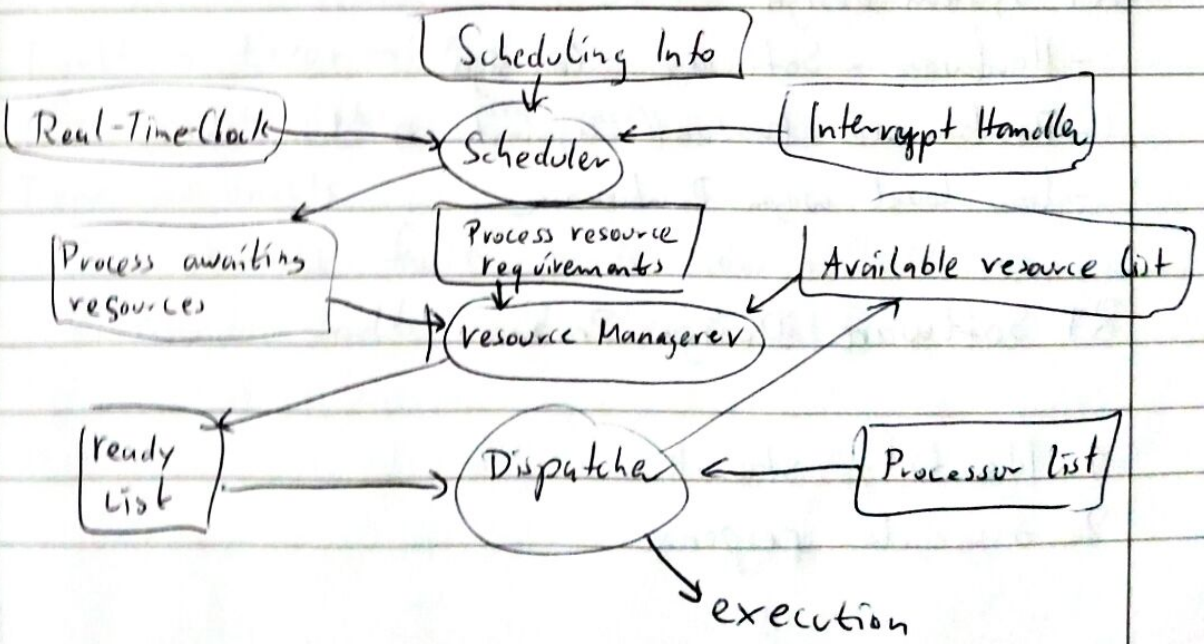
Control: like monitoring, but activate a motor / actor...

Data aquisition systems: collect data from sensors (and process it and display result)

RTOS

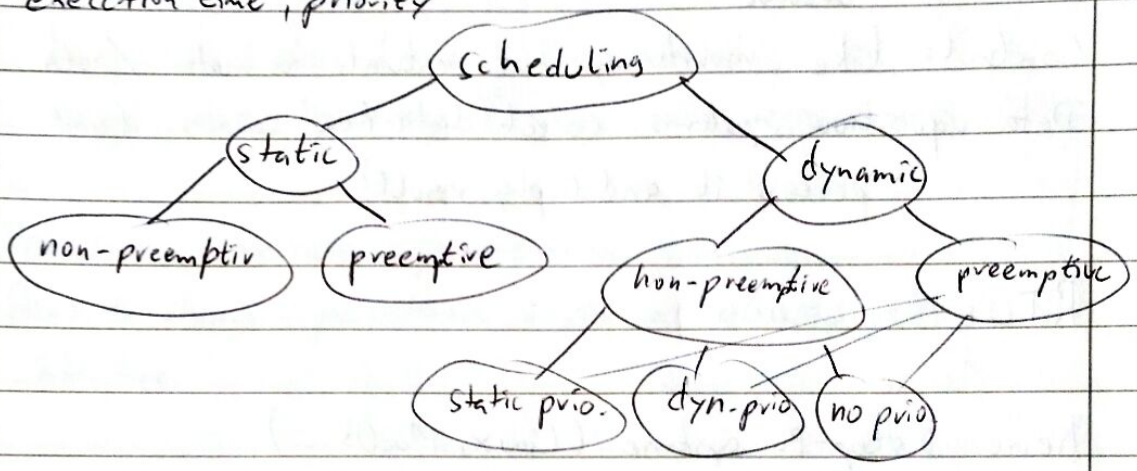
Keine Desktop-PC systeme (Linux, MacOS, ...)
Overhead wegen Netzwerk, UI, ..

Sie haben dagegen:
Echtzeituhr für das Prozess-Schedule
Interrupt-Handler, um auf Signale reagieren zu können
Scheduler, entscheidet, wann welcher Prozess ablaufen darf
Resource Manager, " ob Prozesse ablauffähig sind
Dispatcher, startet Prozesse



Process Management

Interrupt-Level-Priorität > Clock-Level-Priorität
Facts taken into account for scheduling: arrival time, deadline, execution time, priority



FiFO first in / first out: dynamic, non-preemptive, no prio.
FixedPriorities: dynamic, static priorities
Earliest Deadline First: dynamic, dynamic priorities
Least-Laxity-First: " , "
Time-Slice-Schedule: dynamic, preemptive, no priorities

RT System Design
- Hardware - Software - Codesign
- Performance tradeoff against changeability
- low level wegen Performance

RT Software Design Prozess

1. Identify stimuli
2. associate response

3. define deadlines
4. choose execution platform
5. design algorithms (FSM, UML, ...)
6. design schedule

Threads

" can exist at the same time & share data
smallest unit of concurrency
periodic arrival pattern or aperiodic
known WCET & ~~AVG~~ AVGET

communication by variables in memory (blocking / non-blocking)
or (a)synchronous messages

Dependability

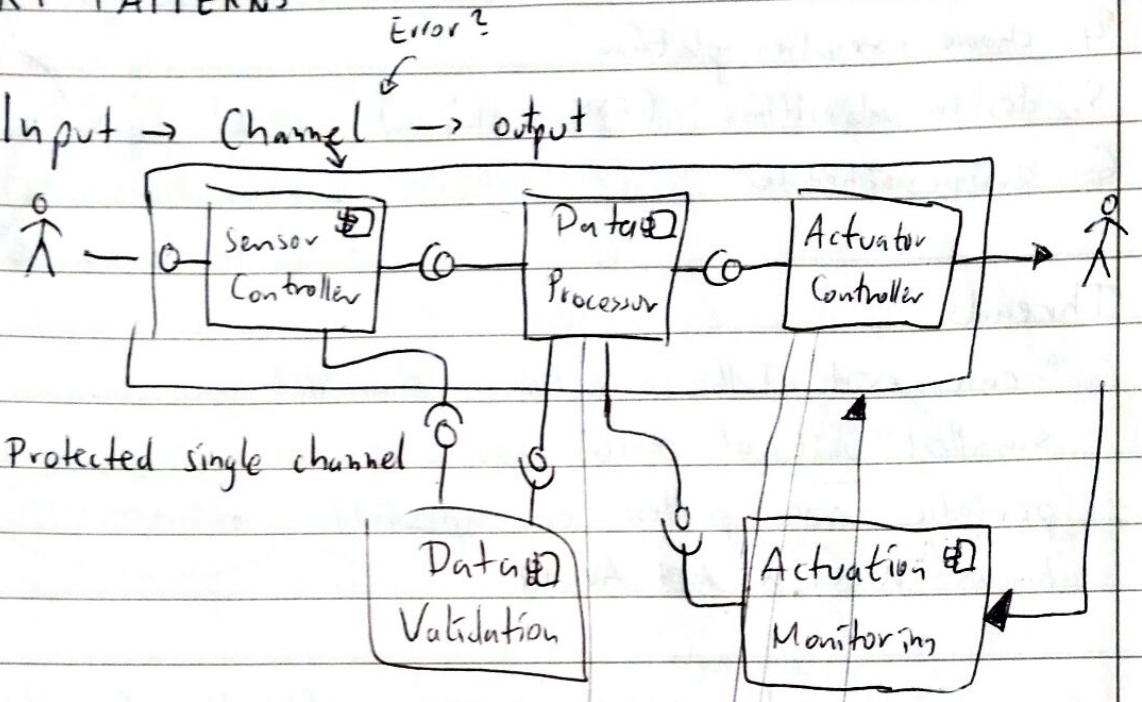
Safety (zur Umgebung)
Security (zu sich selber)
Availability ($MTBF / MTBF + MTTR$)
Reliability ($1 - \mathbb{P}(\text{fail})$)

Fault → Error → Failure
Bug wrong state car crash

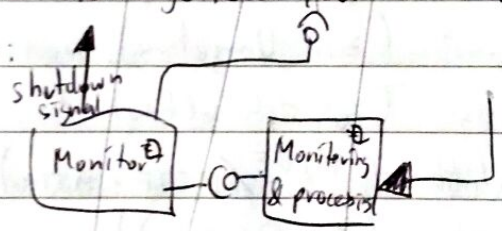
Types of faults

1. Systematic faults: made @ design / build time
2. Random faults: random / persistent

RT PATTERNS



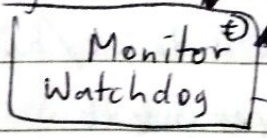
Homogenous Redundancy: 2x single protected Channel
Triple Modular " : 3x Channel + Voter
Heterogeneous " : like homogeneous, bit different implementation
Monitor actuator pattern:



protection against random and systematic faults a fail-safe-state actuator monitor sensor must be other one than sensor

Sanity check pattern: like monitor actuator pattern, but processing is very approximate only

Watchdog



Safety Executive Pattern: for non-trivial mechanisms in complex systems.
optional fail safe processing channel + Monitor + Watchdog

14 Reliability & statistical testing

= ability of a system to deliver services as specified

Stopping criteria for testing:
time / budget (finish when none of both are left)
coverage (when reaching specified coverage goal)
adequate bugs found

reliability ≙ failure-free operation of code for a specified mission time in a specified input environment
Rate of occurence of failures ≙ # failures in a given time (ROCOF)
reciproc ≙ Mean time between " (MTBF)
Availability ≙ MTBF / (MTBF + MTTR)
HW ages, Code doesn't. Error free SW is not possible forever.

Testing strategies
Functional ≙ test all user functions vs. specification
Coverage ≙ test all paths through the control structures of SW
Random ≙ random input & look 4 failures
Partition ≙ input space divided and test cases are selected @ random
Statistical ≙ test randomly with usage profile
Challenge: find most efficient testing methods

Statistical testing is needed to scientifically estimate the reliability of a software
for this we need: operational model of usage
test environment (Simbox)
protocol for analysing output & creating statistically valid inferences about reliability

Software might have infinite test cases, therefore statistically usage samples are a good way to go

probability for not failing in n runs: $(1-p)^n$

number of expected failures " " " : $n \cdot p$

combinatorical way to distribute k failures among n trials in $\binom{n}{k}$ ways

Reliability models

→ the more successful test cases have been executed, the higher is the confidence in the estimated reliability

upper bound for failure probability: $\tilde{p} = 1 - \sqrt[n]{1-\beta}$

$n \hat{=}$ #runs (independent cases/data, successful(...))

$\beta \hat{=}$ confidence that real reliability has at least been tested once: $C[\hat{R}] = P(R \geq \hat{R}) = \beta$

Usage modeling w/ Markov chains

Automated test generation (random walk, coverage test)

Usage profile is computed from model

Core conditions for usage based statistical testing

1. all cases are derived from the same operational profile (which is statistically)
2. all runs are independent and the system state is reset after each run
3. oracle predicts outcome of test run

What can be done w/ them?

automated test generation

statistics computation
requirements analysis
statistics can validate assumptions about usage

15 Security

is a quality requirement

Why Security is important:

insecure software may lead to

- downtimes
- data leaks
- bad reputation
- fixes
- safety issues

} costs money also because firewalls, spam filters, anti-virus software...

Goals of secure software:

- confidentiality (privacy)
- Integrity (of data & system)
- Availability

supporting goals:

- user authentication
- traceability & auditing
- Monitoring
- Anonymity

Security for whom and against what makes decisions about " relative

You can't show the absence of security holes, only their presense (like bugs)

Why is software (becoming) insecure?
often due to "unusual" management constraints
- no time / money left
- requirements change
- system used in different context
- people unexperienced
- bad design
- insufficient input validation
- race conditions
- pseudo randomness

Unlike safety or reliability problems, security problems are caused by smart people (hackers)

Attack approaches
- external: network sniffers / proxies / viruses / worms / trojan horses / rootkits / port scanner
- internal: burglary / insider attacks / theft / social engineering attacks

Security requirements should be considered as early as possible !
later changes are expensive or ineffective

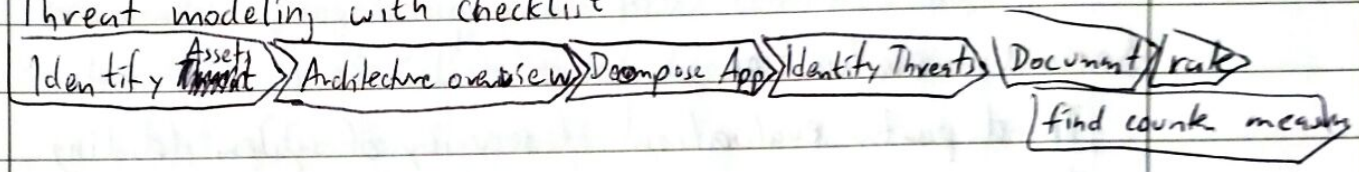
Software security management = risk management

Design for security
consider " in each phase of development
trade it of with performance, usability, functionality, costs and time to market

also secure your development environment

How to find Security Requirements
- consider security goals (however, too general)
- think of misuse cases
- consider system context (credit card, ...)

Threat modeling with checklist



Building secure software
1. secure weakest link (system only as secure as weakest part)
2. practice defence in depth (series of defences)
3. Fail securely (handle exceptions securely, don't expose details)
4. Secure by default (switch on all security features provided)
5. Principle of least privilege (only grant minimum access for shortest time)
6. Obscurity ≠> Security (no hidden or magic URLs)
7. Minimize attack surface (less code, few interfaces ... they all bring bugs)
8. Privileged core (isolate code with security privileges)
9. Input validation / output encoding
10. Don't mix data & code

Further helpful security patterns
1. Cryptography may solve many security problems
2. Use a federated identity management (existing services)
3. Use passwords carefully (measure strength, process them on server)
4. session handling is critical in web apps (cookies, IDs, ...)
5. bugs / careless implementation => security issues, raise awareness of all stakeholders
6. certain language features bring general security issues

Some testing guidelines
1. focus on identified risks
2. use good code coverage metric
3. perform code & system reviews
4. aim to evaluate security in each phase
5. use various sources for test cases
6. set up standard security test environment for reuse
7. have security experts check your system (black-box or white-box)

Security evaluation
Third party evaluation of security of system leading to a security certification
often define development approach that needs to be followed => difficult to add afterwards

Some Example of security issues
- Race conditions
- Buffer overflows
- SQL injections => sanitize inputs

16 Continuous Integration

What CI brings w/ it:
Avoids magical "it works on my machine!"
clean, well defined build environment w/o assumptions and magic strings
ensure the software is actually deployable all the time
"s quality via functional testing & quality tests
which are performed in representative environment
highlighting defects that occur

CI ≙ software development w/ multiple integrations per day
⇒ needs to be integrated into software development proces
automated build test to detect errors as quickly as possible
Reussner "CI is the continuation of agile programming to integration."

CI development process
manually: programming / local tests / commit changes
automatically: materialisation of dependencies / compilation / test (Functional, quality)
provide meaningful developer feedback
↪ then manually again: Fix errors

Benefits:
reduced risk (detected by automated tests) quickly
eliminates uncertainty (what (doesn't) work?)
reduces repetitive processes like testing, compilation, building & deployment
enables rapid deployment of Software (= continuous delivery)
" productive communication in developer team
new quality metrics (did a recent change affect the system positive (or not)?)
is it realistic to meet milestone with acceptable quality?

Best practices
Version all relevant development artefacts
- source code / tests / build automation artefacts / build job descriptions / deployment description for testing

avoid duplicate (configuration) code
design build for fast feedback: fail early
build for any environment: same build job should be able to produce deployment artefacts for private / integration / release build to prevent inconsistencies
if possible: one build per deployment unit
use continuous feedback mechanism (what / who caused build to break)

17 Reviews (almost a silver bullet!)

meetings, where a software artefact is examined to improve its quality

validation ≙ case check on expected behavior
verification ≙ check wether refinement relation holds between two documents:
requirements specification vs. code
" " " architecture & design
architecture & design vs. code

Forms of reviews
- Inspection
- Team review
- Walkthrough
- Pair programming
- pass around / ad-hoc review

Can concern various artefacts: requirements, project plans, checklists, architecture, design, code, test cases

Silver bullet, but rarely used - why that?
consultants cannot make money out of it
not new
increase up-front costs
requires consideration of psychological factors

Dangers of reviews:
- Testing is omitted
- Authors are frustraded or even feel violated → tend to get sloppy

Dangers for reviews
- Managers cut time if deadline approaches
- People are not well prepared (waste of time)
- author may try to protect himself with obfuscation / documents that are difficult to understand

Benefits from reviews
Quality, correctness
Reviews can verify artefacts in natural language
Improved project understanding
knowledge shared and distributed
less dependency on one person who has the knowledge
Teaching of styles and experiences to novices / newbies
Better readability of artefacts
less tricky
better documentation / comments

Phases of a review

1. Planning
2. Overview
3. Preparation
4. Meeting
5. Rework
6. Follow-Up
7. Casaul Analysis

Roles

1. Author
2. Moderator
3. Reader(s)
4. Recorder
5. Verifier

Code reading techniques
reading based on test cases
Control flow structures ⇒ assumptions what could be and check

Psychological interaction patterns
Alcoholic
„now I've got you"
„see what you made to me"
hurried
„if it were not you"
„look how hard I tried"
Schlemiel
"ayes, but"
"wouldn't it be nice, if"

18 Cloud computing

Essential characteristics:
1. Scalability
2. Self-Service
3. Ubiquitous Access
4. Resource pooling
5. Measured service

The illusion of infinite IT-resources which
- are always available
- network-accessible
- as on-demand services that
- do not require ownership, prior reservation, etc.~

Pay for use business model

What are resources?
Compute power
Storage
Network
$
Energy
Real-Estate
Human (power)
Software Apps
Data
Licenses
Updates

Problem: Efficient use of resources
concurrent use by various customers (tenants) and scheduling systems
Varying requirements with respect to operating system, software, hardware, etc...
Unpredictable load, especially often in webservices

Multi Tenancy (Mandantenfähigkeit)
Isolation of workloads
Separation of customers
Customers gain administrative privileges

Partitioning of resources
Classical solution: Separation of servers
⇒ physical resource sets (PRS)
Cloud solution: separation of services
⇒ virtual resource sets (VRS)

Virtualization ≙ the process of presenting physically distributed or in any way not united as a whole, easy to use, things to the customer. Rather logical view than physical.

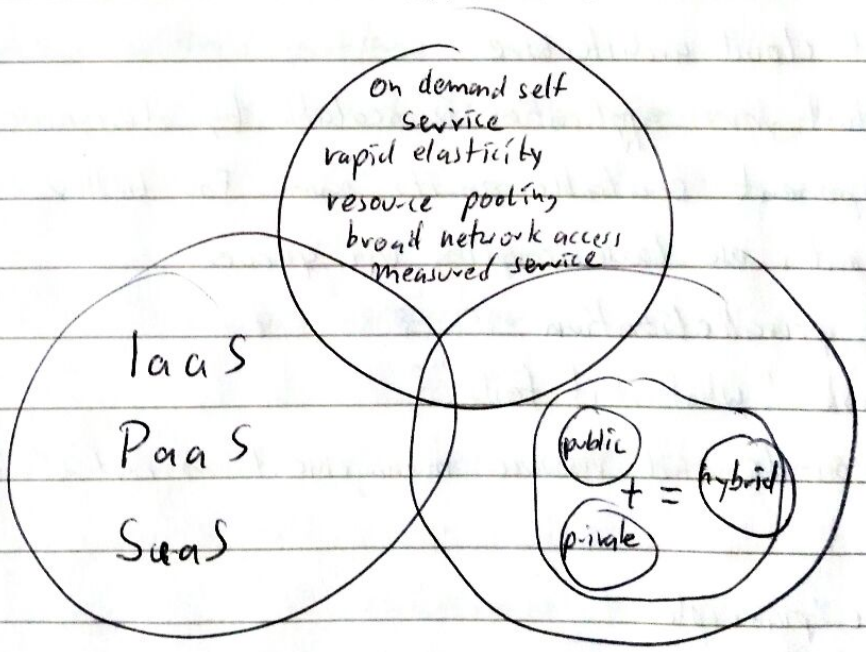
Virtual machines using hypervisor.
Hypervisor ≙ control program
Partitioning ≙ multiple OS on one server
Isolation ≙ no side effects between VMs

Parnvirtualization

no emulated HW Layer, guest OS calls hypervisor for HW demand

Advantages of virtualization
+ improved energy efficiency (100+ VM one one server)
+ availability (migration between VMs without interrupt)
+ Quality of service relies on service level agreement
+ decoupling of demand & resource provisioning
+ overbooking of resources
+ logical view on it improves management
+ only small performance decrease throught virtualized software

Cloud computing conceptual view



SaaS Web apps, access through web (browser)
PaaS Development or execution environments
IaaS Storage / Network / computing resources

multi tenant architecture:
many users share the same hardware, but isolation is provided. every user can configure via metadata, how the application behaves and appears to the users

Service deployment model
private cloud: customer and cloud belong the same organization
public cloud " " " " to different "
hybrid ": combination of the two above
community cloud: Stakeholders share resources to achieve common goal

Principle of cloud architecture
ensure that your application is scalable by designing each component scalable on its own. For better management, use loose coupling via queues
implement parallelization
always ask "what if it fails"?
implement on-demand resource management ⇒ cost efficiency

Architecture principle
Decentralization to remove scaling bottlenecks and single point of failures
Asynchronity, system makes progress under all circumstances

Autonomy, all components can make decisions on their own
Local responsibility, all* operations are designed, such that no or limited concurrency control is required
→ controlled concurrency: each individual component is responsible for achieving its consistency
Failure tolerant: System considers the failure of components to be normal mode of operation and continues operation with minimal interruption
Controlled parallelism: Abstractions used in the system are of such granularity that parallelism can be used to improve performance and robustness of recovery or the introduction of new codes
Building blocks: don't provide a single service that does everything for everyone. Instead build small components that can be used as building blocks for other services
Symmetry: Nodes in the system are equal, code can move between them without problems
Simplicity: The system should be as simple as possible (but no simpler)

19 Software Evolution

Software does change, because the environment in which it is deployed changes.
requirements " => increased complexity
Fix bugs / switch platform / implement new req.
corrective adaptive perfective changes
~17% ~18% ~65%

changing business process
new platforms / libraries
changing constraints (legal...)
} change

time + cost pressure ⇒ shortcuts needed
shortcuts often decrease product quality
reasons: unforseen usage scenarios
mix of old & new techniques
lack of knowledge of new system
non-adherence to process-model
typical syndroms:
limited understanding
outdated documentation / missing
longer development cycles
more bugs & higher fix time
design & code fragility
code smells
longer build times
missing / incomplete tests

Evolution = development + maintenance ⇒
process of changing a product that has been deployed

Importance of Software Evolution
supports changing business needs
keep in pace w/ technological development
ensure up to date knowledge of system
prevent quality degradation

Maintenance cost breakdown:
Understanding ~47%, Coding ~19%, Documentation ~6%,
testing ~28%
consistency between software artefacts
use advanced technology features
maintenance team stability

Maintenance prediction
what part is most likely to change?
how many change requests are expected?
what is the lifetime?
how much will maintenance cost?
which part is the most expensive for changing?

Legacy system
Valuable piece of software
maintained over long period of time
Large & old
outdated platform / technology
" tools & languages
fragile after lots of patches
outdated or missing documentation

SE process is iterative with the following activities:
Change management
" impact analysis
reengineering
software modernization

software quality assurance
quality impact analysis

ensure traceability between software artefacts
forward: requirements → software
backward: code → programmer

Manage change requests: validate & collect, analyze impact, document them, cost estimation, plan implementation

Impact analysis
determine extent of a change by known impacts ~~&~~ ripple effects
supporting techniques, graphs, comparison, static program analysis, test coverage analysis

Software modernization
=> new language, environment, frameworks, libraries, OS, HW
source-to-source translation

Software quality assurance ≙ process of detecting and correcting design problems (to reduce maintenance costs)

Design for evolution
Modularity: Decomposability of the problem into sub problems
Composability of module to build new systems
Understandability of a module in isolation
Continuity ≙ small changes have localized effects
Protection ≙ fault isolation

Meyer's rules of modularity

Direct mapping: structure of solution reflects structure of problem

Few interfaces: module should communicate with as few others as possible

Small " : modules should exchange as little information " "

Explicit Interfaces: modules should only use obvious means of communication

Information hiding: " " " expose a subset of their properties

Encapsulation ≙ software development technique that consists of isolation for a system function or a set of data / operations on those data within a module and providing precise specification for the module

+ improve change localization

+ support software understanding

examples: abstract data types / classes

modules / units

Abstraction ≙ view to something that does only include relevant information for a given aspect. Everything else is omitted

+ filter out details

+ reduce complexity

Examples: Abstract data types (separate interfaces from implementation)

OO, MDA (separate PIM from PSM

Information hiding ≙ software development technique in which modules interfaces only reveal as little information about its implementation / data as possible

+ protect internals of a module
+ restrict access to volatile implementation details
Examples: private members / static global functions

Cross-cutting concerns ≙ any interest in the system that
addresses multiple classes
violates modularity, often not OO or compatible with
the invented system design

Good design (for software evolution) ensures flexibility and
adaptability to changes
eases software understanding
limits the impact of a change
enables large scale reuse
improves communication with stakeholders

Classes should only depend on an interface since they are
rarely subject to changes
+ better stability in the face of changes
+ decreased coupling

Isolate volatile behavior

Design problems
improper Layering (bypass calls, low → high calls, inter-
dependencies)
Consequences: hard to understand architecture
fragility due to unexpected dependencies

Simulated polymorphism
- large switch/case over class instances
- hard to add new classes
- maintenance is error prone

Refused Bequest
- child class refuses inherited functionality
- interfaces empty implemented / not used

Feature envy
- a method is more interested in data from another class

Knows of derived
- a superclass depends on its subclasses
- cyclic dependency

God class
- a large, overly complex class