Software Engineering Zusammenfassung

Software Engineering

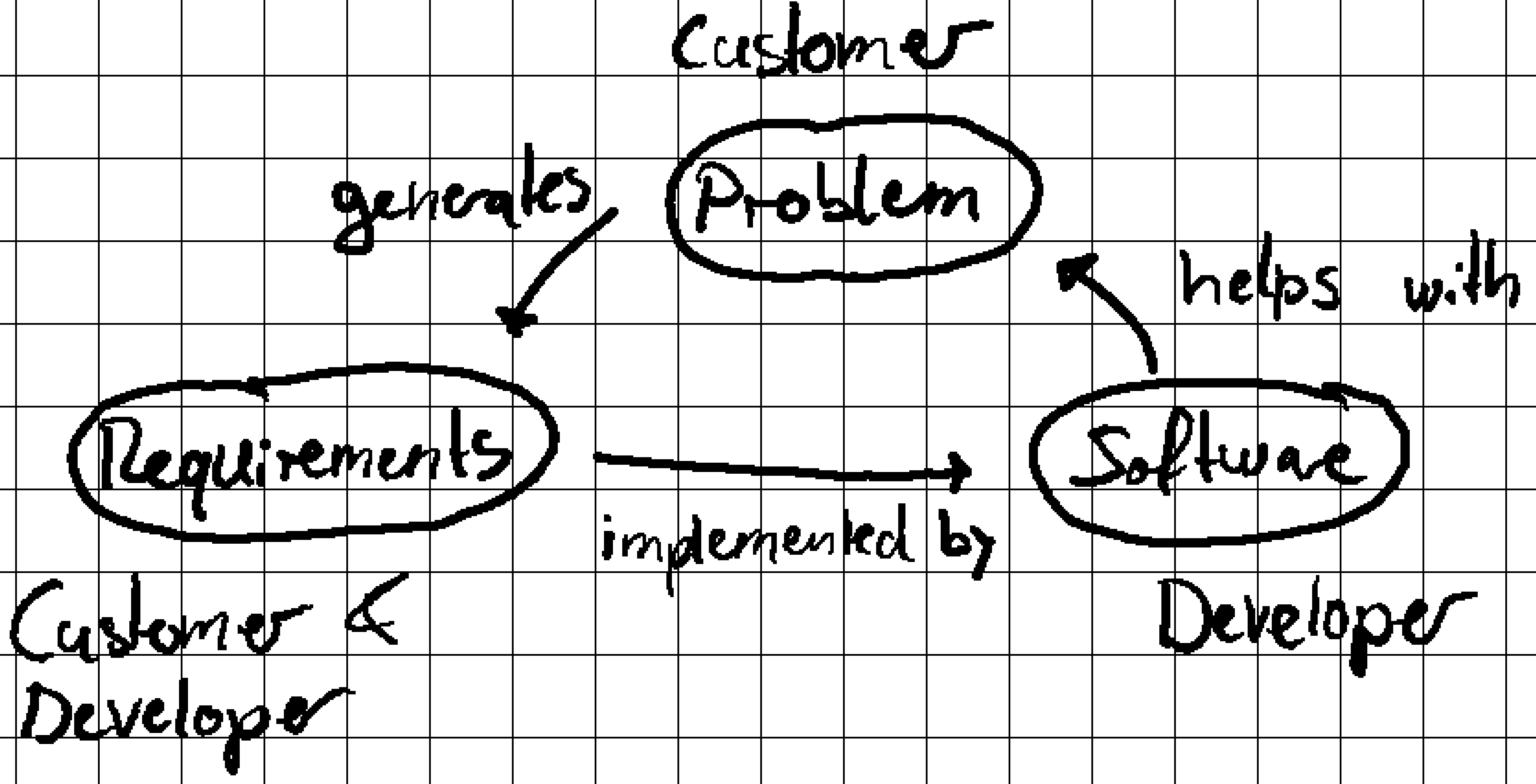
↳ Software engineering is about the systematic application of tools, methods and processes to develop high-quality software.

Software Products

↳ Software Systems, sold to governments, businesses and consumers

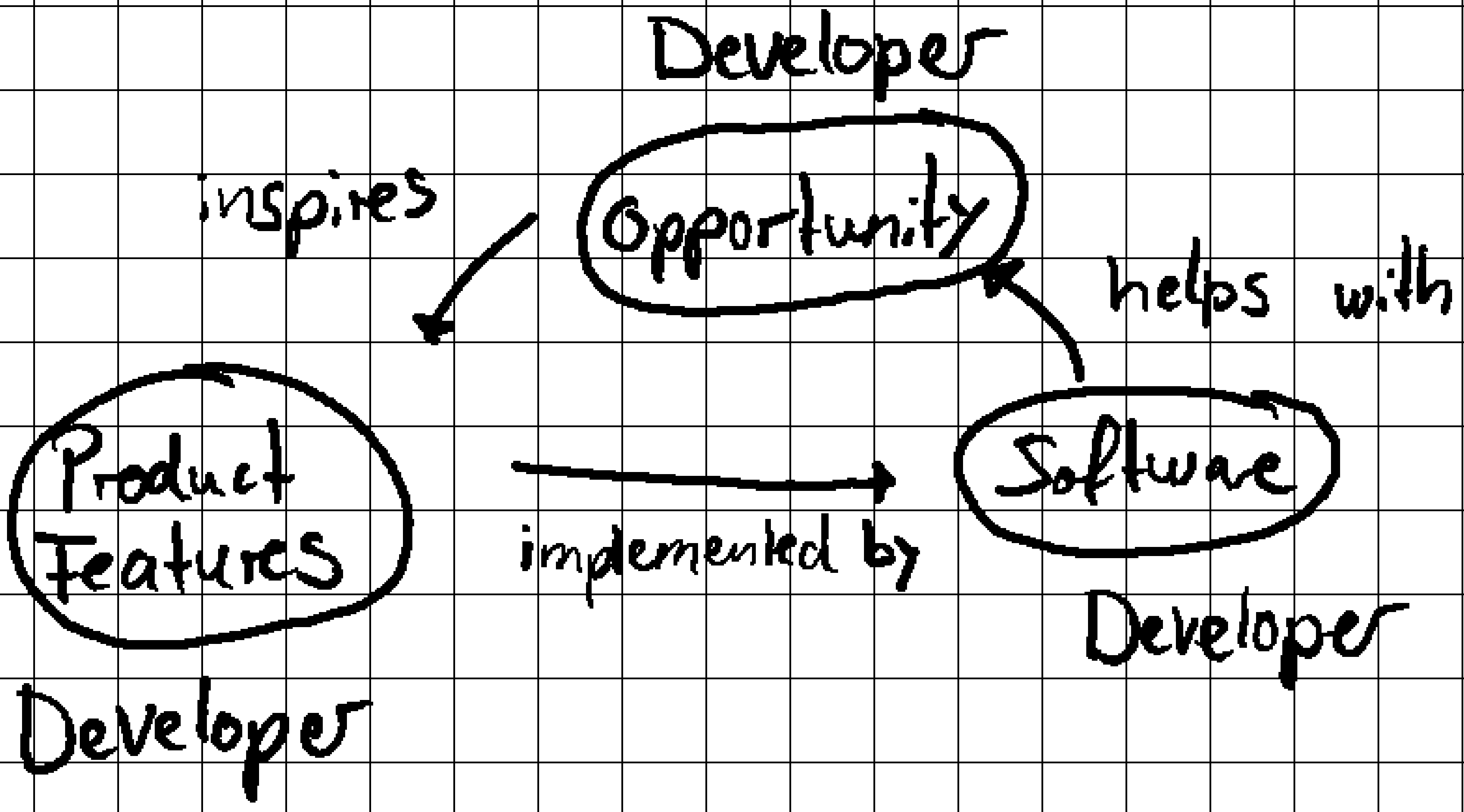
Project-based Software Engineering

↳ dominated for more than 25 years



⇒ starts with requirements is than developed with those requirements, but they could change throughout the prozess

Product - based Software Engineering



⇒ starts with a business opportunity and is implemented according to the companys own timeline, with a focus on rapid delivery

Software Product Line

↳ set of software Products that share a core

Platform

↳ Software or Soft- & Hardware product, that includes functionality to enable building new apps on it

Software Execution Models

- Stand-alone Execution
  ↳ all Features are on the client Pc as part of the Product, updates are provided by a server

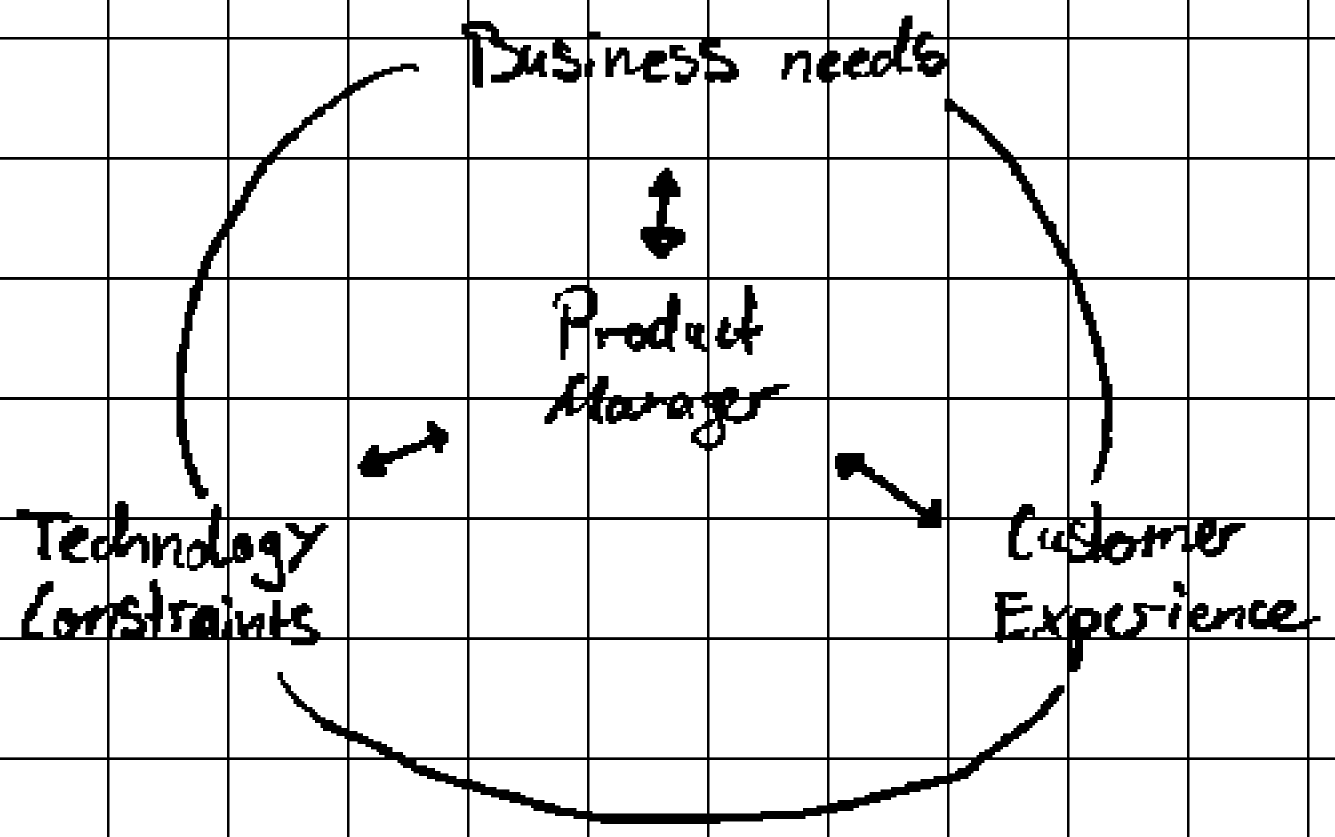
- Hybrid Execution
  ↳ some Features are on the client Pc as part of the product, others are provided by a server as a service (subscription)

- Software as a Service
  ↳ client Pc only has Interface, while all the functionallities are provided by a server as a service (subscription)

Product Vision

→ what is the Product ?
→ who is it for ?
→ why should a customer buy it ?
→ Diffrences to similar Products, what makes it better ?

Software Product Management



→ Product Manager → ensues business goals are met
↳ ensures dev team focus on customer needs
↳ communicates with customer and their good/bad ideas

⇒ manages vision, roadmap, backlog, user interface, user scenarios, testing ...

Product Prototyping

↳ development of early vesion to test ideas and convince you and investors of a real market potential

⇒ two step process
↳ Feasibility demonstration → show that the idea works
↳ Customer demonstration → user studies for concrete ideas & feedback

Agile Software Engineering

↳ focuses on fast functionality delivery, responding to changing specs and minimizing development overhead

Plan-driven Development

↳ based on controlled, rigorous software development processes (ex. used for aircraft control systems)

⇒ Software development is like building constructions
→ plan first, then build/implement

Agile Manifesto

Individuals and Interactions over processes and tools

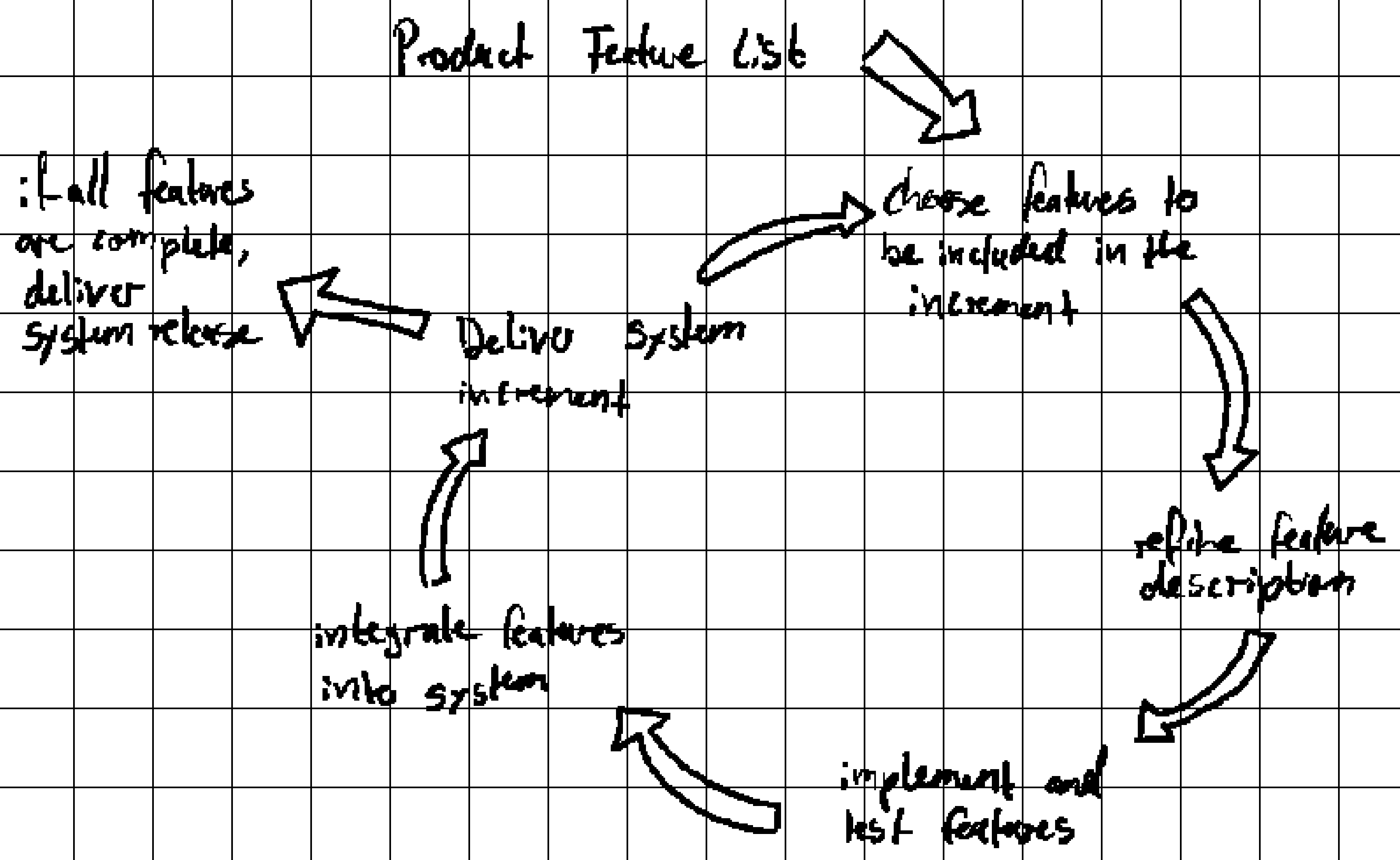
working software over comprehensive documentation

Customer collaboration over contract negotiation

$\underbrace{\text{Responding to change}}_{\text{more valuable}}$ over following a plan

Incremental Development

↳ implement feature after feature, prioritizing by importance


Product Featue List
choose features to be included in the increment
refine feature description
implement and test features
integrate features into system
Deliver system increment
if all features are complete, deliver system release

Agile Development Priciples

- Involve the customer
- embrace change → expect changes and adapt
- develop and deliver incrementally
- maintain simplicity → eliminate complexity
- Focus on people, not the development process

Extreme Programming (XP)

↳ techniques for rapid, incremental software development, change and delivery

- Test-driven Development → write Test first, then implement
- Refactoring → constantly improve structure, readability, efficiency, security
- Small Changes → frequent, minimal releases
- Continuous Integration → when a task is completed, it is integrated into System
- Incremental planing → no grand plan, each requirement at a time
- Collective ownership → every member is responsible for entire codebase
- Pair programming → two programmers, writer & reviewer
- Sustainable pace → constant workload, maintain work-life balance
- On-site customer → real customer present for continuous feedback
- Simple design → avoid over-engineering

Team Contract

↳ Agreement in a team, focusing on Meeting guidelines, roles & responsibilities, workflows, decision making processes, conflict resolution, to fasilitate a better working environment with a clear structure.

SCRUM

↳ method that provides a framework for agile project organization & plening

Terminology

- Product → software product developed by scrum team
- Product Owner → team member responsible for featues & attributes
- Product Backlog → to do list
- Development team → small, self-organizing team
- Sprint → short period, where increment is developed
- Scrum → daily team meeting
- Scrum master → team coach
- potentially shippable product increment → Sprint output
- velocity → estimate of work that can be done in one sprint

⇛ Scrum provides a method to break a product down into chunks and provides a continuous progress with good team communication and transparent progress for the customer

Kanban

↳ visuall representation of workload to manage tasks and monitor progress.

⇒ Think of Post-It notes with tasks, placed in columns that represent their status (in progress, done, ...)

Domain Storytelling

↳ modelling technique to understand and document processes and interactions within a specific domain

→ they use a pictographic language to show activities and interactions

→ they can vary in granularity

↳ level of detail

(Domain = targeted subject area of a computer program)

Domain Story → Scenario about people and their activities

⇒ can be used in Domain-Driven Design to structure the development process

Strategic Domain-Driven Design

↳ focuses on defining the overall architecture

Terminology

- Ubiquitous Language → shared vocabalary to consistently describe the domain and its processes
- Bounded Context → explicit boundaris for a domain model
- Context Mapping → technique to model relationships and interactions between different bounded contexts

Subdomains

↳ different parts of a domain that can be grouped together

Personas

↳ a way to represent different users with different needs

⇒ They inspire scenarios that turn into stories, that turn into features

⇒ a Persona is a type of user. → helps a developer better understand the actual execution of their feature through different users eyes

Scenarios

↳ description of something a user wants to do and why

⇒ helps to understand the actual use of the software

User stories

↳ brings together Persona & Scenario

⇒ As a <role> I want to do something

↳ As a student I want to be able to buy a ticket for an upcoming university event

⇒ User stories can be grouped and from them, new features can be derived

Feature Design

requires

- user knowledge → what do users want
- Product knowledge → research similar products
- Domain knowledge → how & where is it implemented
- Technology knowledge → what is even possible

A Feature should be

- simple but highly functional
- familiar but new and innovative
- automated but the user should feel a sense of control

Feature Creep

→ too many features

→ too complicated features

⇒ keep a good balance, only implement generally useful features while trying to reuse code that was already written

⇒ Features can be derived from product vision scenarios, user stories and domain Stories

⇒ A Feature should contain
- Description
- Constraints
- Comment

Requirements

↳ they can be derived from all the scenarios, stories by breaking them down.

Domain-Driven Design (DDD)

Terminology

- Entities → object with a distinct identity that has some sort of Lifespan and can have attributes
  ex: Customer with id, address etc.
- Value objects → Object that is defined by their attributes
  They are immutable and have no identity
  ex: Address with attributes like street etc...
- Aggregates → cluster of domain objects that are treated as a single unit.
  The individual objects can only be accessed through the Aggregate root.
  ex: an Order Aggregate has order lines with product, price and quantity, it also has an address. To change the address of an order, you have to go through the root, The order.
- Domain Events → Events that can trigger an action
  Ex. "order placed" triggers update inventory actions.

Domain-Driven vs. CRUD-Driven

- **Domain-Driven** → If someone wants to make a change to something that has an effect on multiple systems, he only has to change it at a predefined location and the changes cascade to the other systems

  ex: Contract termination → disable contract, systems like payment, access etc. get notified and apply changes

- **CRUD-Driven** → If someone wants to make the same kind of change in a CRUD environment, he has to change it in every subsystem manually.

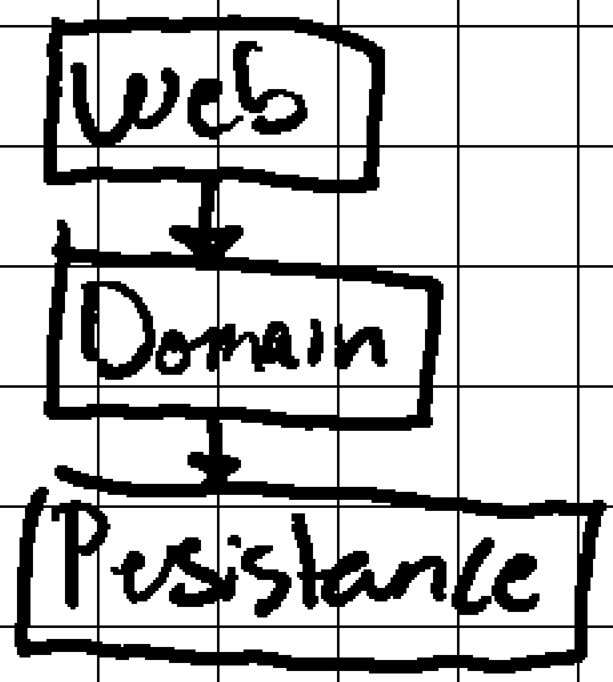
From Domain Story to Class Diagram

1. Identify concept & Entities
   ↳ main nouns and Subjects of Domain Story
2. Determin Relationships & Interactions
   ↳ Interaction between Entities → associations, dependencies, hicarchies
3. Define Attributes & Methods
   ↳ identify behaviours of each Entity
4. Categorize Entities & Value Objects
   ↳ based on Attributes, is it mutable?, has unique identifier?...
5. Establish Aggregates
   ↳ group related objects that can be treated as one
6. Incorporate Domain Events
   ↳ identify events that call for action in other parts of the system

Layered Architecture

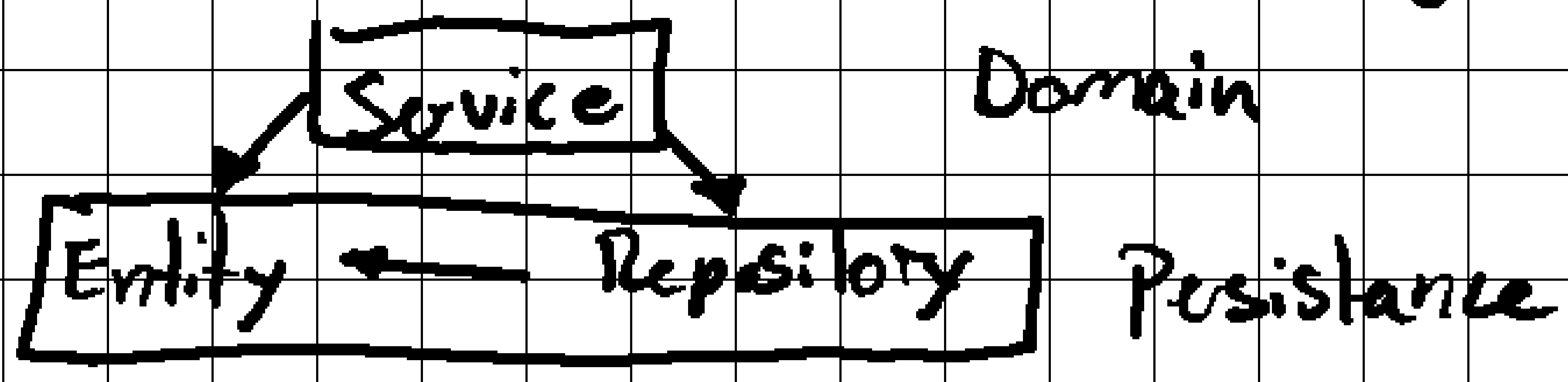
↳ Logical grouping of components

↳ changes to one level can only effect higher levels

ex: 

Drawbacks

- Vulnerable to changes
- few guard rails to keep architecture on track
- relies on human discipline
- Layers promote database-driven design

↳ 

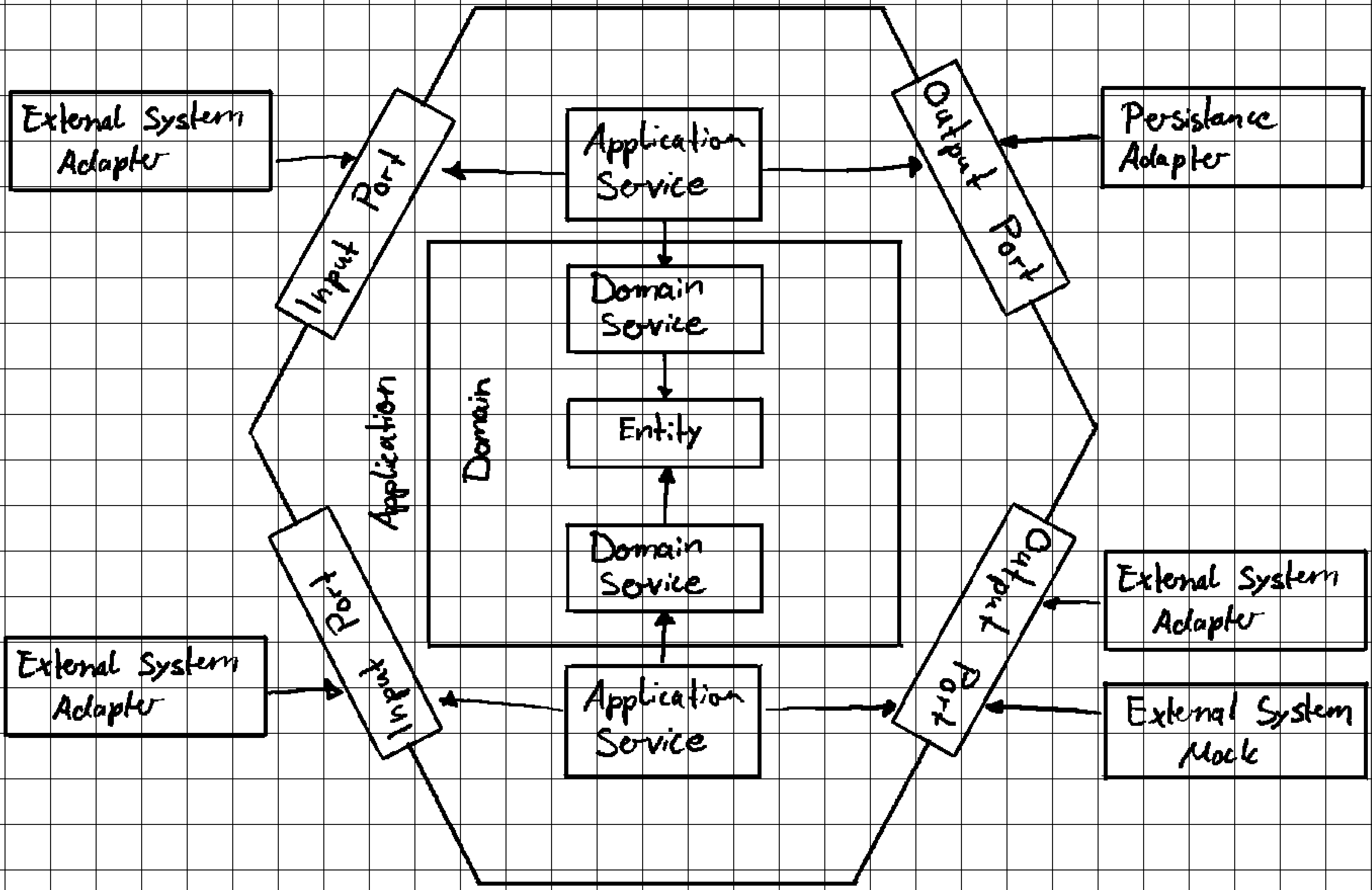
⇒ Strong coupling between domain & Persistence

↳ Persistance layer gets continuously bigger

⇒ evolution would be skippable layers

- prone to shortcuts
- hide use cases
- grow hard to test
- difficult to work on in parallel

Hexagonal Architectwe



Ports

↳ Interfaces that define how external agents can interact with the application

Adaptes

↳ adapt the external request into a form that the application can understand (inward communication)
and adapt the application responses into a form that external agents can understand (outward communication)

Application Service

→ orchestrates how the outside world interacts with the application

→ contains domain specific business logic

→ used to fetch Entities from db / outside world through ports

→ tells entities to do some logic

→ declares Infrastructure Services through ports

→ Executes other out-of process communication through ports
  ↳ ex. sending emails

⇒ Inputs & Outputs
  ↳ decide what data it uses and what it returns
    ↳ use entities directly from the domain model
    ↳ use Data Transfer Object (DTO)
    ↳ use Domain Payload Object (DPO) which are combinations of DTO and direct access

Invariants

↳ A condition that stays true throughout the life of the system

ex. the total price of a shopping cart must always be the sum of the prices of all products in the cart.

Difference of Layerd vs Hexagonal architecture

↳ **Layerd** is about organizing code by its role in the application (UI, business logic, data access, etc..)

↳ **Hexagonal** focuses on isolating the core logic from external inputs and outputs

Context Maps

↳ describes the relationship and interactions between different bounded Contexts within a system

⇒ **Integration Pattens**

↳ define how data and commands flow between contexts

↳ **common Pattens:**

- **Event - Driven**
- **API Calls**
- **Shared Database**

Software Architecture

↳ fundamental organization of a software system and the principles guiding its design and evolution

Non functional System Quality Attributes:

- Responsiveness	→ System returns results in a reasonable time
- Reliability	→ features behave expectedly
- Availability	→ System can deliver service if requested
- Security	→ System is protected from attacks
- Usability	→ uses can access features they need
- Maintainability	→ possibility for easy and uncostly updates
- Resilience	→ can a system maintain functionality on failure

⇒ Architectual Design influences

- Non functional product characteristics
- Product lifetime
- Software reuse
- Number of uses
- Software compatability

Architectural Design Guidelines

- Seperation of Concern
- Implement once
- Stable interfaces

Technical Partitioning vs. Domain Partitioning

Technical Partitioning

↳ organizing system based on technical concerns.
It seperates the system into seperate layers based on their roles and responsibilities

Domain Partitioning

↳ organizing system based on business domain and its core concepts.
It seperates the system into different bounded contexts each representing a speific part of the business domain

⇒ modular monolith

Code Management

Software Development Lifecycle (SDLC)

Analyse Design Code Build Test Deploy Operate

$\longrightarrow$

Software Configuration Management (SCM)

↳ Code Management / Version Control

Version relationships

→ Revision of x → version intended to supersede x

→ Variant of x → version intended to coexist with x

Functions related to Code Management Tooling

- Structure and Components
  - ↳ identifying component versions and reason behind versioning
  - ↳ identifying baseline for releases and their extensions
  - ↳ identifying product structure with component relationships
  - ↳ selection of compatible component for consistency
- Construction
  - ↳ Building application from code while creating snapshots, asses the changes made and roleback if needed
- Auditing
  - ↳ keep a history of all changes, trace related components and their evolution and keep a log of all the work that has been done
- Accounting
  - ↳ generate reports about the current status of the project with statistics related to the development process
- Controlling
  - ↳ divide system into mangable parts, controll access, track bugs, manage changes and their deployment to the whole system

Git

↳ distributed, open source version control system (VCS)

- Centralized repository with different branches
  - ↳ local repositories can sync withe the central repo

Some Commands

git stash → shelve changes that are not ready to commit and restore them later

git revert → undo some commits

Github

↳ web based git Platform that provides tools collaboration and menegement of projects

Concepts:

- Repository → Storage space for your projects code
- Branch → vesions of main code with differences like new features
- Clone → copy a repository to your local machine
- Commit → Save code changes to your local machine (snapshot)
- Push → Upload local changes to remote repository
- Pull → Fetch changes from remote repository
- Merge → combine changes from one branch into another
- Rebase → Move changes from one branch to another to simulate Sequential changes
- Pull Request → Propose change to codebase and allow others to review
- Fork → create personal copy of someone elses repository
- Issues → create notes for found bugs or other issues

Additional Features

- CI/CD
- kanban

Software Testing

↳ Execution of Software, using simulated Inputs to observe if the software meets its specifications

→ dont put logic in tests

↳ Tests should be DAMP < DRY

DAMP: Descriptive & meaningful phrases

DRY: Do not repeat yourself

⇒ A Test should not change unless:

- The code has been refactored
- Found Bug indicate missing test

Types of Testing

- Functional Testing → test functionality with goal of finding bugs
- User/Usability Testing → test if the product is useful and usable by end-user
- Performance/Load Testing → test if the software works quickely and is able to handle the expected load
- Security / Penetration Testing

  ↳ tests that a software maintains integrity and protects usr info
- Fault injection / disaster recovery test

  ↳ test that a software is robust and degrades gracefully

Stages of testing

functions, methods, classes...

- Unit testing → test program units in isolation
- Featue testing → ensue diffeent unit work together as expected
- System testing → test that there are no unexpected interactions between the features
- Release testing → test that the whole system operates as expected

Automated vs. Manual Tests

↳ Automated Tests can be useful to ensure that a software behaves the same after a code change, as it did before

↳ Manual Tests sometimes require human judgement
ex: assessing the user experience

Common Unit Test Types

- Test egde cases → upper & lower bounds etc..
- Force errors → inputs that should generate invalid outputs
- Repeat yourself → test whether a unit correctly deals when the same Input is presented multiple times
- Floats → Be aware that floating point operations are not exact
- Overflow & Underflow → if numeric callculations are done, check with inputs that result in very large/small results
- Don't forget Null and Zero → test Null-pointer and 0
- keep count → check for list length consistency
- One is different → check sequence of only 1 element

Test Doubles

↳ an object or function that stands in for a real implementation in a test.

(ex: database, current time, API, ...)

⇒ reduces complexity

→ In Java we can for example use the Mockito Library

Test Terminology

- Test doubles → object or function stand in for a real implementation
- Mock → object complying with an interface but simulating only some behaviours of the corresponding interface contract
- Fake → working implementation of a dependency but is simplified or has some shortcuts
  ↳ not suitable for production
- Stub → stand in with predefined responses to method calls
- Spy → object that logs each method call that is performed on it including parameter values

Integration Testing

↳ level of software testing, where individual units or components are combined and tested as a group

⇒ Focuses on interfaces and interactions
↳ In Hexagonal Architectures this means testing External System Adaptors (ex: Persistance, web Adapter etc...)

Test Driven Development (TDD)

↳ write Test first, then code

1. Identify partial implementation
2. write mini-unit tests
3. write a code stub that will fail test
4. run all existing automated tests
5. implement code that should cause failing test to pass
6. rerun all automated tests
7. refactor code if necessary

Benefits

→ ensures that tests cover all of the code

→ Simpler debugging, as you only have to check new added code

Exploratory Testing

↳ Manual testing without a script

⇛ try to find bugs by just "free styling" through the program, by testing different scenarios, by trying stuff that you know you haven't written a test for, or by hunting specific bugs

UX Testing

↳ test user experience of the software

assessing usability, utility and interface impressions by observing users doing typical tasks

⇒ Evaluate results

Security Testing

↳ aim to find vulnrabilities that may be exploited by an attacker

Penetration testing

↳ using different methods like phishing, network exploits etc. to try to get access to priviledged data

Risk-based security testing

↳ identifying common risks and developing tests to demonstrate that the System protects itself from these risks

Canary testing

↳ Deploy to a small subset of users to ensure no issues

A/B Testing

↳ Deploy two different versions to assess some metric

Code Review

↳ one or more people examening code to check for errors and anomalis

Static Analysis

↳ automated code analysis without executing code

- Type checking
- Data-flow analysis → computes which Values are available
- Control-flow analysis → computes possible paths through the program to find dead code
  ↳ find race conditions and deadlocks
- rule based analysis → ex: variable declared but not used
- ML-based Systems → find vulnrabilities

Resilience

Terminology

- Failure → Deviation from specification
- Error → Part of system state that can lead to failure
- Fault → cause of an error

Addressing Faults

- avoid faults
- mask faults
- manage failure → recover from failure
  - ↳ fail safe
  - ↳ compensate failure

Failure Avoidance

↳ reduce complexity
- ↳ reading complexity
- ↳ structural complexity
- ↳ Data complexity
- ↳ Decision complexity

⇒ refactoring to reduce complexity

Input Validation

↳ checking user input for a correct format

Methods for Input Validation

- Built in functionallity → provided by framework
- Type coercion → convert input to valid input if possible
- Explicit comparisons → Define List of valid inputs and try to match
- Regular Expressions → Match input against format scheme

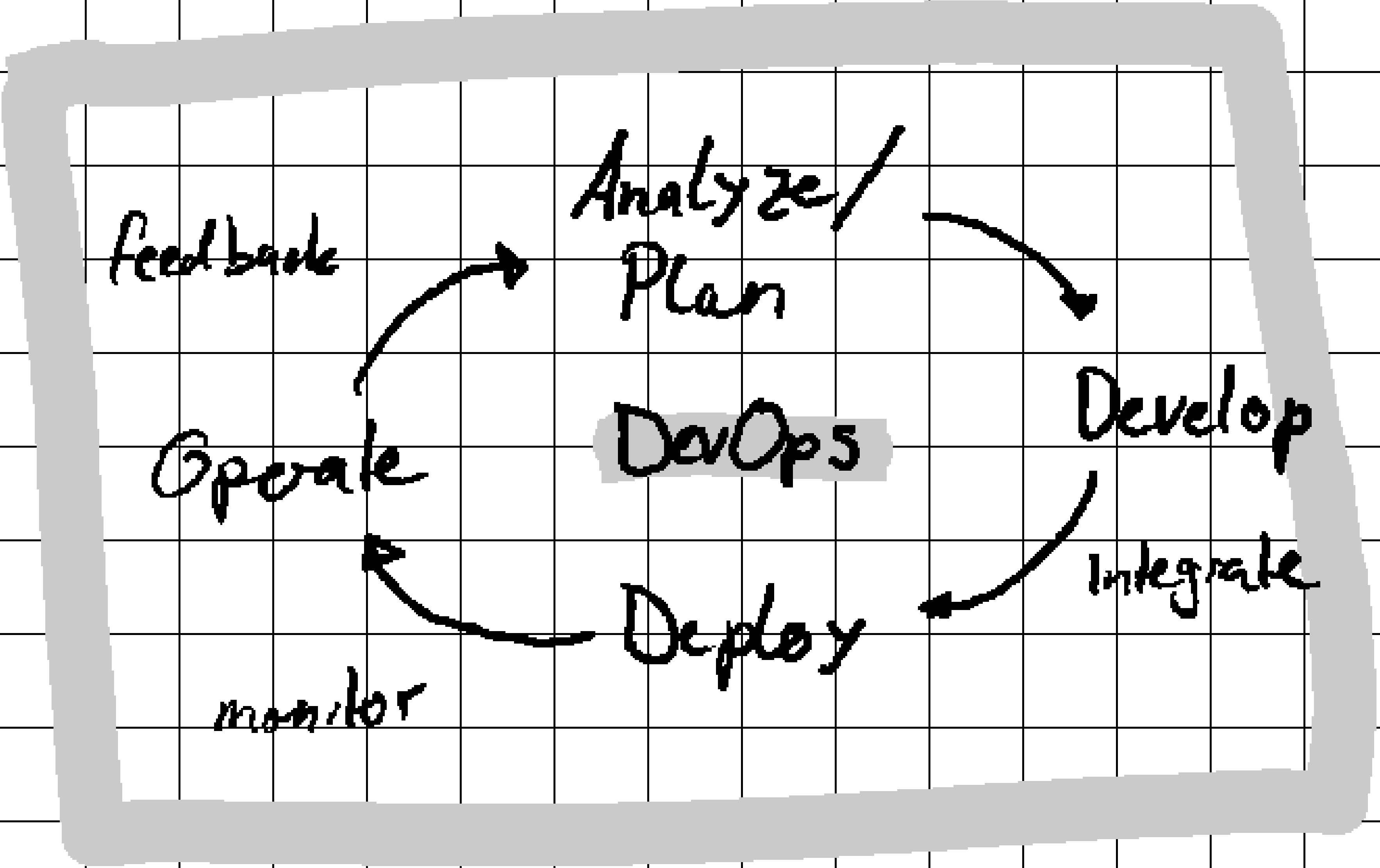
Regular Expressions

^	→ start string
$	→ end string
+	→ indicate next element
$\alpha\beta$	→ α then β
$(\alpha \mid \beta)$	→ α or β
α^*	→ α 0 - ∞ times
$\alpha+$	→ α 1 - ∞ times
$\alpha?$	→ α 0 or 1 time
$\alpha\{3\}$	→ α 3 times
$\alpha\{2,5\}$	→ α 2 - 5 times

.	→ any character exept \n
[a-q]	→ any character between a and q
[0-9]	→ any digit between 0 and 9
[02-7]	→ 0, or 2-7
[^a-w]	→ any character not a-w
\\.	→ the character "."
\d	→ one digit
\D	→ one character not digit
\w	→ one word
\W	→ no word character (ex. punctuation)
\s	→ one white space

DevOps

↳ Approach to improve the speed and quality of software delivery by combining the dev and the ops team into one that handles both.



→ collect feedback
  ↳ react accordingly

CI/CD

↳ Continuous Integration
  ↳ for every change to the main branch, a new, executable version of the system is built and tested

↳ Continuous Delivery
  ↳ Capability to deploy a system into a testing setup at any given time

↳ Continuous Deployment
  ↳ for every change to the main branch, a new release of the system is made available to the users

Dev Ops Metrics

- Process Measurement
  ↳ collect & analyze data about deployment, testing and deployment process

- Service Measurement
  ↳ collect & analyze data about software performance, reliability and acceptability to customers

- Usage Measurement
  collect & analyze data about how customers use the product

- Business success Measurement
  ↳ collect & Analyze data about how the product contributes to the overall success of the business

Cloud

↳ a pool of computing resources (compute, storage, etc.) that is accessed remotely (Techical Definition)

↳ A set of business models to rent computing resources that you and your customer access remotly (Business Definition)

Terminology

- Public cloud → public rental service
- private Cloud → personal resource pool (self owned or rented)
- Hybrid Cloud → composition of public & private cloud
- Multi Cloud → utilization of clouds from multiple provides

Hardwae Virtualization (VMs)

Host Hardwae → Host OS → Hypervisor → Guest OS

↳ emulates hardwae resources to the guest OS

⇒ Virtual Machines provide ways to manage server loads and resilience as they can be cloned/moved for uptime guarantees

Containers

↳ OS - level virtualization

$$\begin{tikzcd}
\text{Host Infrastructure} \arrow[r] & \text{Host OS kernel} \arrow[r] & \text{Container Manager} \arrow[d] \arrow[dr] & \\
& & \text{OS Name Space 1} \arrow[d] & \text{OS Name space 2} \arrow[d] \\
& & \text{Container Software} & \text{Container Software}
\end{tikzcd}$$

⇒ Ex: Docker:

$$\begin{tikzcd}
\text{Dockerfile} \arrow[r, "\text{builds Image}"] & \text{Container Image} \arrow[r, "\text{runs on}"] & \text{Container}
\end{tikzcd}$$

managed by Docker Daemon

⇒ Images can be stored on a centralized Hub like DockerHub to push & pull updates

Docker compose

↳ multiple, interacting containers (like app & DB)

Distributed Architectures

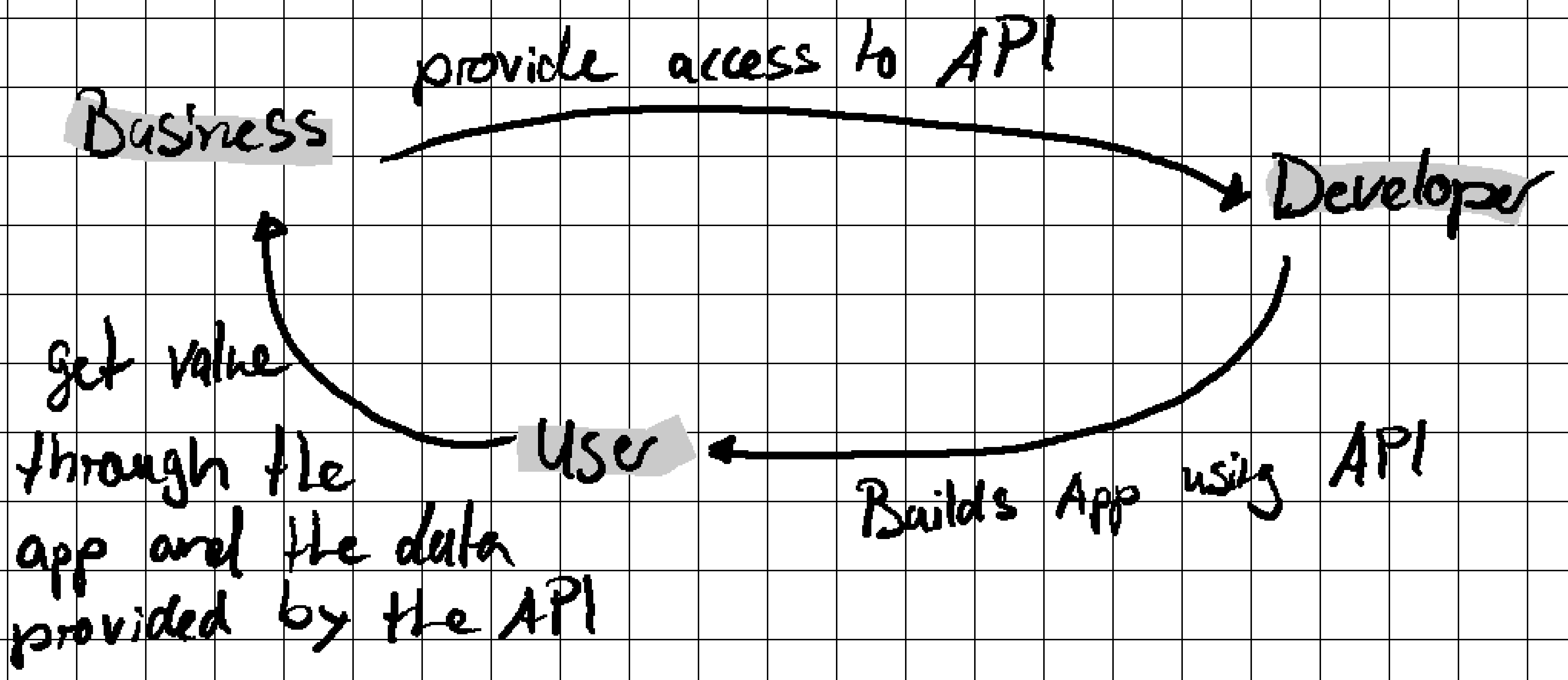
Architectural Quantum

↳ independently deployable component with high functional cohesion, which includes all the structural elements required for the system to function proporly.

⇒ multiple Quanta can be used in a Service based architecture

API

↳ Application Programming Inteface



⇒ already trained machine learning models can be accesed via an API

⇒ enables Devs to create ML enabled apps

⇒ cloud managed model

ML- Enabled System

→ ML almost always used as a component of a lager system

Challenges:

- Data Quality → garbage in, garbage out
  - ↳ Accuracy → free from errors
  - ↳ Completeness → free from missing values
  - ↳ Consistency → free from contradictions
  - ↳ Currentness → degree of up to date data

- Drift → change in environment or date over time, that can lead to degegation of the generated output
  - ↳ Hardwae drift
  - ↳ Environmental drift
  - ↳ Human drift
  - ↳ virtual drift

- Reproducibility → Ability to reproduce an output given the same input

- Scale → build a system that is able handle lage datasets, has enough compute power to handle these datasets and handle requests of a growing number of users

- Multiple Objectives → design Algorithms that can simoultaneously optimize for multiple metrics
  ↳ manage the trade-offs between different objectives

MLOps

↳ set of practices at the intersection of Machine Learning, Data Engineering and DevOps with the goal to deploy and maintain ML-enabled systems in production efficiently and effectively

Artifacts of ML Projects

- Data
- Model
- Code

Further research in AI

→ how to debug deep learning code ?

→ how to test for fairness ?

→ how to find bugs ?

Interpretable vs Non-Interpretable Machine Learning

Interpretable → Models are designed in such a way that their decision and the process by which they make decisions can be easily understood and explained by a human

Non-Interpretable → also black-box models
↳ decision making process is unknown

Process Mining

↳ discover, monitor and improve real processes by extracting knowledge from event logs readily available in today's information system