Kapitel 1

20.01.25

- What is ~~a~~ Software?

It includes programs and ~~a~~all of the relevant information needed for its:

1. Installation
2. Operation
3. Repair
4. Enhancement

It ~~s~~includes configuration data, documentation (system documentation like Architekturmodell and user documentation like configuration / how to use the software), support, ...

Types of software:

1. Anwendungs~~s~~oftware: application software interacts directly with end users. It can be a general ~~sof~~ like a text editor or used for more specific purposes like CAD.

2. System-Software: ~~ap~~ do not directly interact with end users. They keep the system running by operating in the background. For example, OS (Windows)

3. Software as a Service (SaaS): softwares that do not need to be installed to be used. They are accessed through client softwares (browsers) ~~servers~~. For example: zoom, netflix. They run on servers.

- Systemische Entwicklungsansatz: systematic development approach which is a structured way to develop systems/softwares. Needed for lesser chance of failure of a software project.

- Eigenschaften of a software:
  1. No physical boundaries ⇒ challenging to separate subsystems
  2. No control: ~~over~~ in SaaS users have not control over when updates are applied and what features are added/removed.
  3. Long life expectancy
  4. Virtually no distribution costs so bug fixes are also easily distributed.
  5. Software can be accessed through networks ⇒ large attack surface.

- Reasons a software project could fail?
  1. Team members leave
  2. Insufficient experience in development
  3. Insufficient ~~ab~~ Knowledge about the tools or programming languages used.
  4. Insufficient knowledge about the business domain
  5. Requirements change or arise during project lifetime

- What is Software Engineering?

Sdq dom

The systematic, disciplined, quantifiable approach to the development, operation and maintenance of software, that is, the application of engineering a software.

→ Systematic: defined by set of rules.
→ disciplined: ~~always~~ doesn't break ~~within~~ the rules
→ quantifiable: ~~m~~ decisions based on metrics of the software.

- Bereiche of Software Engineering:

1. Entwicklungsprozesse: how each team should work on the software.
2. Anforderungsanalyse: to establish what the software should be able to do.
3. Domänenmodellierung: ensures the software is as close as possible to the problem domain.
4. Entwicklungswerkzeuge: helpful tools.
   → IDE: (Integrated Development Environment) used for debugging, refactoring, navigating code, etc.
   → Shells: manage processes
   → Buildsysteme: for repetitive tasks
   → Versionskontrollsysteme: managing different versions of a code.
5. Testen (or statische Analyse - both are forms of validation)
6. Designsprinzipien: desired is high cohesion (ability to perform a specific task) and low coupling (dependency of individual components on one another)
7. Entwurfsmuster: solutions to common programming problems as to not waste time/effort trying to solve problems that have been already solved.
8. Architektur: describes the abstract design, how the software should be structured in a component, and how components should interact and communicate with each other
9. Betrieb: how the software should operate and distribution of new versions.

Übungsblatt 1

Aufgabe 1,

- Unterschiedliche Schwerpunkte / different focuses of the types of softwares:

1. Anwendungs-Software:
   → greater focus on user interface
   → avoidance of user errors
   → testing is done from the user's perspective
   → distribution (installation / updates)

2. System-Software:
   → interface should be usable by other developers
   → interaction with hardware
   → compatibility with higher-level application softwares.

3. Software as a Service (SaaS):
   → Scalibility
   → accessible through network so security requirements are different.

Aufgabe 2,

- Software Engineering should be systematic, defined and quantifiable. Provide an example for each.

1. Systematic: code must be reviewed before deployment
2. Disciplined: style checker / reviews must be signed
3. Quantifiable: count the number of violations of the programming guidlines in a code.

Kapitel 2
Entwicklungswerkzeuge

20.01.25

- Warum sind Entwicklungswerkzeuge wichtig?

They increase productivity, However, they need to be learnt (time spent learning how to use them). It is important to pick suitable tools at the start of the project and only change them if there will be a significant advantage.

1. IDE (Integrated Development Environment):

Some features of IDEs:

1. Data explorer: lists directories and project files
2. Syntax highlighting: dif. colors for better readability
3. Code completetion: suggests code snippets
4. Problem indicators: finds and highlights errors
5. Code navigation
6. Testing
7. Debugging: steps through code execution with stack/memory inspection
8. Refactoring: Umbennenung/rewriting code
9. Version control
10. Terminal/Shell: runs commands

2. Shell:

an interactive programming language that manages processes and carries out operations. It can summarize Befehlen/commands in scripts, and is well-documented (manual pages)

Example:

```
LS --recursive --ignore=*.png dir1 dir2 dir3
```

Flags: can be in short (-R) or long form (--recursive) and can have arguments (--ignore=*.png)

- Useful Shell commands:

- touch: create empty file
- ls: list
- mv: move
- cp: copy
- rm: remove
- find: find data
- chown: change owner
- chmod: change mod

  To change Access rights
- cat: concatenate
- wc: word count
- sort
- grep: search texts
- echo: "print" text (ausgeben)
- head: start of long text (print/ausgeben)
- tail: end of long text (print/ausgeben)
- less: print part of a long text
- tee: print text auf Shell AND in a file
- pwd: print working directory
- mkdir: make directory
- cd: change directory
- rmdir: remove directory (delete)

- stdin: reads text from Standardeingabe
- stdout: prints text onto Standardausgabe
- stderr: prints Fehlermeldungen on the Standardfehlerausgabe.

  error

Example: pipeline

cat file1 | sort | tee file2

- cat reads file1 and prints to stdout (default behavior of cat)
- sort sorts the lines from stdin and writes the sorted lines to std out.
  → because file1 was passed via the pipeline (|), sort reads directly from std.
- tee reads from stdin, writes the text to a file named file2 and simultaneously outputs it to stdout.

- Input- und Output-Redirection

- Befehl 1 | Befehl 2
  stdout redirected to stdin for Befehl 2
- Befehl 1 > Datei
  stdout is redirected
- Befehl 2 > Datei
  stderr from Befehl is redirected
- BEFEHL &> DATEI
  stdout and stderr from BEFEHL ~~as stdin~~ redirected to DATEI
- BEFEHL < DATEI
  content of DATEI as stdin
- BEFEHL 1 && BEFEHL 2
  carry out 1, and if successful, then carry out 2
- BEFEHL 1 || BEFEHL 2
  carry out 1, and if it fails, then carry out 2
- Exit Codes 0 - 255:
  0 means success, 1-255 shows error (exact no. specifies error)

Shell code:

- Assigning values to variables:

```
FOO =123
echo " The value of FOO is $FOO"
```

Ausgabe: The value of FOO is 123

- Bedingungen:

```
if [ $FOO -lt 123 ] ; then echo "hello" , else
echo "bye" ; fi case $FOO in (123) echo "hello" ;;
(456|789) echo "bye" ;; esac
```

- These are 2 dif. blocks – an if block and a case block. To end both block we write the word reversed so if = fi and case = esac

- -lt 123: checks if FOO is less than 123

- (456|789) : checks if $FOO in (456) OR (789)

3. Build systems:

automate repetitive tasks. They have a built-in support for common tasks.

- sbt compile: compile code into executable file
- sbt test: run tests
- sbt clean: remove compiled files
- sbt console: start an interactive interpretor
- sbt doc: generate code documentation as HTML or PDF
- sbt publish: publish executable files to a server
- sbt scalafmt: format code
- sbt scalastyle: check code style

- Build systems are configured with build scripts (configuration files). They usually only process changed files and their dependencies (• incrementality)

4. VersionsKontrollsysteme (VCS)

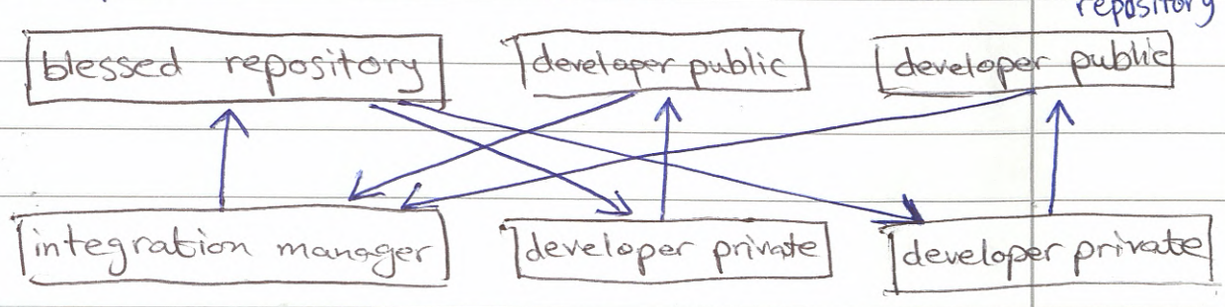
manage ~~diff~~ multiple versions of code enabling:
→ collaberative work
→ history tracking of code changes
→ integration of changes by different developers

Types:
1. Centralized VCS: has a central repository
2. Decentralized VCS: has multiple independent repositories (like Git). Widely used today.

- Decentralized VCS features:

1. Commit: records changes with description, authorship, diffs (difference between 2 versions of a file)
2. Branches: independent work lines
3. Merge: integrates changes from different branches
4. Repositories: organize branches and commits, sync local and remote repositories.
5. Workflow: describes how developers synchronize changes across repositories. Never modify the commit history of 'blessed repository'

Workflow



- Useful Git Commands:

- git init: create new local repository
- git clone (<Repository-URL>): clone a remote repository to local
- git status: show changed files
- git diff: show line-by-line changes
- git log: show change history
- git add (--all | <File-Path>): add files to next commit
- git commit (-m "<Description>"): create new commit
- git checkout (-b <Branch-Name>): create new branch
- git checkout (<Branch-Name>): switch to existing branch
- git merge (<Branch-Name>): merge
- git fetch: fetch commits from remote repository to local one
- git push: push commits from local to remote repository
- git pull: combines git fetch and git merge

Übungsblatt 2

Aufgabe 1,

- 2 advantages of using build systems especially in team development.

1. <u>Uniform build configuration</u>: If the same build configuration is used for all developers, then a developer can not forget any steps. The build ~~system~~ is always executed in the same way.
2. <u>Automation</u> (One-click build): developers often do not need to see the exact steps being executed (e.g through a one-click build). Quality assurance steps can be defined in the build configuration.

Aufgabe 2,

In google doc

Kapitel 3
Entwicklungsprozess

21.02.25

- Software-Engineering-Process:
  systematic approach to design ~~&~~ or implement a software that maximizes ~~th~~ benefits for stakeholders at a minimal cost while reducing risk of project failure and time / budget overruns.

- Types of Stakeholders:
  1. Internal Stakeholders: ~~a~~ developers, project managers, companies whose employees are working on the software.
  2. External Stakeholders: clients / customers, government

- Advantages of Software-Engineering-Process:
  1. Risk management
  2. Continuous improvement
  3. Scalibility and efficiency
  4. Consistency
  5. Quality assurance
  6. Collaberation and coordination

- Bereiche of Software Engineering Process:

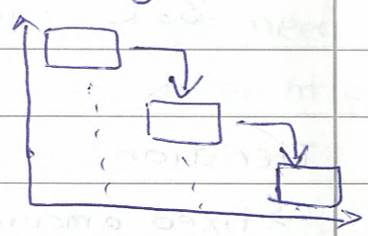
ADEVBW
1. Anforderungsanalyse
2. Design
3. Entwicklung
4. Validierung
5. Betrieb
6. Wartung / Maintenance

- What is a Process Model?

it's like a recipe for structuring/guiding a software which specifies the steps needed and the order they should be carried out in, in order to achieve a goal. We have Waterfall model, V-model, Agile models.

1. Waterfall model:

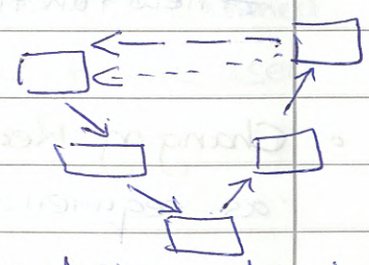
→ divides activities into distinct phases.
→ specifies a sequence ~~the~~
→ a new phase can only begin after the previous one is finished.
→ each phase must produce a set of approved outputs
→ an error may require revisiting previous phases
→ it is similar to traditional engineering.



2. V-Model:

similar to Waterfall model but with a greater focus on:

→ Design
→ Testing / Validierung



~~left~~

Diagram is a V-shape. with left side being development steps and right side testing.

<--- Verification (internal, checking code)

<..... Validation (external, testing from user perspective)

- Why Waterfall and V-model fail?
  1. Validation happens too late ⇒ expensive fixes
  2. Frequent requirement changes during software development.
  3. Software deployment and feedback are delayed, risking failure

  ↳ Agile process models are better at handling changing requirements.

3. Agile Process Models:
   Like lean, scrum, XP

   They are much better at handling frequent changing requirements because they have:
   1. Shorter development cycles
   2. Continuous Feedback
   3. Continuous Improvement

- Iteration:
  → fixed amount of time after which a program or a feature is presented for feedback
  → unfinished functions are passed onto the next iteration
  → new function can not be added mid-iteration

- Changing Requirements:
  as requirements change or new requirements arise, we use User Story or Use case to express them.

  - User Story: a short, simple description of a feature from the user's perspective.

- Use case: detailed description of how users interact with the system.

- Story Points:
a metric used in Agile Process Models to estimate/measure how complex and time-consuming a ~~task~~ User Story is compared to other/previous User Stories.

- Poker Planning: a method used by the team to determine the Story Points of a User Story, where all members secretely pick a number, then they reveal it. If there are disagreements (e.g one person picks 2 and the other picks 8), they discuss and reevaluate until they are in agreement.

- Iteration Planning:
developers and stakeholders plan what steps need to be taken in the next iteration. They have different goals:

Stakeholders:
1. Scope: what features do users need?
2. Priority: which feature is most needed first?
3. Interaction/Interdependence: how the planned features work together to bring value to the users.
4. Release date: possible deadlines.

Developers:
1. Consequences: technical consequences
2. Detailed planning: what resources are needed, etc.

- Velocity:
the sum of Story Points of all User Stories finished in an Iteration is the velocity in that iteration.

For example:
In an iteration (2 weeks), sorting (3 Story Points) and ~~de~~ new feature (1 Story Point) were finished.

Velocity = 3 + 1 = 4

- Capacity:
estimates the total number of Story Points a team can achieve in an iteration based on:
  → velocity in previous iterations
  → if team members left

The team is not allowed to plan more Story Points in an iteration than their capacity allows!

- Iteration Coordination:

1. Kanban-Board:
→ Columns represent different phases.
→ different colored notes ~~or fonts~~ are used for dif. types of tasks
→ It shows who is working on what and how much progress was done.

2. Daily-Standup:

→ developers have a meeting about updates and possible problems/delays while standing to keep the meeting short and concise.

- Iteration Completion:

1. Sprint Review:

→ developers and stakeholders have a meeting after an iteration with feedback from stakeholders.

→ Open discussion on how the product can be improved in the next iteration.

2. Retrospective:

→ team (developers) discuss how the process was in the last iteration (good/bad parts) and how to improve it. (Focus on Process)

- Agile Principles: (Manifest)

Agile Process Models focus on:

1. ~~Focus~~ flexible plan rather than a strict one
2. Working software rather than extensive documentation
3. Interation/Collaberation over tools
4. Working closely with clients rather than rigid negotiation contract.

- Criticism of Agile:

→ Challenges in Stakeholder Collaberation

→ Unclear Project Scope (Time/Budget)

→ may take longer due to flexibility

Übungsblatt 3

Aufgabe 1,

- Advantages and disadvantages of Waterfall model:

Advantages:

1. Low management effort
2. Improvement from unstructured development (code and fix)
3. It is easy to understand

Disadvantages:

1. Risk is not being considered
2. Fewe opportunities for feedback
3. Late-stage testing
4. No knowledge-driven development (follows a rigid plan)

Aufgabe 2,

- How does V-Model improve Waterfall model?

→ Better Quality Assurance because of sufficient testing after each phase.

Aufgabe 3,

- Agile Manifest:

1. Working closely with clients over rigid negotiation ~~plan~~ contract
2. Working software over extensive documentation
3. Interaction and collaberation over tools and processes
4. ~~Flexi~~ ~~Lasi~~ Adjusting to changing requirements over strict plan.

Aufgabe 4,

- In What ways are Agile Process Models different from traditional Process Models.

  1. Shorter development cycle
  2. Continuous feedback
  3. Continuous Improvement

Aufgabe 5,

- Advantages of a short Release-cycle in software development.
  1. Faster reaction to changes
  2. Earlier Feedback
  3. Risk Management
  4. Incremental addition of functions

Aufgabe 6,

- Explain Sprint-Review and Retrospectives.

  - Sprint-Review: developers presents the 'new' User Stories (from the last iteration), Stakeholders give Feedback. Discussions on improving the product (software) take place.

  - Retrospective: developers discuss how efficient the process was and how to improve it (velocity, project management, etc)

Aufgabe 7,

(a) Velocity

Sum of story points of all finished User Stories in the last iteration

(b) Capacity:

→ should not exceed velocity in part (a)

→ unfinished functions from last iteration have highest priority.

→ Add from highest to lowest without exceeding velocity.

Aufgabe 6,

Velocity = 34

- ~~Next iteration: 3, 8, 14, 10, 11, 15~~

- Unfinished Story points do NOT have highest priority, just stick to priority numbers.

- Next Iteration : 14, 8, 10, 3, 11, 9, 16 (whatever fits now)

Kapitel 4
Anforderungsanalyse

- Anforderungsanalyse:
  the process to Identify, Document, Validate, Manage what the software Should be able to do and what its qualities are.

  → Good and clear communication between developers and stakeholders is important. They determine the ~~→ Requir~~ requirements together.

- Types of Requirements:

  1. Stakeholder Requirements:
     → User Requirements
     → Business Requirements
     → Regulatory Requirements (e.g data protection laws)
     ⋮ etc
     there are more...
  2. System Requirements: the technical needs required to ~~meet~~ fulfill the Stakeholders' ~~needs~~ requirements.

- Functional vs Non-functional Requirements:

  Functional Requirements:
  → what the software should be able to do.
  → focuses on functionality
  → e.g: users should be able to log in with username and password.

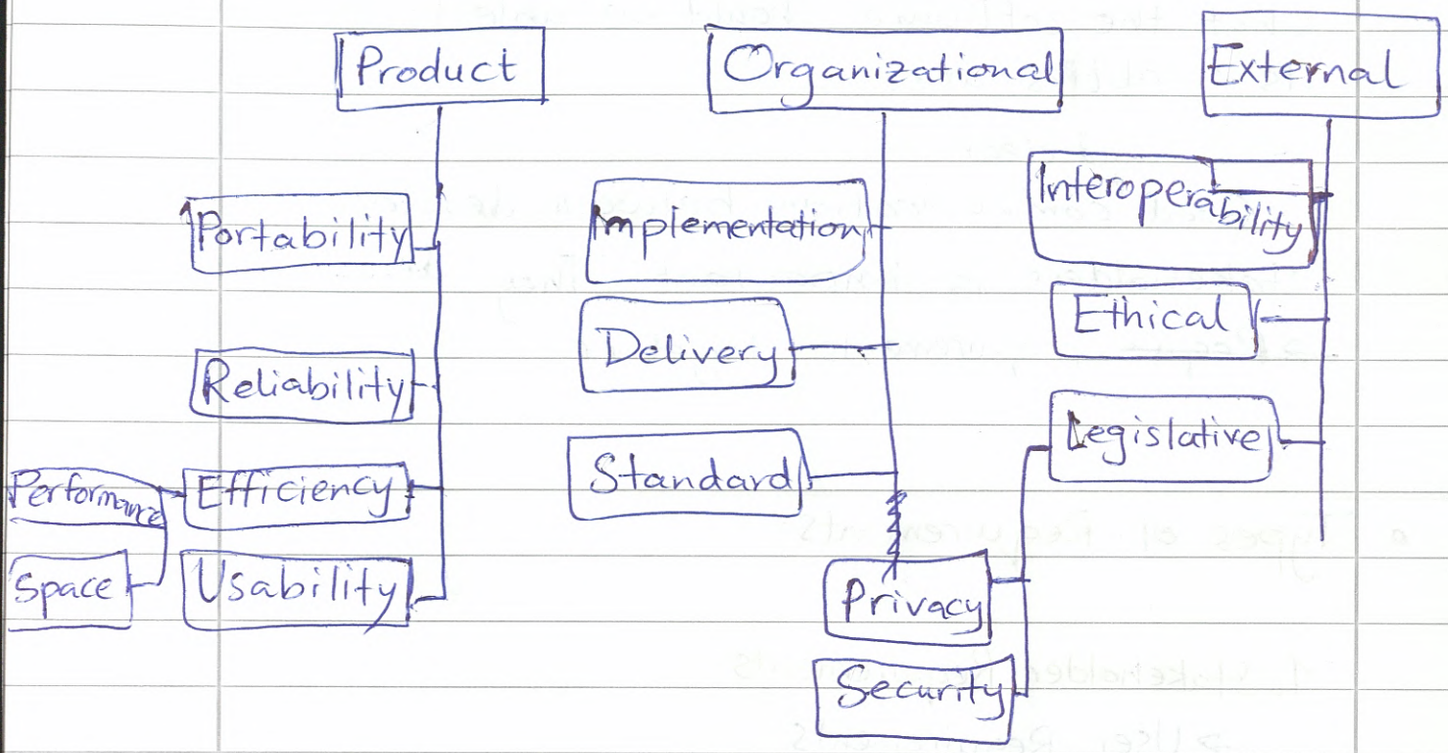
Non-functional requirements:

→ how the software should be able to peform

→ focuses on operationality (design, security, ...)

→ e.g: pages should load in 2 seconds.

Non-functional Requirements



- Requirements Analysis Process:

1. Elicitation: identifying the requirements

2. Documentation: documenting the requirements.

3. Illustration: visualizing the requirements.

4. Validation: ensuring requirements have been met.

1. Elicitation Requirements:

some of them are:

1. Identifying the Stakeholders
2. Considering the different perspectives of the Stakeholders
3. Interviews
   - ↳ closed: pre-defined questions
   - ↳ open: free discussion
4. Surveys
5. Feasibility test (how doable it is)

⋮

- Feasibility test:
  1. Technical Feasibility
  2. Economic Feasibility
  3. Operational Feasibility
  4. Legal Feasibility
  5. Time Feasibility

2. Documentation Requirements:

1. <u>Functional (documentation) requirements</u>:
   - → User Story
   - → Use Case
2. <u>Non-functional (documentation) requirements</u>:
   - → Free form

- Once AGAIN:

1. User Story: a simple description of a feature from

the user's perspective.

2. Use Case: detailed description of how a user interacts with the system.

Use Case:

1. Title
2. Main actor
3. Goal
4. Scope
5. Precondition
6. Stakeholders
7. Main Success Scenario
8. Success Guarantie
9. Acceptance Criteria

- Principles for good User Stories and Use Cases:
  1. Understandable
  2. Useful
  3. Appropriate size
  4. Estimable
  5. Independent
  6. Testable

- Level of Detail
  1. Under-Specified Requirements: critical deets are left out.
  2. Over-Specified Requirements: too many unnecessary deets causes time to be lost on finding solutions that may be too over-complicated

Finding the perfect level of detail is not easy!

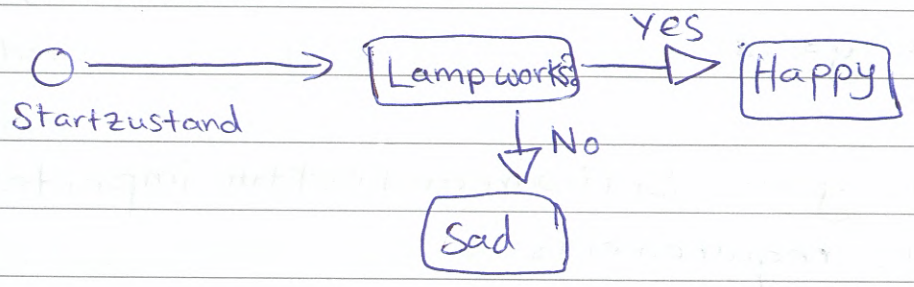
3. Illustration Requirements:

communication (using visual representation) between developers and stakeholders. For example, Use-Case diagrams, Flow charts, BPMN, ...

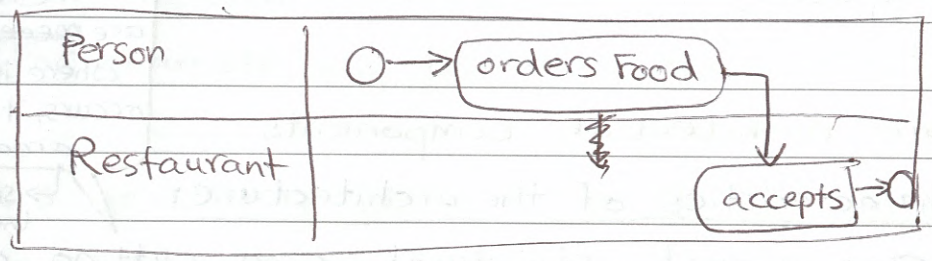
1. Use-Case diagrams:

shows how a user interacts with the system.

2. Flow Charts:



3. BPMN (Business Process Model and Notation):



4. Validation Requirements:

1. Acceptance test: for functional requirements
2. Acceptance Metrics: non-functional (performance)
3. Code Review.
4. Legal Review: complies with laws?

- User Story format:

As a <u>who</u>, I want to <u>what</u>, so that <u>why</u>

Kapitel 5

22.01.25

Softwarearchitektur

- Was ist Softwarearchitektur?

provides an abstract design of the software ~~and~~, how the software should be structured in each component, and how components interact with each other.

→ The recommended software architecture changes every 10 - 15 years.

- Why is good Softwarearchitektur important?

→ fulfills requirements
→ maintainable and extendable
→ high quality

- Software architecture components:

1. Characteristics of the architecture:
   ↳ Operational, structural, cross-cutting characteristics

   Cross-cutting are ~~means~~ areas in software where if a problem occurs, it affects other areas as well
   ↳ so parts that affect the entire software

2. Architecture style:
   ↳ Presentation layer, Application layer, Persistance layer, Database layer (PAPD)
3. Architecture decisions:
   ↳ e.g what frameworks?
4. Design principles:
   ↳ e.g avoid local caching

- ~~Ea~~ Developments that had an effect on Softwarearchitektur

1. Open source:

softwares whose source code are openly available to anyone to view, modify or distribute. For example, MySQL.

2. Open Standards:
softwares that use common rules or ~~rules~~ guidelines that anyone can view. This allows for interoperability and compatibility. For example: a USB has open standards because it can be used to charge your phone using your laptop regardless of which brands they are.

3. Container Technology:
softwares that are packaged together with their dependencies, so they are ready for use. (For example any libraries they might need)

- Let's go through some of the 4 Softwarearchitektur components in more detail:

1. Characteristics of Architecture (Dimensions of architecture)

1. Operational characteristics:
   a. Scalability : available to many users
   b. Availability : when for example?
   c. Performance : e.g how fast should software react?
2. Structural characteristics:
   a. Maintainability
   b. Extendability
   c. Reusability : components can be used again?
   d. Configurability: users can adjust software to their own needs

3. Cross-cutting characteristics:

a. Localization: supports different languages, currencies, etc.

b. Usability: for people with impairments

c. Data Protection

d. Security

2. Architecture Style:

→ Pipes and Filter

→ Layers

→ Service-oriented

→ Broker

→ Microservice

- more than one style is usually used in a software at the same time!

~~Pesce~~

2.1 Layers:

Presentation
Application
Persistance
Database

- Presentation layer: displays data
- Application layer: processes data according to the business logic rules.
- Persistance layer: converts data into a storable format
- Database layer: stores data

- Why do we separate the layers?

It allows for better testability and for the problems/errors in one layer not to interfere with the next layer.

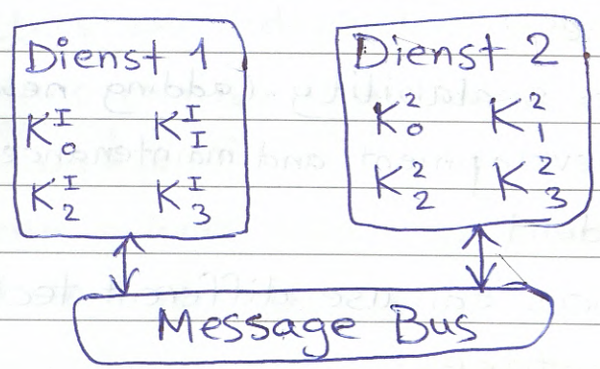
- What is Closed and Open layer?
  - Closed Layer: the request must pass through the layer
  - Open layer: the layer can be bypassed.

- Advantages and Disadvantages of Layers:

  - Advantages:
    1. Easy to understand and implement
    2. Splitting tasks makes it maintainable.

  - Disadvantages:
    1. No inherent vertical scalibility: if we wanted to add more servers to reduce the load (horizontal scaling), it would be challenging.
    2. Architecture-Sinkhole Anti-Pattern: a request passing through layers with minimal modifications means there is unnecessary code used for redirection.

2.2 Service-Oriented architecture:



- Each Dienst (service) is independent (low coupling)
  ↳ a service does not need to know which service(s) receive its request

- ~~Mess~~ They communicate using the Message Bus
- Encapsulation (the internals of a software are hidden from other services)
- Services can have their own database or share one with other services.
  - ↳ If one database, one service : format or database system can't be changed without problem.
- New services can be added (Scalibality)
  - ↳ new services can react ~~to~~ or respond to existing messages.
- Allows integration of third-party services, e.g payment processing.

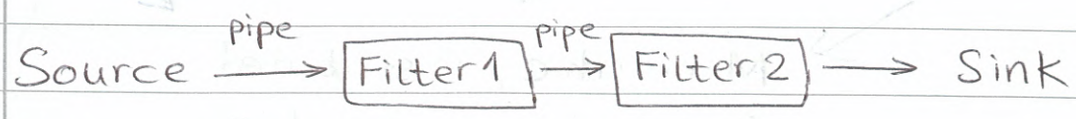
→ If shared database among services (2 or more): higher coupling, lower chance of inconsistency.

- Law of Conway : each team is responsible for a fixed set of services.

- Advantages and Disadvantages of Service-Oriented architecture.

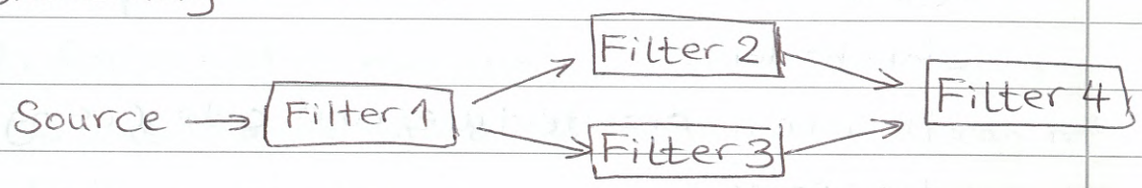
- Advantages :
  1. Allows scalability (adding new services)
  2. The development and maintenance of services is independent.
  3. Services can use different technologies.
- Disadvantages :
  1. Difficult to trace data ~~over~~ across multiple services
  2. Overhead for communication
  3. On a large scale, it is difficult to change a service without disrupting others.

2.3 Pipes and Filter



- Filters transform data. They are connected by pipes.
- Filters are independent and can work in parallel
- Filters are stateless and do not depend on past data (only current input)

Branching



- Filter 1 tests current input and redirects to either Filter 2 or Filter 3. Filter 4 processes ~~both~~ data from both Filters.

Advantages and Disadvantages of Pipes and Filter:

-Advantages:

1. Filters can run in parallel
2. Filters are independent so can split tasks
3. Filters can work on small data blocks, reducing memory requirements

-Disadvantages:

1. Some filters may require all data and can not work on data blocks (e.g sorting)
2. Error-handling partially processed data may be challenging because the source code needs to be retained.
3. Intermediate serialization and storage is resource intensive.

Serialization: Converting data into a storable format

Architecture characteristics

Requirements

Domains

Monolithic / Distributed
Data storage location
Communication style

Development process

Organisational factors

Important:

- Good communication between developers and stakeholders.
- Document architecture and code (using text or diagrams)

Übungsblatt 5

Aufgabe 1,

(a) Rate (poor, medium, good) the Elasticity, Reliability, and Modularity of ~~Soft~~ the architecture style 'Layers'.

1. Elasticity : poor
   ↳ layers architecture lacks features that facilitate dynamic adaptation, as there are no rules for division or independence of components within a layer.

2. Reliability : medium
   ↳ easy to understand and implement but can not run tasks in parallel.

3. Modularity : medium
   ↳ separates technical layers but doesn't organize them according to what they do.

(b) Compare Fault Tolerance and Scalability (as better or worse) in 2 Variants A, B ~~who~~ that use service-oriented architecture. However, A uses a shared database for all services, while B has a database for each service.

1. Fault Tolerance (A worse than B)
   ↳ If database (in A) crashes, all services are affected.

2. Scalability (A worse than B)
   ↳ In A, one database has to handle all requests, which can limit performance.

Aufgabe 2,

1. Localization : not relevant
   ↳ Target audience is all around the same area (Duisburg) so same language and currency.

2. Security: not relevant
   ↳ customer personal info is not stored and payments are handled by a third party.

3. Configurability: relevant
   ↳ partner stores need to be able to integrate sales into their own websites

4. Accessibility: relevant
   ↳ limitation of users unknown

5. ~~Av~~ Scalability: relevant
   ↳ system already plans to expand the customer base.

Kapitel 6
Domain-Driven-Design

- Domain-Driven-Design (DDD):
  - → develops software close to the problem domain.
  - → aims to reduce cruft (useless code / partially finished code that currently serves no pupose)
  - → better communicatëon with Stakeholders.

- Domänen-Modellierung / Domain Modeling:
  the process of creating the Domain Model by identifying:
  - → Domain Concepts (Customer, Seller)
  - → Generalizing Domain Concepts (grouping/categorizing)
  - → Eigenschaften of Concepts (Attributes / Associations)
  - → Derived Eigenschaften (e.g Total Price)

- Identifying Domain Concepts:
  1. Speak with experts
  2. Identify Nominalphasen (nouns), e.g customer, business, etc

- Verhaltensmodelle:
  A model that additionally describes the behavior of concepts.

  - Zustand / State:
    - → entry : action on entry
    - → do : action while in state
    - → exit : action on exit

  - There are ~~tg~~ Start states, End states, Terminal state (shows error in execution)
  - Guard is a boolean expression

Kapitel 7
Designprinzipien

- Design Principles:

  Some help manage complexity:

  1. KISS (Keep It Simple, Stupid!)
  2. Separation of Side Effects
  3. Avoid Ambigious Data types
  4. LSP (Liskov Substitution Principle)

  others ~~help~~ minimize the impact of changes:

  1. DRY (Don't Repeat Yourself)
  2. SRP (Single Responsibility Principle)
  3. ISP (Interface Segregation Principle)
  4. High Cohesion, low Coupling

1. KISS

   helps reduce intended complexity.

- Reasons for unintended complexity:

1. Side Effects
2. Ambigious data types
3. Inexperienced developers
4. Lack of documentation
5. Poor architectural decision
6. Premature Optimization
7. Software Evolution (no time to change outdated code)

- Metrics to measure complexity:

  1. <u>Fan-in</u>: number of functions that call measured function

2. Fan-out: number of functions called by measured function

3. Code size

4. Cyclomatic Complexity: number of independent paths in a function

5. Nesting depth: depth of a nested function, e.g if statement

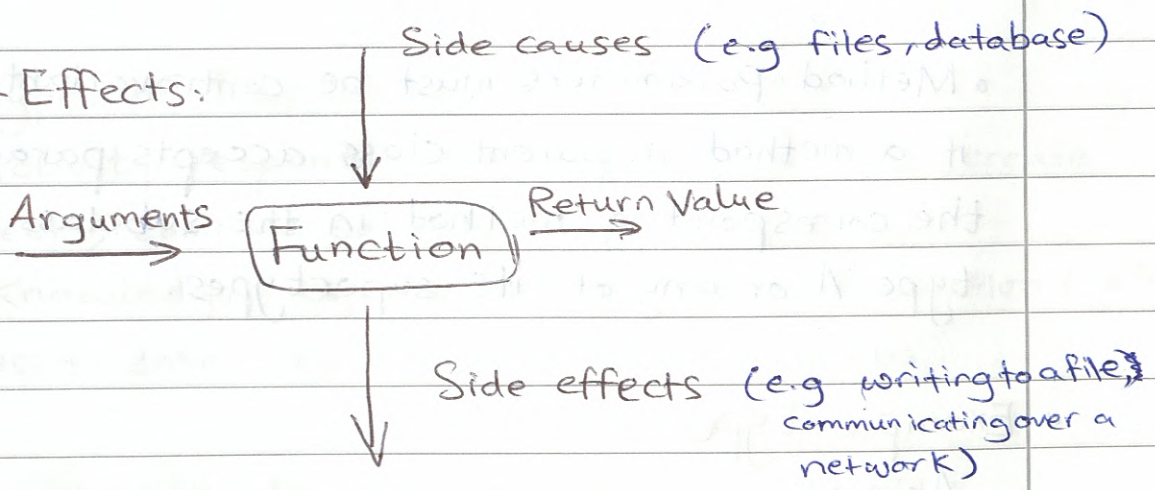
6. Coupling: ~~dep~~ number of dependencies in a class.

- The afore-mentioned metrics are all static analysis (do not take time into consideration)

- Static Analysis:
measuring metrics at a specific point in time. The goal is to find outliers.

- Longitudinal Analysis:
track how metrics change over time. Patterns can be identified, e.g certain actions make the software more complex.

2. Side-Effects:



- Separation of Side-Effects:
  ~~Ber~~ "purer" code (with less side effects) is more easily understandable. For example, clearer method names with more distinct functionality.

→ make datastructures unchangeable where possible to avoid destructive changes.

3. Avoid ambigious data types:
   like int, boolean, String

4. LSP (Liskov Substitution Principle)
   also known as starkes verhaltensbasiertes Subtyping (strong behaviorial subtyping). Types must be replaceable by their subtypes without affecting the program.

   - Method Return types must be covariant:
     if a method in parent class ~~retup~~ returns a type A, the corresponding method in the subclass must return type A or any of its subtypes.

   - Method Parameters must be contravariant:
     if a method in parent class accepts parameter type A, the corresponding method in the subclass must accept type A or any of its supertypes.

- Example type:
  Animal
  Subtype: Dog
  Supertype: Living Thing

- No new exceptions allowed, only subclasses of already thrown exceptions.

- Preconditions must not be strengthened in the subclass
- Post conditions must not be weakened in the subclass
- Invariants must not be weakened
  ↳ Invariant is a condition that must always hold true for an object during its lifetime.

5. DRY (Don't Repeat Yourself)
reduce repetition because in case change is required, many instances would have to be changed. Forgetting to change an instance would lead to an error.

→ use variables instead of repeating expressions.
→ use wrapper methods (e.g open_file_read_only(file) instead of open_file(file, "r"))

6. SRP (Single Responsibility Principle)
every class/module should have exactly one responsibility.

- 2 Types of Responsibilities:
  1. Activity responsibility: performs a task (create object, compute something)
  2. Knowledge responsibility: provide information (allow access to data, maintain references to objects)

7. High Cohesion, Low Coupling
- Coupling: dependency between classes/modules. High coupling ⇒ low modularity. Changes to one class can affect other classes. Classes are difficult to understand independently.

- X is coupled with Y if:
  - → X has an attribute that references Y
  - → X contains an (partial) expression of Y
  - → X is a subtype of Y
  - → X calls a method from Y
  - → X uses instances from Y

- CBO (Coupling Between Objects):
  a metric for measuring coupling of a class by counting the number of classes it is coupled with.

- Cohesion:
  describes the relatedness of a class's methods.

- LCOM (Lack of Cohesion Of Methods):
  quantifies cohesion of a class. The higher the LCOM, the lower the cohesion.

- LCOM(c) $\in [0, n]$ where 0 is perfect cohesion and n is no cohesion (each method is independent)

- Beispiel 1 Code:
  - → contains(A item) accesses both tail and value (look at if statement)
  - → Graph has one connected component because all methods are indirectly connected through contains(A item)

LCOM = 1 (out of 3)
High cohesion

$$LCOM = 1 - \frac{\text{no. of connected pairs}}{\text{Total no. of method pairs}}$$

8. ISP (Interface Segregation Principle)

→ larger interfaces should be split into smaller ones (lower coupling)

→ ensures users of an interface only need to know the functions they require.

$$LCOM = 1 - \frac{\text{number of connected pairs}}{\text{total number of method pairs}}$$

$$\text{Total method pairs} = \frac{n(n-1)}{2}$$

where n is number of methods

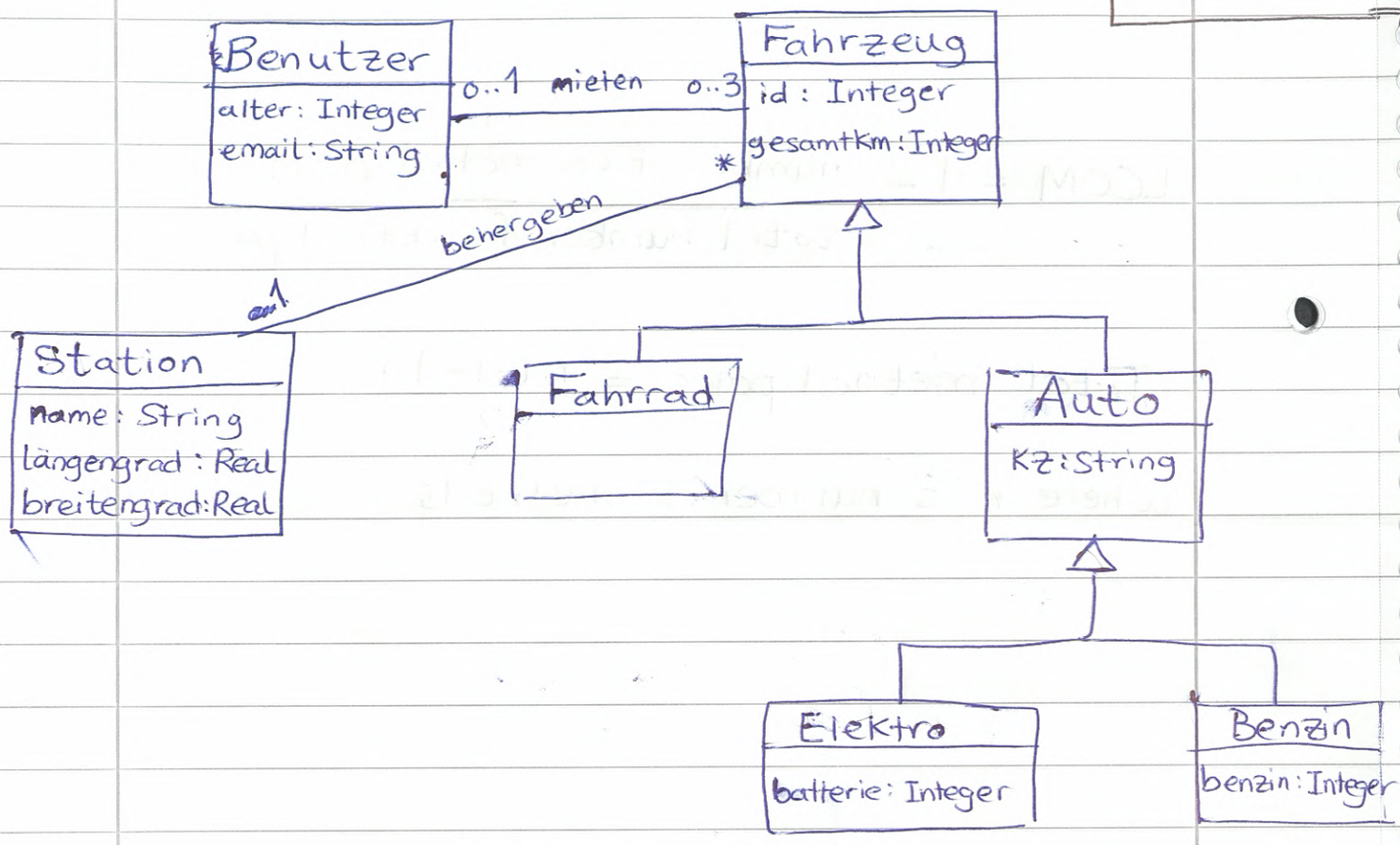
Übungsblatt 6

23.01.25

Capital or small letter
Class – capital
Attribute – small
datatype – big

Aufgabe 1,

- Create a UML class diagram.

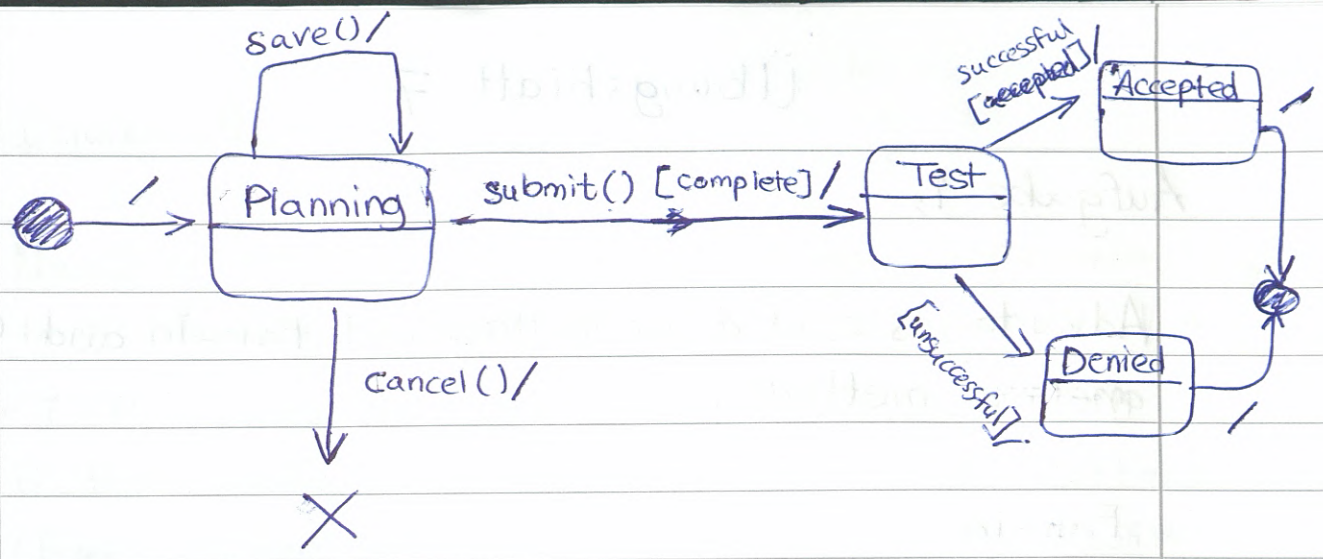


Aufgabe 2,

- UML-Zustandsdiagram erstellen

→ The / Notation: used to indicate actions that are triggered as part of a transition when entering/exiting a state.

→ The [] Notation: indicate guard conditions which are logical conditions that must be satisfied for a transition to occur.



Aufgabe 3,

check that multiplicities match

e.g each club at least 3 persons

Übungsblatt 7

Aufgabe 1,

- Advantages and disadvantages of Fan-in and Codesize metric methods.

  - Fan-in:
    - Pro: gives an idea how important a function is and how carefully changes to it should be tested.
    - Con: only works if we know which functions call which others, which is difficult in some systems.

  - Code size:
    - Pro: gives an idea of the size/complexity of a project
    - Con: unreliable as different programmers write code very differently.

Aufgabe 2,

- Explain in LSP what it means that the return type is covariant and parameter type is contravariant.

- In a subclass, if a return type of a method is changed, it should always be the same (as the corresponding method type in the class) or "more specific" (i.e subtype).

  And ~~if~~ a parameter type should always be the same or "more general" (i.e supertype).

Aufgabe 4,

- Name a measure you can take if your application has high coupling.

→ ISP (Interface Segregation Principle)
if the calling class interacts with a smaller surface (fewer methods/functions), it becomes coupled with less code, reducing overall coupling

Kapitel 8
Entwurfsmuster
(Design Patterns)

- Entwurfsmuster:
  → ~~common~~ proven solutions for common programming problems.
  → used to avoid wasting time/effort rediscovering solutions to known solved problems.

  It helps:
  - improve communication with other developers
  - understandable code
  - complexity, coupling, reusability of structure

- Decorator Pattern uses delegation and inheritance
- RAII Pattern uses automatic memory management.

- Disadvantages of Entwurfsmuster:

  1. Boilerplate: often requires code which does not directly lead to solving the problem domain.
  2. Unnecessary use: simpler solutions may exist.
  3. Dogmatism: following the Entwurfsmuster too strictly, instead of tailoring it to the program.

- Types of Entwurfsmuster:
  1. Erzeugungsmuster (e.g Abstract Factory)
  2. Strukturmuster (e.g Decorator)
  3. Verhaltensmuster (e.g Iterator, Observer)

1. Erzeugungsmuster: Objekte erzeugen
2. Strukturmuster: structure of components
3. Verhaltensmuster: interaction between classes.

1. Abstract Factory Pattern: (Slide 98)
→ Erzeugungsmuster
→ delegates Objekterzeugung to subclasses.

- Abstract factory: declares methods for creating products
- Concrete Factory: implementation with defined methods (creates objects)
- Abstract Product: defines common interface for a family of product
- Concrete Product: implementation of abstract product interface.

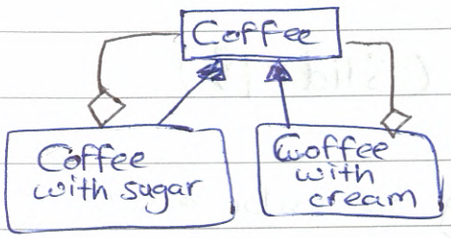
- Advantages and Disadvantages of Abstract Factory Pattern.

- Advantages:
  - ensures consistency between products
  - exchanging family products is easy (change one line of code Concrete factory)
  - adding new family products is easy
  - abstracts the creation of concrete products.
- Disadvantages:
  - unconventional way of creating objects
  - difficult to add new product (abstract factory and its subclasses may need to be angepasst)

2. Decorator:
→ Strukturmuster
→ allows you to add new functionalities to an object dynamically without altering its original class, instead of

creating subclasses using inheritance, the object is "wrapped" (decorated) using delegation (composition).

- Inheritance limits scalibility



Can't it be both?

- Component: the interface that is to be extended, e.g coffee
- Decorators themselves implement the component interface.
- BasicComponent implements the basic functionality without extension (so a coffee without (sugar & cream))

- Advantages and Disadvantages of Decorator:

  - Advantages:
    - flexible
    - reusable
    - high modularity
  - Disadvantages:
    - difficult to be independent of the order of decorators.
    - Boilerplate for methods that are not extended.

3. Iterator:

allows elements in a collection to be accessed sequentially, ~~to~~ hiding the complexity of a system, e.g switching between channels on a radio.

→ Verhaltensmuster

- Advantages and Disadvantages of Iterator:

- Advantages:

1. hides complexity of datastructure
2. avoids traversal errors

- Disadvantages:

1. difficult to parallelize
2. Implementation may be difficult
3. has many states
4. Problems dealing with changes to the underlying data-structure.

4. Observer: (Slide 21)

→ Verhaltensmuster / behavioral design pattern

→ allows changes to an object to be observed without creating coupling.

- Subject, Concrete subject, observer, concrete observer

- Advantages and Disadvantages of observer design pattern:

  - Advantages:

    1. enables broadcast communication
    2. Coupling is only abstract between the subject and observer

  - Disadvantages:

    1. risk of update cascades in case of a change
    2. Notifications are sent to all observers even if they are not interested

- Examples

→ Monoid, Monade, Foldable, Category, Functor

4.1 Monoid

an algebraic structure used for accumalition. It has:

- → a binary operation
- → an identity element (like 0 or "" (one ~~that~~ does not change the total value when added)
- → fulfills Associativität
- → e.g Strings, lists, numbers, boolean values

4.2 Monade

extends types for additional functionality

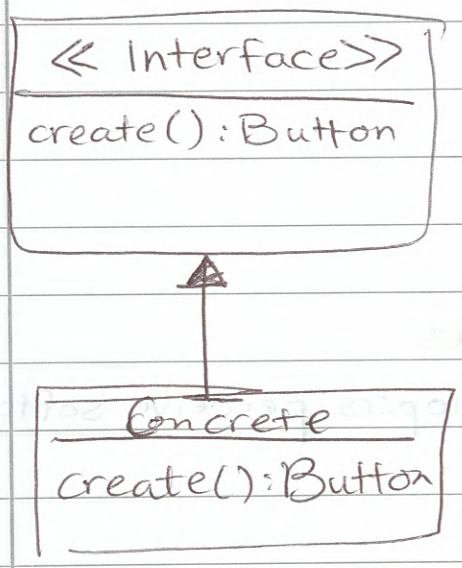
- → Assoziativität
- → e.g Option, Try, IO, Log, Lists

Übungsblatt 8

Aufgabe 3,

- Difference between Structural and behavioral design patterns.

  - Structural design pattern : how classes and objects are organized, ~~and~~ ~~e.g~~ e.g decorator, adapter, ...
  - Behavioral design pattern: how objects interact and communicate with each other, e.g observer, iterator, ...



Kapitel 9
Software Quality

- Software Quality:
  measures how well the software is perceived by stakeholders (external) and developers (internal)

- External Software Qualities:
  factors influencing how stakeholders perceive software

  1. Functionality
  2. Validity (behaves as expected)
  3. Responsiveness
  4. Compatibility
  5. Efficiency
  6. Portability
  7. Usability
  8. Reusability

- Internal Software Qualities:
  factors influencing how developers perceive software

  1. Understandability
  2. Maintainability
  3. Extensability
  4. Enjoyability

→ Internal software quality is very important in the later stages of development

- Quality Assurance (QA) / Qualitätssicherung
  measures to ensure software quality

  - External Software Quality:
    1. Continuous Stakeholder Feedback
    2. Domain Modeling
    3. Testing
    4. Static analysis

  - Internal Software Quality:
    1. Architecture
    2. Design principles and metrics
    3. Design Patterns

Kapitel 10
Testen

- Testing:
  helps ensure that the software does what it is supposed to do.

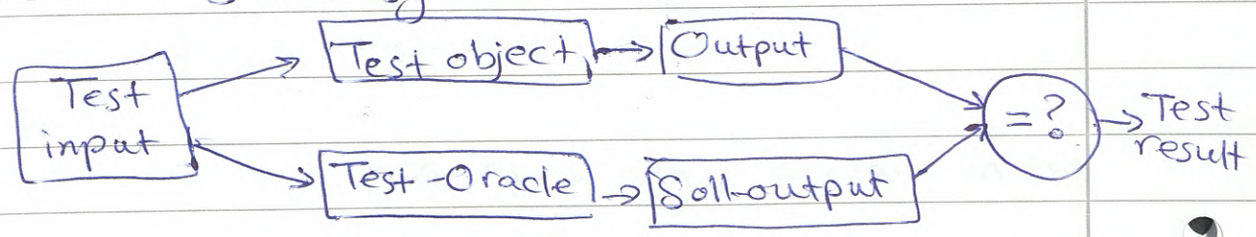
  Testing ≠ finding errors, debugging

  → helps ensure requirements are fulfilled
  → helps identify bugs and security issues

- 'Debugging'
  localization of error states and (potentially) error correction.

- Testfall / Test case:
  → Precondition: state system should start in
  → Post condition: state system should end in
  → Inputs
  → Expected outputs
  → Test instructions

- Test Oracle:
  we need Soll-Ausgaben



→ automatically generated from specification

→ Problem: even expected results may contain errors. So when there are deviations between actual and expected output, the expected output must also be reviewed.

~~Test levels~~

- Exhaustive testing:
  verification of all possible input combinations
  → usually not feasible

1. Representative
2. Error-prone
3. Redundancy-free
4. Economical

- Types of Tests:

1. Execution
   → manual: by human
   → automatic: by machine
2. Technique
   → Example-based: ~~gen~~ use example inputs
   → Property-based: use generated inputs
3. Perspective on Test object
   → Black box: ~~doesn't~~ disregards. take code & internal processes
   → White box: takes code and internal processes into account
4. Test Input Source
   → Function-oriented / Specification-based: derived from specification
   → Structure-oriented / Source-code-based: derived from code structure

- Test levels:

- By the Software Component being tested:
  1. Component Test
  2. Integration Test
  3. System Test
  4. User Interface Test

- By Purpose of Test:
  1. Acceptance Test
  2. Accessibility Test
  3. Usability Test
  4. Smoke Test (checks for major issues)
  5. Performance Test
  6. Security Test

1. Component Test:
testing one small piece of software, like a single function, class or module to make sure it works by itself. It focuses on functionality and robustness, possibly also non-functional requirements like performance.

- Test environment: is where the testing takes place. Usually happens near where the component is being built.
- Test Driver: helper program that runs the code we are testing and checks its behavior.
- Stub: A placeholder for components required by the component being tested.
- Test Bed: everything we need to run, evaluate and log the tests.

2. Integration test:

tests how two or more components interact.

Test environment:

1. Test drivers / Stubs: reused from component tests
2. Monitor: to track data movement between components

- Important concept to understand (not in slides)

1. High-level components: these depend on lower-level components but are not called by any lower-level components, e.g user interface.
2. Low-level components: these do not depend on other components but are called by other components, e.g database, libraries.

- Integration strategies:

1. Top-down: starts testing from the top. If a lower-level component is called, which hasn't been implemented or tested yet, we use stubs.
2. Bottom-up: starts testing from the bottom. If the lower-level components provide data or services to a higher-level component that isn't ready yet, test drivers are used.

3. System test:

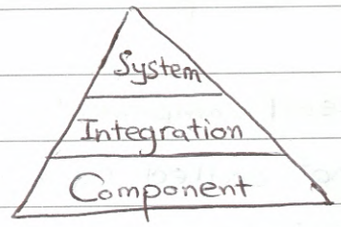
tests the entire system to check that functional and non-functional requirements are fulfilled.

- Test environment:

  should mimic the production environment closely but not be identical to the customer environment to ensure safety, e.g when testing payment features, entered "made-up" data could match someone's actual data.

- 50% of Quality Assurance happens at this stage!

- Test Pyramid



  → large number of component tests
  → fewer integration tests
  → small number of system tests

  aims to find most errors as cost-effectively as possible

- Example-Based vs Property-Based Testing:

1. Example-Based Testing:
   → use example inputs and evaluate the outputs
   → most common testing techniques
   - Advantages:
     1. checking outputs is straightforward
     2. Creating test inputs is easy
   - Disadvantages:
     1. Test quality heavily depends on selected inputs.
     2. Rare cases can be forgotten (when thinking of inputs)
        (Edge cases)

2. Property-Based Testing:
   → tests ~~code~~ to ensure certain properties are true.
   → Test Frameworks are used to generate input
   → a ~~count~~ minimal counterexample is provided if a test fails

- Commonly-tested Properties:

  - Unary functions: accept only one input

    1. Totality: sort(X) defined for all X, produces no errors
    2. Injectivity: ~~eve~~ $f(X) == f(Y) \Rightarrow X == Y$
    3. Surjectivity: for all y exists an x : $f(X) = Y$
    4. Idempotency: sort(sort(X)) == sort(X) doing something multiple times is the same as doing it once.

  - Inverse functions:

    1. Isomorphism: two functions can undo each other perfectly, e.g deserialize(serialize(X))

  - Binary Functions:

    1. Identity: X ++ "" == X
    2. Communitativity: X ++ Y == Y ++ X
    3. Associativity: X ++ (Y ++ Z) == (X ++ Y) ++ Z

  - Binary Relations:

    1. Reflexivity: X == X
    2. Symmetry: $X == Y \Leftrightarrow Y == X$
    3. Transitivity: $X == Y, Y == Z \Rightarrow X == Z$

- Custom generators are used to generate inputs that meet a specific requirement.

- Advantages:

  - more thorough because of many generated inputs
  - more likely to uncover Edge cases.
  - minimal counterexamples are automatically generated

- Disadvantages:
  - evaluating outputs may be difficult without an external test oracle (which we can't use here because inputs are generated)
  - not always easy to know which properties to test
  - only applicable to mathematically expressable relationships

- Testing shows the presence of bugs but never their absence!

Übungsblatt 9-10

24.01.25

Easy Peasy Lemon Squeezy

- Disadvantages:
  - evaluating outputs may be difficult without an external test oracle (which we can't use here because inputs are generated)
  - not always easy to know which properties to test
  - only applicable to mathematically expressable relationships

- Testing shows the presence of bugs but never their absence!

Übungsblatt 9-10

24.01.25

Easy Peasy Lemon Squeezy

Kapitel 11
Statische Analyse

24.01.25

Testing methods

- Fuzzing:
  generate inputs to trigger errors / bugs

  Pro: find bugs that traditional ~~errors~~ testing methods may miss

  Con: also can not detect the absence of errors

- Formal Verification:
  experts write mathematical proofs to verify software properties.

  Pro:
  - can prove absence of errors with certainty

  Cons:
  - still difficult to determine which property to prove
  - very time-consuming
  - specific to a particular program

- Dynamic Analysis:
  uses real ~~data~~ inputs to collect and analyze data during program execution.

  Pro:
  - uses real execution with real inputs

  Cons:
  - finite number of executions (time issue/complexity)
  - selecting inputs may be difficult

- Monitoring:
  ~~passively~~ collecting data during actual software usage and observing for deviations.

  (Own thoughts in my own words): dynamic analysis is getting testers active users to use the software, monitoring is maybe like using cookies to consensually get user data during software use.

  Pros:
  - based on real execution
  - detects relevant problems

  Cons:
  - problems are only detected after they start affecting usage.

- Static Analysis
  gathering information about a program without executing it

  Advantages:
  - Fully automated
  - Strong gurantees
  - Covers all possible execution paths (even those that may not occur during runtime)

  Disadvantages:
  - some properties depend on the runtime therefore can not be verified
  - some problems only arise upon execution, e.g the halting problem

- used for example in IDEs to suggest improvements

- Challenges of Statistical Analysis

1. Undecidability:

Program outcomes: True/False Positive, True/False Negative

Approaches to undecidability:

→ no definitive answer output (skipping unclear parts)

→ Overapproximation (potential false positive)

→ Underapproximation (potential false Negative - dangerous)

2. Completeness:

does n't produce results that are impossible at runtime

→ No overapproximation

$$\text{Precision} = \frac{\#\text{True-Positives}}{\#\text{Total Results}} \qquad \frac{TP}{TP+FP}$$

Total number of issues

3. Soundness:

considers all possible runtime results ~~are consid~~

→ No underapproximation

$$\text{(Trefferquote) Recall} = \frac{\#\text{True-Positives}}{\#\text{Expected Results}} \qquad \frac{TP}{TP+FN}$$

Total number of actual issues (True Positives and False Negatives)

→ vollständig soundness is difficult to achieve because of loaded dynamic code, reflection (run-time behavior difficult to predict)

4. Scalability:

producing results within limited time/memory

→ example goal: fast results in IDE

- Precision, Recall and Scalability are conflicting properties. Improving one may degrade another!

- Types of Analysis:

1. Control Flow Analysis

describes the order in which instructions are executed

2. Data Flow Analysis

tracks the flow of values through a program

- Properties of Static Analyses:

1. Scope:
   - Intraprocedural: examines each ~~each~~ method individually
   - Interprocedural: takes entire program into account
2. Flow Sensitivity
   - flow insensitive: compute a globally valid value
   - flow sensitive: compute a value for each individual method
3. Context Sensitivity:
   - context-insensitive: consider each part of code only once
   - context-sensitive: use multiple contexts (e.g functions calling other function)

- Analysis Techniques:

1. Type System:

→ assigns a type to every expression

→ an expression's type is derived from its subexpressions (compositional)

→ used to detect errors during compilation (it's like translation of code)

2. Effect System:

→ assign an effect to each expression

→ an expression's effect is derived from its subexpressions

→ less commonly used

→ e.g 'throws' clause in Java

3. Monotone Data Flow Analysis:

analyzes how data flows between variables

→ Intraprocedural

It considers:

1. Liveness: can a value be read after given point?
2. Reaching Definitions: where do values originate?
3. Available Expressions: has this been calculated before? avoids repeated calculations

→ used for optimization, e.g removing dead-code (unreachable code), moving loop-invariant code (unchanging code moved outside of loop).

4. Interprocedural Data Flow Analysis (IFDS/IDE)

treats interprocedural data flow as an Erreichbarkeits-Problem

→ efficient for suitable problems

→ good for analyzing tainted (dangerous) data.

5. Symbolic Execution:

→ interprets programs with symbolic rather than concrete values

→ splits at branches to consider all paths (path explosion)

6. Big-Step Abstract Interpreter:

→ looks at the bigger picture by using approximations instead of real values.

→ groups/merges results together to avoid path explosion

→ uses signs, ranges ([1,∞])

7. Constraint-Based Analysis:

→ generates rules (constraints) for the program and solves them to better understand the program

→ may be difficult for complex constraints

8. Slicing:

extracts program statements related to a <u>slicing criterion</u>

- Backward Slicing: what parts influence the slicing criterion.
- Forward Slicing: what does the slicing criterion influence?

- Important Analyses:

1. CFG (Control Flow Graph):

describes which instruction is executed directly after

2. Dominator Tree:

shows which instructions ~~be~~ must have been executed before a specific instruction.

3. Post-Dominator Tree:

shows which instructions must be executed after a specific instruction.

4. Call Graph:

- Connects a caller method to the methods it calls (callees)

→ required for interprocedural analyses

5. Points-To Analysis:

determines which object a variable (or field) can reference

→ often traces location where object was created

→ useful for building call graphs, alias analysis (do 2 objects have same location), taint analysis.

1. Type Abstraction (low precision)
2. Allocation-Site Abstraction (mid)
3. Shape Abstraction (high precision)

6. Escape Analysis:

checks if an object can be accessed outside its restricted context.

→ useful for understanding lifetime of objects

7. Alias Analysis:

examines relationships between variables

- "May" Analysis: determines if 2 variables may point to the same object (often flow-insensitive)
- "Must" analysis: determines if 2 variables always point to the same object (always flow sensitive)

8. Immutability:

determines if a field/object/class can be modified after initialization.

9. Purity:

checks if a method is deterministic (same input always produces the same output) and free of side effects

→ keeps code safe and maintainable

→ can be enforced via a type system

→ supported in modern programming languages

Kapitel 12

27.01.25

Test Coverage

- Test Coverage:
  shows how much of the code has been tested and how much hasn't to ensure everything works properly.

- ① Black box / Specification-based testing
  ② White box / Source-code-based testing

① Black box / Specification-based testing (check later)

- Aquivance class:
  It's impossible to try out all combinations of inputs so we use aquivalence classes.

  Inputs in the same aquivalence class, should have the same test result (pass/fail)

  $a \sim b$ means a and b are in the same aquivalence class.

- Testing becomes easier this way because you just have to test aquivalence classes instead of individual inputs and just prove an input belongs to an aquivalence class.

- Valid/Invalid aquivalence class: whether or not it fulfills a specification, e.g $x \leq 42$ given:
  $x \leq 42$ valid , $x > 42$ invalid

Total number of test cases = product of no. of valid aquivalence classes for each input parameter

② Whitebox / Sourcecode - based testing

Techniques:

1. Control - flow - oriented testing: tests how data flows from one instruction to another
   - → Structure - based test coverage: all lines run at least once.
   - → Loop-testing:
   - → Logic - based test coverage: logical conditions are tested

2. Data - flow - oriented testing: how data moves through the program.
   - → tracking use of variables

① Blackbox techniques:

1. Aquivalence ~~test~~ class tess
2. Cause - Effect analysis
3. State-based test

Back to Whitebox testing:

$$\text{Test Coverage} = \frac{\text{number of tested elements}}{\text{total number of elements}}$$

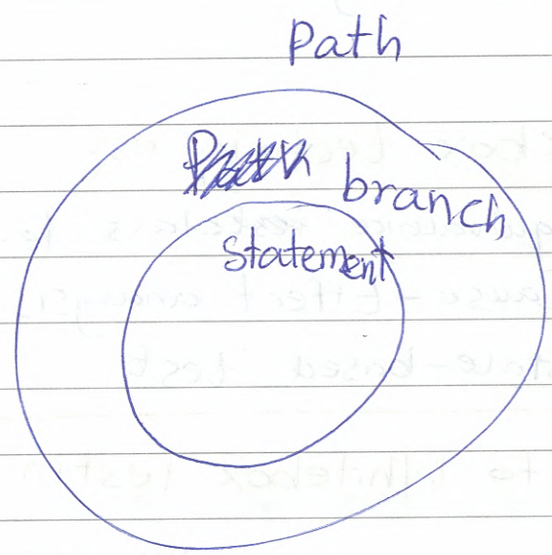
- CFG (Control Flow Graph)
  connected graph that shows the execution flow of a function or method.

- A valid path: starts at entry node, ends at exit node, and does not have to have all nodes

$$\text{Statement Coverage} = \frac{\text{number of executed statements}}{\text{total no. of statements}}$$

- 100% branch coverage ⇒ 100% statement coverage but reverse is not true.
- 100% path coverage ⇒ 100% branch coverage but reverse is not true

Path ⇒ branch ⇒ statement



Übung 12

Aufgabe 1,

Number of Test Cases = ~~6~~ 1 x 1 x 3 = 3 ✓

~~only ca~~

Age → Experience → C1, C2, C3 } 3 test cases

3 + each invalid case tested once individually

= 3 + 4

= 7 test cases

Aufgabe 2,

(a) skip else condition cause nothing to be checked there.

Kapitel 13
Performance

- Poor Performance issues:
  → reduces productivity
  → increases cost

- System Performance: the useful work a system performs across multiple processes.
- Application Performance: the useful work a single application performs.

- When should Perfomance be improved?
  ① Measurable problem exists
  ② Stakeholders are affected

- Common Performance Metrics:
  1. Execution time (runtime)
  2. Throughput
  3. Number of threads
  4. CPU usage and memory consumption
  5. Number of disk I/O operation
  6. JIT (Just In Time) Optimization: dynamic improvements during execution.

- Execution time: time required to complete a task. It depends on some factors, e.g: System load, memory usage, JIT optimizations.

- Execution time management must be repeated to obtain more reliable average results.

- Real-time: actual elapsed between between start and end of task
- User-CPU time: time spent executing application code in CPU
- System-CPU time: time spent executing kernel code in CPU

- Single-Threaded Applications:
  runs ~~an single~~ one CPU core at a time.
  → User CPU time cannot exceed real-time (CPU ≤ real-time)

- Multiple-Threaded Applications:
  Can run on multiple CPU cores at a time.
  → User-CPU time > Real-time

- Benchmark:
  executes parts of a program and measures behavior.

- ~~Variance~~
- Why do Benchmark results vary?
  - Garbage Collection (GC)
  - System Load
  - ...

  Solutions:
  - Allow time for garbage collection
  - run in isolation
  - repeat execution

- CPU Usage:
  No. of CPU cycles needed to execute a ~~each~~ task.
  → can exceed 100% in multi-core usage

- Memory: ~~Consumption~~
  - → measured in Bytes (8 bits)
  - → SI (decimal) prefixes 10^n, or IEC (binary) prefixes 2^n

- RAM (working memory) uses binary prefixes
- Storage devices (hard drives, SSDs) use decimal prefixes

- Virtual Memory:
  - → separate for each process(es)
  - → OS maps Virtual ~~memory~~ addresses onto physical memory (RAM)

- Resident Memory:
  physically memory currently in use
- Shared Memory:
  memory shared between multiple processes
- % MEM:
  percentage of total system memory used by ~~program~~ process

- Throughput (Durchsatz):
  number of specific operations performed in a given time frame. (current system load)

- Bandwidth (Bandbreite):
  theoretical maximum throughput a system can handle.

- Some JIT Optimizations:
  - remove null checks
  - remove loop-invariants outside of loop

- Time Complexity:

measures how runtime changes depending on the input size.

  - Worst-Case
  - Average - Case

Kapitel 14
Estimate & Prioritize

- Predictability is very important!

- Reasons for Uncertainty in Software processes:
  1. Constant changes
  2. Imprecise Information
  3. Unclear understanding

- Software must be continuously updated
  Prioritization:
  1. Corrective maintenance: debugging
  2. Perfective maintenance: improving non-functional properties
  3. Adjustive maintenance: due to changing conditions
  4. Evolving functionality

- Goal (Ziel): fixed value determined by external factors (budget, deadline)

- Purpose of Estimates:
  to assess whether goals set for the project are realistically achievable.

- Cone of Uncertainty:
  defines maximum achievable accuracy of estimates during the project lifecycle.

- Benefits of Good Estimations:
  1. Early Risk detection
  2. Prevention of last minute issues
  3. More professional decision-making

- Anchoring Effect:
  a cognitive bias where people rely too much on given info when making estimates, e.g "am I 18 years old? estimate my age"

- Priming:
  our decisions are influenced by context and subtle cues, often unconciously, e.g location of work.

- Confirmation Bias:
  searching for infos (subconciously) that confirm our pre-existing beliefs.

- Quantification Problems:
  people struggle to scale values properly.

- Objectification Strategies:
  → get multiple independent opinions on estimates
  → ensure no biases between participants.

- Prioritization Criteria:
  → Financial value
  → Costs
  → Customer Satisfaction

- MoSCoW Prioritization:
  - → Must-Have
  - → Should-Have
  - → Could Have
  - → Won't Have

- Rocks, Pebbles, Sand Technique: prioritize big tasks first, then fit in smaller ones
  - Rocks: large, important features
  - Pebbles: medium-sized enhancements
  - Sand: small fixes and adjustments

Wiederholungsblatt 30.01.25

Aufgabe 11,

(a) • Static Program analysis:

executed without the program. Code is tested for errors.

Pros: error detected before execution

Con: issues that only arise during runtime are not detected.

• Dynamic Program analysis:

program is executed and its behavior is observed.

Pros: real-time issues are detected

Cons: ~~[illegible] issue~~ – limited no. of executions

(?) – issues only detected when they have occured

is this monitoring

(b)

(Soundness)

1. Correctness: interval contains all possible run-time values (or more, it doesn't matter as long as it contains!)

(Precision)

2. Accuracy: interval excludes unnecessary values while still including all actual ~~values~~ runtime values. However, having the smallest possible range takes priority here.

Percent of correct runtime values captured in an interval:

$$\frac{\text{Correctly captured values}}{\text{Total values in interval}} = \frac{\text{True positive}}{\text{True positive} + \text{false positive}}$$

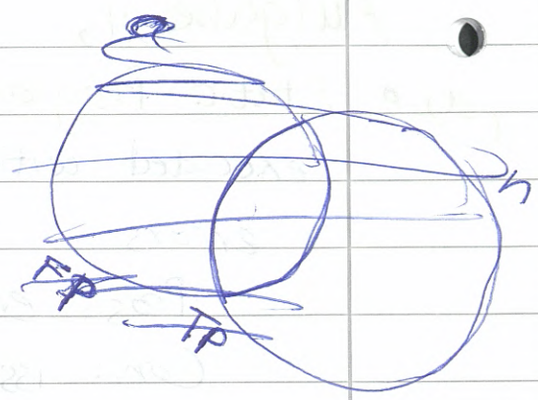
Correct runtime values: $a = \{10, 9, 8, 7, 6, 5\} = 6$ vals, $n = \{10, 8, 6, 4, 2, 0\} = 6$ vals

Total = 12

Ergebnis 1: $\frac{12}{6+11} \approx 71\%$

Ergebnis 2: $\frac{11}{6+9} \approx 73\%$

Ergebnis 3: $\frac{12}{11+11} \approx 55\%$



(c)

$$\text{Recall (Trefferquote)} = \frac{\text{True positive}}{\text{~~expected~~ results}} = \frac{TP}{TP+FN}$$

~~Graph~~

1. Actual Runtime Call Graph: includes only methods that actually happend. This is the ground truth.

2. Static Call Graph: includes predicted method calls without running the program (generated ~~th~~ using static analysis). It includes extra calls that never happened (false positives).

$$\text{Accuracy} = \frac{\text{correctly predicted}}{\text{total predicted}} = \frac{TP}{TP+FP}$$

mainly ~~only~~ looks at static call graph

$$\text{Recall} = \frac{\text{correctly predicted}}{\text{Total in actual runtime}} = \frac{TP}{TP+FN}$$

mainly looks at actual runtime graph

Aufgabe 1,

1. Centralized Version Control System:
history of changes stored on central server. Local copy of specific version at a given time stored on developer's ~~device~~ machine.

Pro: easier to manage

Cons: - requires constant internet access.
- has a single point of failure (central server)

2. Decentralized Version Control System:
history of changes is stored on both a central server and each developer's machine.

Pro: changes can be made locally and reverted without making them visible to others.

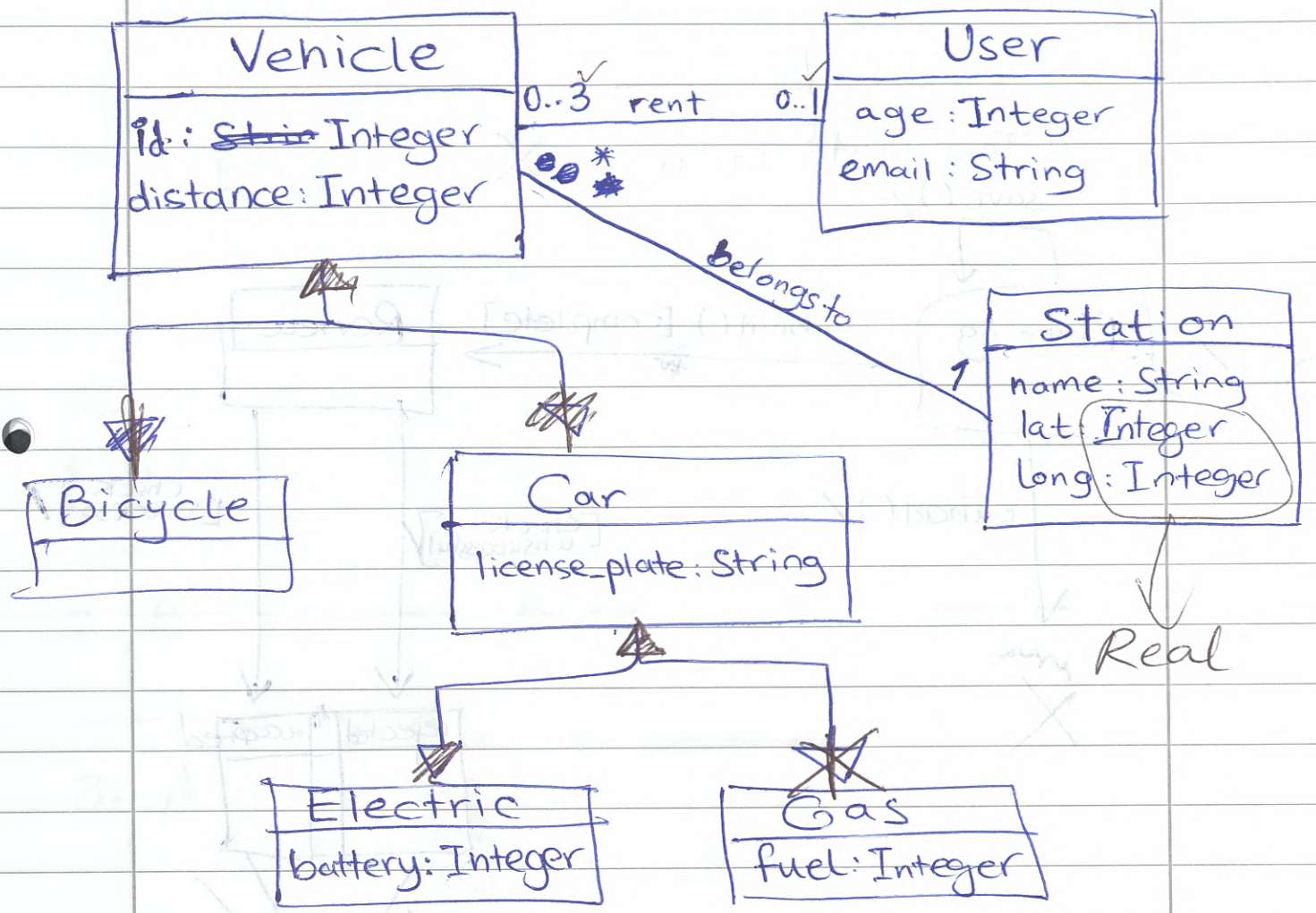
Con: more complex for beginners

~~Aufgabe 3,~~

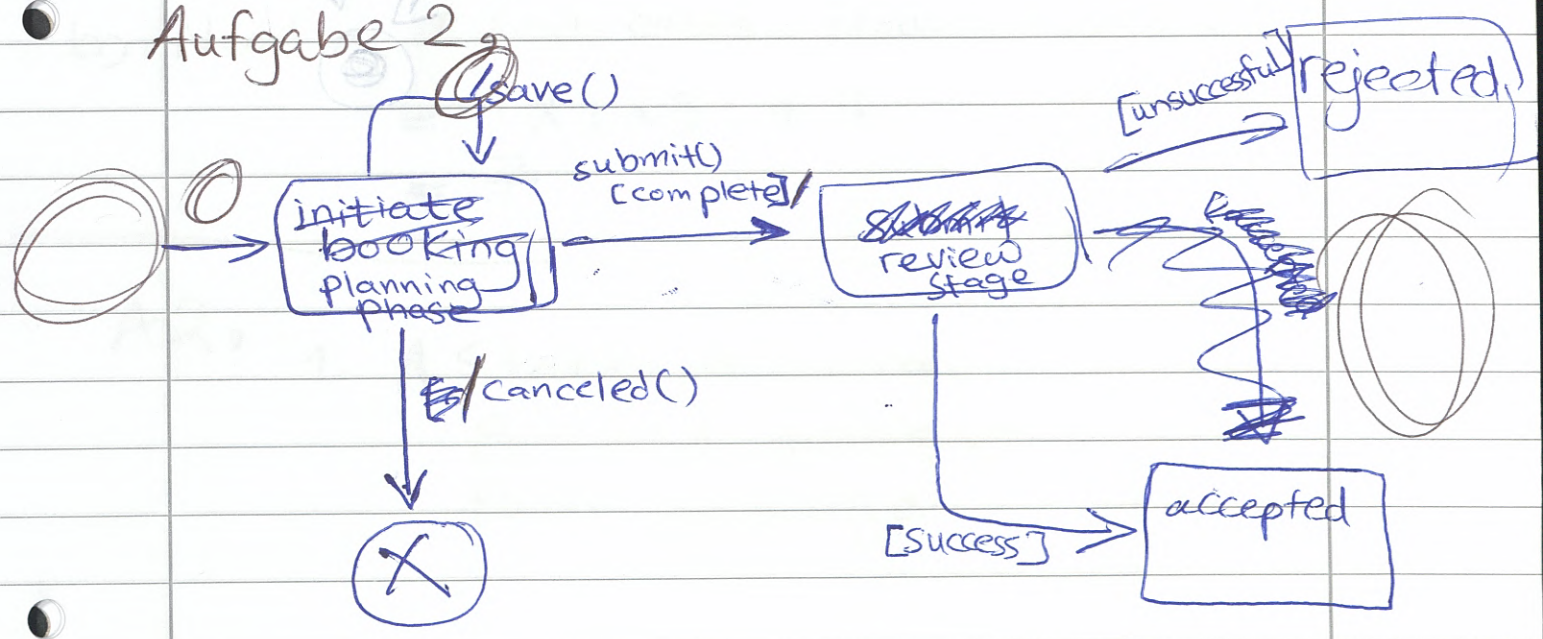
Revision

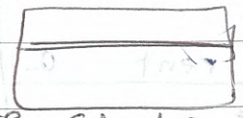
Übungsblatt 6

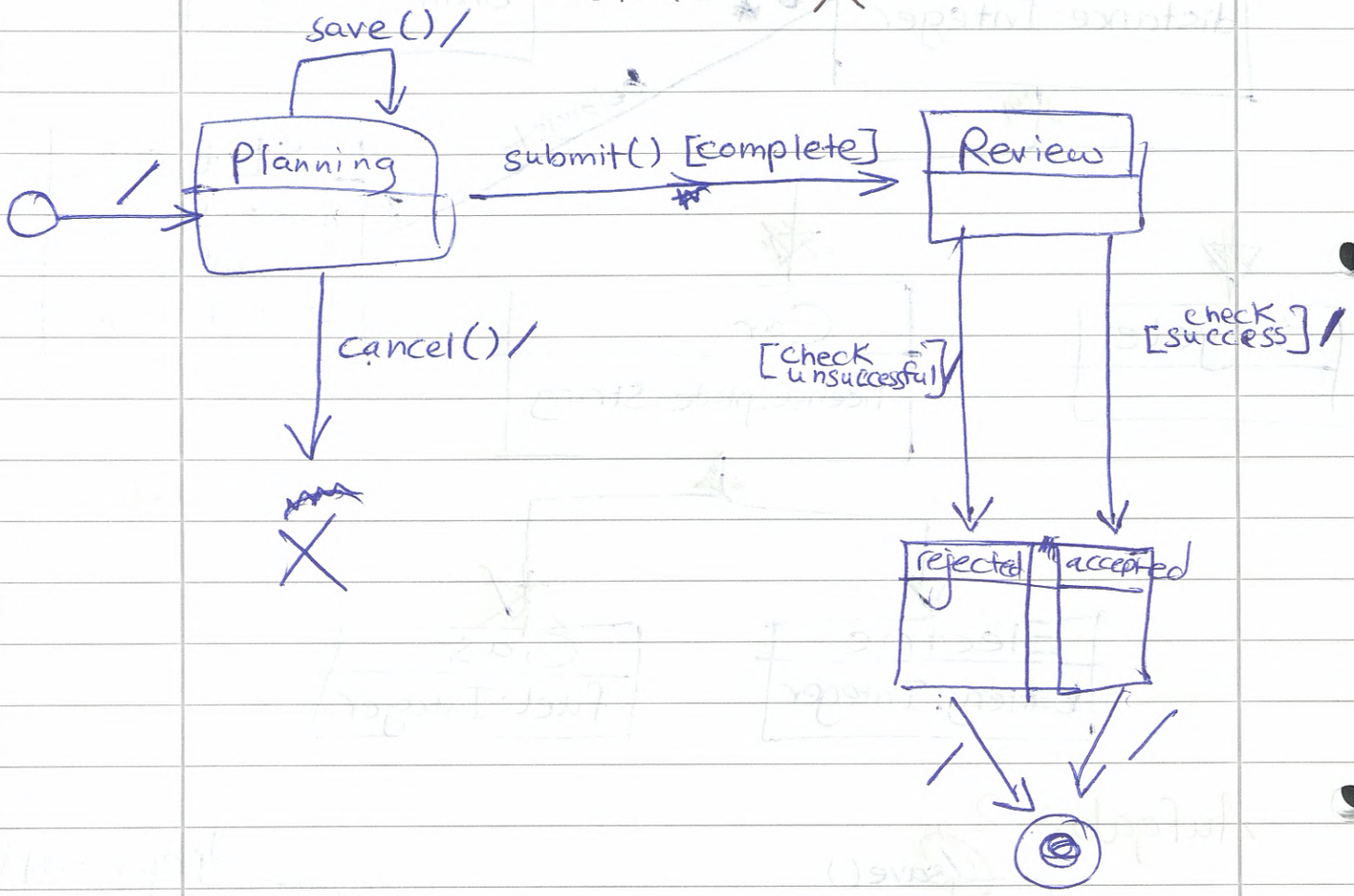
Aufgabe 1,



Aufgabe 2,



1. Start / End states
2. Each transition should trigger an action "/"
3. Boxes 
4. Terminate state : X



Übungsblatt 12

Aufgabe 1,

~~Eingabe~~

Alter $\geq 18 \wedge$ Alter ≤ 45

$B \geq 2$

$A = \{ \overset{C1=}{\text{Bachelor}}, \overset{C2=}{\text{Master}}, \overset{C3=}{\text{Diplom}} \}$

Eingabe	Gültige Klassen	Ungültige
1. Alter	1.1 $18 \leq$ Alter ≤ 45	1.a Alter < 18 2.b Alter > 45
2. B	2.1 $B \geq 2$	2.a $B < 2$
3. A	3.1 A = ~~Bachelor~~ C1 ~~A~~ 3.2 A = C2 3.3 A = C3	3.a not in $\{C1, C2, C3\}$

b) Number of test cases = product of valid + invalid

$= 1 \times 1 \times 3 + 4$

$= 7$

A2,

1. ~~A~~ Statement coverage
   Branch coverage
   Path coverage