SOFTWARE ENGINEERING

COURSE INFORMATION

SoPra is the application of concepts learned in Software Engineering.
Flipped classroom: each week before class watch videos on youtube and readings on OLAT. Each week before class there is a quiz (only 5 minutes of time for it). If you pass quizzes you get 5% of the grade already.
Midterm and Final Exam: on-site, closed book, BYOD with Safe Exam Browser.

No need to pass both (we can compensate but try to pass both!)

30% Midterm BYOD (in person) (05.4, 10am – 12pm)

65% Final BYOD (in person) (21.6, 10am – 12pm)

midterm & final most likely multiple choice (Kprim questions)

1 wrong -> half points, 2 or more wrong -> 0 points and not negative points.

5% Participation (weekly survey, at least for most weeks)

each quiz/survey has a few questions, one attempt to answer it

5 minutes time to answer

4 points per week, overall >=75% correct

questions similar, grading slightly different to midterm/final

---

INTRODUCTION

Software is eating the world, we are very dependent on it and use it everywhere.
Software is the barrier between CS and the outside world.

Programming/Software Engineering is the process of transforming a mental plan of desired actions for a computer into a representation that can be understood by the computer. Software Engineering involves people working together, to create a robust software system that satisfies the client. This involves technical and interpersonal challenges. While programming software is hard, and is an important and necessary part of the software engineering process, the tasks that come before (like figuring out what to build and how to do it) and after programming (like deploying, maintaining, and evolving the existent system) are often crucial for ensuring that the programming effort is effectively applied.

History of Software Engineering

Before digital computers: computers were people calculating things manually. Computing was only reserved for the few people that could afford and maintain a room-sized machine. Since computers were slow and programming required a long planning in machine code, most programs were not complex. Computer programs did not solve problems that had no solution yet, but were translating existing solutions into machine instructions -> the power of computing was not in creating new realities or facilitating tasks, but rather accelerating the old ones.

SOFTWARE PROCESS

Process is worth it, with process by the end of the project the team will operate at a high speed instead of spending all their time in meetings and correcting defects with little or no time extending the software.

4 Main Questions:

- What: what software is being built?
- Who: who are the people involved in the software development process?
- How: how do these teams coordinate with each other and build the software?
- When: when is the software considered finished?

Stakeholders:

- Technical: developers, QA (quality assurance), DevOps, managers
- Non-technical: sales, users, support

Each stakeholder needs a form of documentation to communicate effectively with each other.

4 types of documentation:

1) Specification: business objectives, requirement (involved: managers, users, sales -> marketing team)
2) Development: system architectural plans, design and implementation plans (involved: developers, qa, managers)
3) Validation: test plans, risk assessments (involved: qa, developers)
4) Deployment / Evolution: DevOps plans, made-in strategies

→ Not all of these documents are required and used when building a project

A software process is a structured set of activities to develop a software system. It defines who does what, when, and how, to reach a goal.

Processes have descriptions that discuss:

- Products: outcome of a process activity
- Stakeholders: people who care about the outcome (developers, qa, DevOps, customers, testers, ...)

There are many different software processes, but all include requirements elicitation, architectural design, detailed design and specification, implementation, integration, testing, maintenance.

Waterfall process

- Linear and sequential approach: process is followed in steps without possibility of going back, so it is made sure that one step is done before moving to the next ones
- Resistant to change in practice
- Allows for easy understanding about what is going on in development of the system at any point of time -> at each step it's clear what is involved in that step (high clarity between each of the phases in the model)

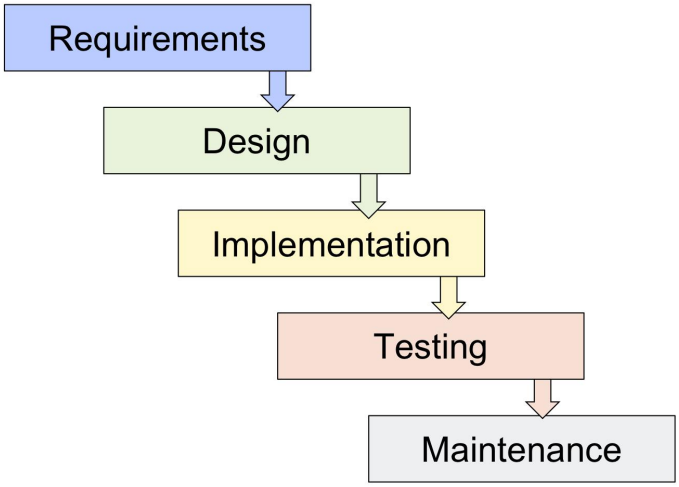
Advantages

- Good for well-understood but complex projects. It tackles all planning up front.
- Provides support for an inexperienced team -> easy to follow model, reviews at each stage

Disadvantages:

- Difficult to get requirements right up front
- No sense of incremental progress

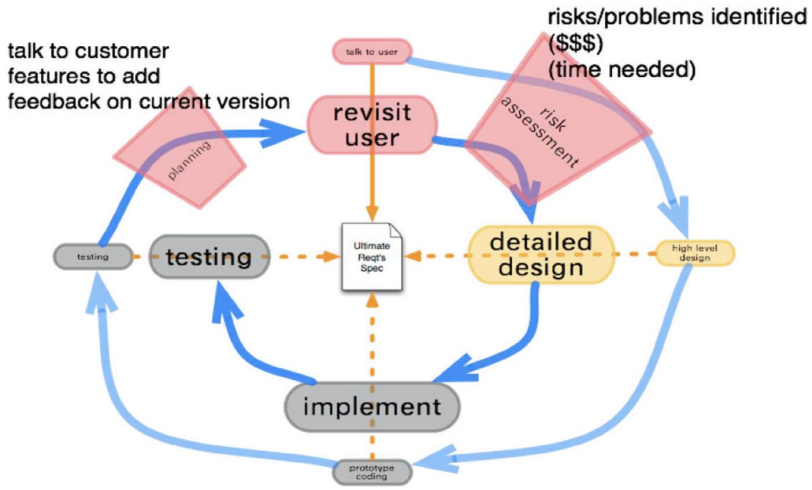
- No integration until the end
- Final result is not necessarily client-driven



Spiral Model

- Planning: describes what the requirements for the product are
- Risk analysis: describes what can go wrong and what sources of risk are present
- Engineering: describes the phase where the system is implemented
- Validation: involves checking that the built prototype matches the client's requirements
- For most iterations the entire spiral process takes about a year (once a year there is check with customer)

Disadvantages: documenting the risk analysis process can be overwhelming and waiting for customer feedback for a whole year at a time can slow down the software development process (feedback that a customer could give right away is delayed)



Agile Methodologies

- Follow the idea of tightening the feedback loop between the development team and the customers, by enabling developers to be more flexible and achieve a higher development speed.

Agile Manifesto

4 values:

- Focus on individuals and interactions over processes and tools -> value the stakeholders involved in the software development process
- Focus on working product over comprehensive documentation -> software should always be buildable and executable
- Focus on customer collaboration over contract negotiation -> allow the customer into the development team so that feedback from him can be received and quick response to his needs can be provided
- Focus on responding to change over following a plan (favor agility over planning): be flexible to the feedback that customers are

giving you and respond by building a product that reflects their needs and wishes more accurately

Agile does not mean no documentation and/or no process. Agile embraces change, requires close customer involvement and requires planning.

Agile Principles

12 Principles.

- Satisfy customer through early and continuous delivery (heavily iterative and incremental)
- Deliver working software frequently (short iterations). Delivering software is the primary measure of success, it focuses on a small number of prioritized requirements
- Daily collaboration between stakeholders
- Welcome changing requirements
- Build project around motivated individuals and provide supporting environment -> focus on individuals
- Teams are self-organizing
- High value on face-to-face conversations
- Team reflects regularly on how to improve and adjust
- Teams work at a rate that can be maintained for the entire project duration (sustainable pace) -> they are less likely to make mistakes, burn out, or leave the company
- Simplicity (art of maximizing the amount of work not done) is essential -> leave extension for later, focus on the requirements in the present

- Continuous attention to technical excellence and good design enhances agility
- Working software is the primary measure of progress

Traditional approach: work is organized around the team

Agile approach: team is organized around the work

Agile Methods

- Extreme Programming (XP): specific practices -> customer driven development, daily builds, small teams
- SCRUM: project management approach, rely on self-organizing teams

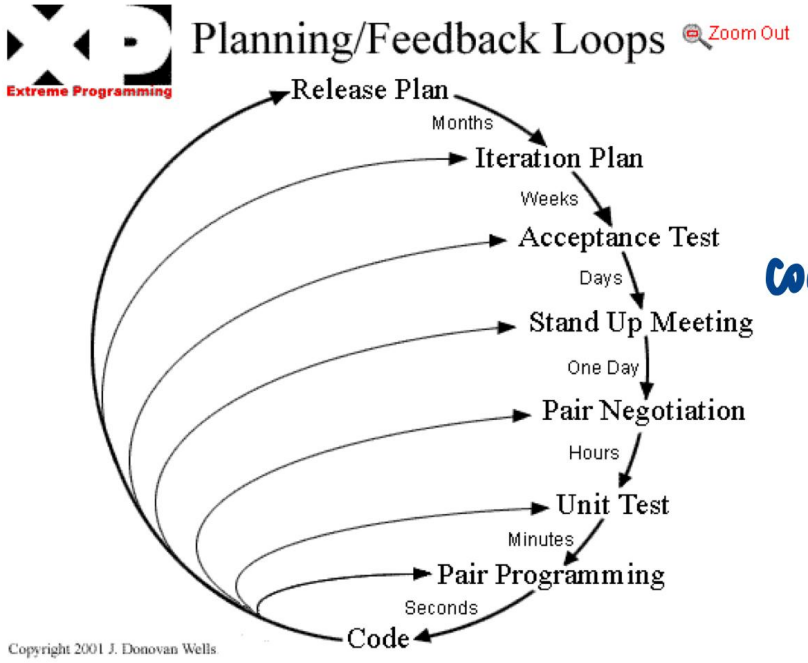
Extreme Programming (XP)

- Idea: have a buildable system at all times so that the development team can start small and slowly grow the system over time in response to the customer

Five principal values:

- Communication: common metaphors, frequent verbal communication, customer involvement -> encourage stakeholders to talk to each other at all times so that the developers can get feedback from the customers crew as quickly as possible
- Simplicity: do the simplest thing that could possibly work, then refactor -> better simple than over-engineered solution

- Feedback: from the code (unit tests), the customer, the team -> encourage customers and developers to talk to each other (from customer to developer but also from developer to customer)
- Courage: be willing to throw things away -> otherwise problems occur later in time (empower the development team to experiment with different solutions)
- Respect: don't do things that make work for others -> don't break the build when committing code, so you don't waste your colleagues time and energy



continuous iterations

XP - 12 core practices

Fine scale feedback: pair programming, planning game, test driven development, on-site customer / whole team

Shared understanding: coding standards (uppercase, camelcase underscore -> creates less conflicts), collective code ownership, simple design, system metaphor (overall vision of project)

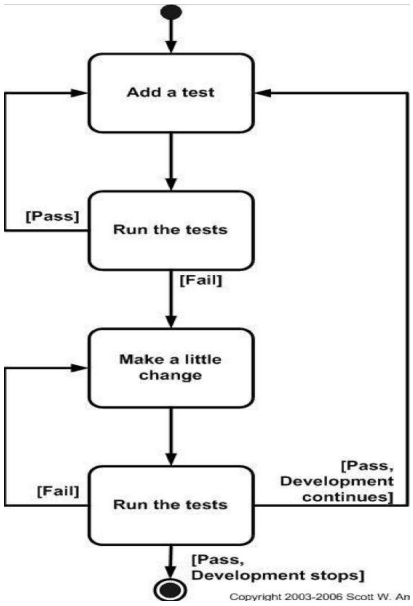
Continuous process: continuous integration (continuously refactor software and make it better), refactoring, small releases

Programmer welfare: sustainable pace

XP - Pair Programming: useful to know how other programmer works, but match up must be good otherwise the couple is not productive

Test-Driven Development (TDD) -> part of Agile

Test cases are written first, so that they cover new functionality or improvement. Then the necessary function is implemented -> code is complete when all tests pass. Refactor before adding feature if design could be better (no refactoring in between, less error-prone)



TDD - Advantages:

- Easier to test and validate written code
- Tests can be reused and re-run after changes -> regression testing

- Tests define units of work
- Tests document expected functionality of code
- Tests are contracts for the methods
- Failure indicates breakage
- Small steps

TDD - Disadvantages:

- Hard to create test cases before writing code (especially for GUIs)
- Can require extra time during implementation
- Hard to do for real systems
- Designing good tests is difficult

SCRUM

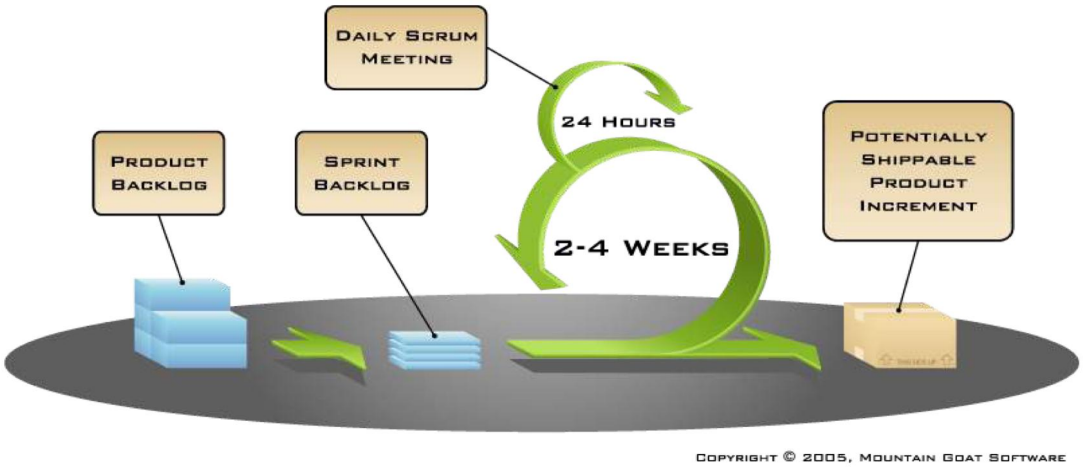
Most common form of agile methodology used in practice today.
It's a management framework for incremental product development. It is an iterative and incremental process to control risk and optimize for predictability. It requires transparency, inspection, and adaptation.
SCRUM based methodologies do not prescribe SE practices that a team needs to follow, but instead the team itself will figure out what makes the most sense for the product they are building.
Principles:

- Self-organizing, cross-functional teams
- Product progresses in a series of two to four-week iterations (fixed length) called sprints -> where ideas are turned into value. In sprints the development team works to complete the items that are

present in the sprint backlog. At the end of each sprint a potentially shippable product is delivered

- Every iteration produces a potentially shippable (properly tested but not complete) product
- Requirements are captured as items in the product backlog
- Teams self-organize to determine the best way to deliver highest priority features
- No specific engineering practices prescribed

Other than doing all of one thing at a time, scrum teams do a little of everything all the time.



Scrum Roles

Product Owner: defines features of the product, prioritizes features according to market value, and adjusts features and priorities every iteration as needed. He is outside of the team and communicates with it. He ensures that the product backlog is transparent and understandable

Scrum Master: highest priority. Facilitates the scrum process, helps resolve impediments, and shields the team from external interferences. He is not the manager. He is responsible for keeping the team on track during the sprint and making sure that the team is working specifically on the issues that the product owner has prioritised for that sprint. He can also write and contribute code (often one of the developers).

Team: self-organizing, self-managing, cross-functional. Composed of developers, designed, and testers. Usually from 5 to 9 people. The development team creates and works on the Sprint Backlog. Developers in the team hold each other accountable

Scrum Artifacts

Product Backlog: prioritized list of requirements or user stories that describes all desired functionality of the product. It helps stakeholders to understand what features and requirements are needed for the product.

Product Backlog Item: specifies a customer-centric feature (User Story form) -> effort estimated by team, business value and priority estimated by Product Owner

Sprint Backlog: contains a list of sprint items (user stories) that are negotiated by the team and the product owner from the product backlog for the sprint broken down into specific tasks. Development tasks should be small (estimated effort < 1 day)

Burndown Chart: total remaining team task hours within one sprint. The progress rate is measured by the amount of story points completed during each iteration.

Scrum Events

Sprint Planning: collaboratively lay out work to be performed

Daily Scrum: short progress inspection towards sprint goal. It is not a problem solving session nor a way to collect information about who is behind the schedule, but instead it has other goals: developers make commitments to each other, scrum master tracks progress, and adaptation if necessary (e.g., task estimates). It should only take approximately 15 minutes. Everyone says what he did yesterday, what he will do today and what are the obstacles to progress. Tasks estimates may need to be adjusted

Sprint Review: scrum team presents the outcome of a sprint to stakeholders. It is a working session to determine future adaptations

Sprint Retrospective: analysis on what went well, and what problems were encountered

Misconceptions about Agile

Agile = no process, agilists don't document nor design, agile leads to poor quality software, agile only works with small teams, agile ignores business concerns, extreme programming is the only agile process.

Agile – Advantages:

- Embraces change
- Early increments act as a prototype to obtain requirements
- Constant customer involvement and validation
- Lower risk of overall project failure
- Continuous feedback from customer, can always adjust to it

Agile – Disadvantages:

- Can fallback into cowboy coding (software development approach where developers write code without following a structured process, standardized practices, or proper documentation)
- Increases risk of feature creep (feature increase exponentially) -> it's good to adapt but it is also important to draw the line at a certain point
- Not appropriate a comprehensive requirements specification is required as contract up front (e.g., specific security system where everything is already fixedly specified before implementing the system or fire alarm system (rather waterfall), but instead implemented when building a new social network application in a startup, since there are continuous changes)
- Requires close customer involvement -> requires their time
- Can be less efficient: lots of meetings, lots of interactions when changes have to be made

Challenges

Requirements are prescriptive and may hinder creativity, reduce team autonomy and not focus on the underlying objective (customer goal). What customers want might not be what they need, and what they want might not be the most efficient solution to meet their needs. What they want might not be achievable within the available time and budget.

Quiz on Process:

- The product owner chooses features from the product backlog and assigns them to a sprint in the sprint-planning phase -> True, it represents the customer and determines what is relevant (high priority) and what is not
- For software service providers, the product owner is always the end customer -> False, it can also be somebody from inside the company
- Imagine planning the software for an automatic fire extinguisher system in a new building. Which process makes more sense? Waterfall, because the requirements are clear before construction and can be fixed in a contract according to fire regulations.
- Scrum is a software development methodology where the team works together in sprints, members meet daily, does a little bit of everything instead of doing all of one thing at once, and needs to communicate with stakeholders or end-users
- Scrum only works with large teams (e.g., > 100 developers) -> False

REQUIREMENTS SPECIFICATION

Wicked problem: problem that is hard to define and has no right or wrong solution. It can only be clearly defined and understood by solving it. Software development is often considered a wicked problem: high-level requirements can have important ramifications for low-level implementation and vice versa.

Common characteristics of wicked problems:

- No definitive formulation
- No stopping rule (never really done)
- Solutions are not right or wrong, generally just better or worse
- Many interactions with stakeholders to make sure that the software being built is the correct one

Software Requirements

They define what the system must do (not how to achieve it). They help to understand, communicate, connect and define when done. Every stakeholder must understand them.

Specification captures:

- Functional requirements: what the system must do
- Non-functional requirements: properties the system must have (called ilities) -> accessibility, usability, reliability, testability

4 properties of requirements:

- Complete: contain all information needed
- Consistent: requirements do not contradict each other and are never without answer
- Precise
- Concise: allows easier check for completeness and consistency

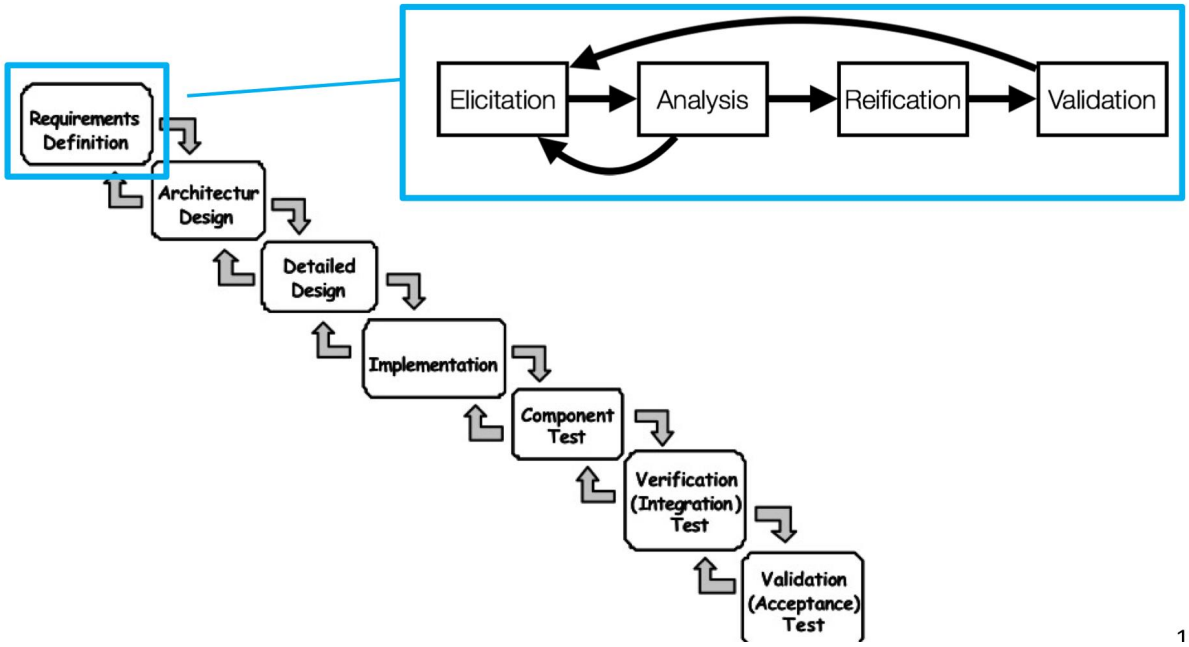
Example

1) The system must allow the user to undo the last changes functional
2) As a customer, I want to be able to run the game on all versions of Android non-functional
3) As a user, I want to be able to change the mouse sensitivity to change the moving speed functional
4) As a user I want to see a warning when the connection fails to know what is going on functional
5) As a user, I want the site to be available 99.9 % of the time I try to access it so that I don't get frustrated and switch to another site non-functional

A perfect implementation is no good if it solves the wrong problem. Validation of a specification is making sure that the right problem is solved.

A good specification is complete, consistent, precise/specific, concise (makes less space for misinterpretation), and unambiguous

Requirements specification in the software process



Elicitation: gather requirements from various stakeholders (e.g., via questionnaire, interviews, brainstorm, mock-ups)
Analysis: understand and model desired behavior, check for consistency -> negotiation phase between customer and development team
Reification/Specification: document the behavior of the proposed software system -> formally write down the requirements
Validation: check that your specification matches the users requirements

Documenting Requirements (Reification)

User Story:

- Approximately 3 sentence description of what a software feature should do, focused on the value or result that a user will receive from doing this thing
- Written in the customer's language
- Should only provide enough detail to make a low-risk time estimate (a few days)
- Used a lot in agile development
- Should not be too specific (otherwise developer is limited)
- For each user story there is a definition of done and an accepting criteria, as well as a sentence in the format roal-goal-benefit, a priority and a time/effort estimate (overall cost).
- Roal-Goal-Benefit: As a <role>, I want to <goal> in order to <benefit>
- Definition of done, acceptance tests, and acceptance criteria provide a specific description of how the story can be validated by developer and product owner to ensure it is completed correctly. They also help avoiding unnecessary functionality

- Definition of done: they are the contracts to the clients. They are useful to know when a user story is completed, and mark it as resolved. It also helps to know whether you can test the user story (if not, then a different user story is needed). Avoid making it vague, specify but do not restrict. Definitions of done almost always map directly to the sequence of calls made. The trick is to find out where the call is coming from (sender) and where it is going (receiver). Example of a definition of done: user clicks the button buy, and it appears in their purchased items, and is shipped to their home

Properties of good user stories - INVEST:

- I: independent -> no need to wait for each other to work (not always possible, especially in the beginning). An independent User Story is one that can be implemented without depending on another User Story or piece of functionality.
- N: negotiable -> leave some space for creativity, retalk about it
- V: valuable -> to other users or customers
- E: estimable -> time deadline amount (reasonable amount of time)
- S: small -> minimal viable product. The smaller it is, the faster it is to have it done
- T: testable -> customer and developer can make sure that it has been implemented in a way that is agreeable to both parties

Not user stories: implement contact list view in contractlistview.java, define product table database schema, design team creation interface, add extra tests

Role: which user to test with

Goal: what behavior to build

Benefit: informs definitions of done

Definitions of done: what tests to specify

Tasks: what needs to be done, informs effort estimate

Most common requirements errors:

- Omission: requirement is missing
- Inconsistency: conflicting requirements
- Incorrect facts
- Ambiguity

Inspection is the primary way to validate requirements (user stories, ...); should include various stakeholders

Quiz on Requirements:

1. Given the following list of functional and non-functional requirements, select all the functional requirements:
    - The crucial components of the software systems should have a 95% statement test coverage
    - As a user, I want to see a loading bar when I upload a document to the website
    - The whole website must be accessible to people who are colour blind
    - The website source code must be maintainable by non-experienced programmers

- A requirement is precise if it also describes the implementation details of a certain feature -> False, don't provide implementation. It is precise if the customer understands it.

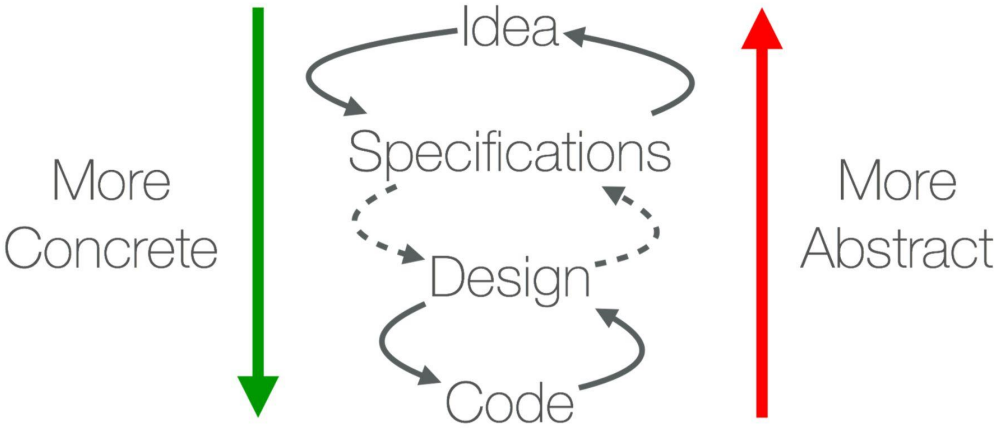
- The more requirements there are for a system, the better -> False: not too many, not too little, find balance
- Functional or non-functional: As a visually impaired user I want the option to enlarge the text in each menu and window for the application so that I am able to read it -> Non-functional, it's for the whole system
- Who should be able to understand software requirements? Developers, designers, customers (and rest of stakeholders)
- Who is involved in prioritizing the user stories in the Product Backlog? Customer, and the product owner
- Who is involved in prioritizing the user stories in the Sprint Backlog? Customer, product owner, and development team

Given the following user-story: "As a user, I want to be able to recover my password such that I don't get locked-out of the system when I forget my login credentials." Which of the following statements are correct? Select all that apply.

a) This user-story has a clear value according to INVEST
b) This user story is a good example for the role-goal-benefit format.
c) In this user-story, the "Benefit" is that the user can reset his password → i don't get logged out is benefit

DESIGN MODULARITY & HIGH LEVEL DESIGN

The software designer should be the person with overall responsibility for the conception and realization of the program



Design facilitates communication, eases system understanding, eases implementation, helps discovering problems early, increases product quality, reduces maintenance costs and facilitates product upgrade
From idea to specification (like user stories) is easier than from specification to design, because specification and the idea are in plain-text, but design is much more concrete (e.g., UML).
Modularity: break a complicated problem into simple pieces. It involves two ideas:

- Interdependence within and independence across modules
- Abstraction, information hiding, and interface. The abstraction hides the complexity of an element, the interfaces indicates how the element interacts with the larger system

Why is modularity good? It allows to focus on one module only, without thinking about the others. It minimizes complexity, and enforces

reusability, extensibility, portability, and maintainability. It creates options: decisions that are hidden inside modulus and that can be changed create options

Modularity allows parallel work on independent parts of the system, and is essential for enabling the development of complex systems

Software Architecture

It is defined as the set of principal design decisions about the system. It includes structure (components, connectors), behavior, and non-functional properties. It describes the gross structure of the system (main components and their behavior, connections between the components and configurations: bind components and connectors together in a specific way).

Components: elements that encapsulate processing and data at the architectural level. A component is an architectural entity that encapsulates a subset of functionality, restricts access via explicit interface, and has explicit environmental dependencies

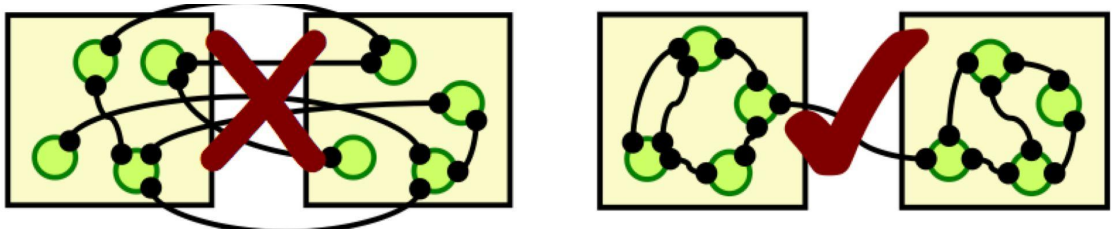
Connectors: a connector is an architectural entity tasked with effecting and regulating interactions between components. They often consist of method calls, but can be much more. They frequently provide application-independent interaction mechanisms. In large heterogeneous systems they are often more challenging than components.

Configurations: an architectural configuration is a set of specific associations between the components and the connectors of the system's architecture. Configurations bind components and connectors together in

a specific way. A configuration differentiates a bag of components and connectors from an implementable system.

Topological goals:

- Minimize coupling between components: the less components know about each other, the better (also known as information hiding) -> enforces independence
- Maximize cohesion within each component: components should be responsible for a logical service, and extraneous functionality should not be present -> enforces interdependence



User Modeling Language(UML)

A diagram language to visualize, specify, construct and document a software system.

Two types of UML diagrams:

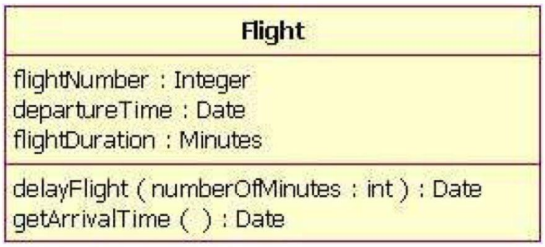
- Behavioral / Dynamic diagrams: state machine diagram, sequence diagram, activity diagram (used to represent the flow of activities and actions in a system, rectangle boxes represent actions and arrows represent flow of activities) -> dynamic view of the system (model how the system components interact with each other dynamically at runtime)

- **Structural / Static: class diagram, object diagram, component diagram, deployment diagram -> static view of the system (capture key elements and their relationships within the system)**

Class Diagram

Focus on core classes the design requires and the relationships between these classes.

- Class name (*italics* means abstract)
- Attributes (fields)
  - Name : Type
- Operations (methods)
  - Parameter : return type
- Can also be used for interfaces (without fields or just constants)



Questions:

- *What is an abstract class?* Cannot be instantiated, but can be extended. can have body.
- *What is the difference to an interface?* Cannot be instantiated, but can be implemented. No method body, only method signatures

Difference between an interface and a super class: extends vs implements, dotted line, straight line

Associations:

- **Bi-directional: both classes are aware of each other**
- **Role: usually maps to a field name**
- **Multiplicity: indicates how many instances can be linked (e.g., a list of)**
- **Uni-directional**

Difference between uni-directional and bi-directional association: if something changes in class that associates, nothing happens for uni-directional. Less associative is better. Code is easier to change because it is more independent. Aim to have as few associations as possible. Put lists as associations, not as fields.

Aggregations:

- Advanced type of association
- Contained object is part of container
- Two types:
  - Basic aggregation: children can outlive parent (e.g., car - wheel) -> white diamond in UML
  - Composite aggregation; children life depends on parent (e.g., company - department) -> black diamond in UML

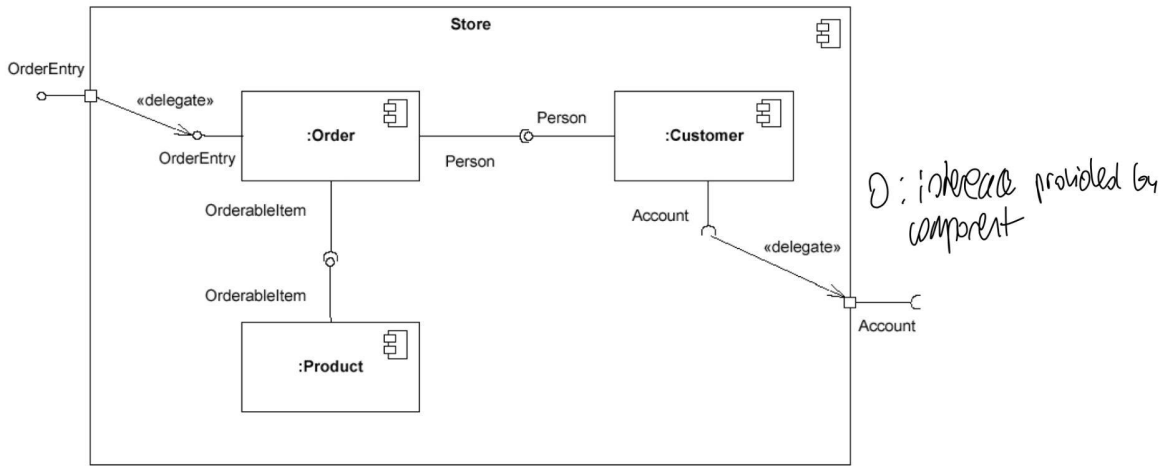
Class Diagrams – Comments:

- Do not put inherited fields and methods in subtype unless you are overriding them
- Class names: singular nouns or doers (start with capital letter)
- Method names: verbs, actions or isX checks (start with lowercase letter -> camelCase)

Component Diagram

Focus on core software components.

Example



Components can be composed of other components or classes

Sequence Diagram

Low-level design tool to describe sequences of invocations between objects. Sequence diagrams capture the order of execution of particular methods and the data being passed between them.

Full arrow: synchronous (blocking/waiting until message returns) or asynchronous (can continue processing). Only represent associations, not also attributes.

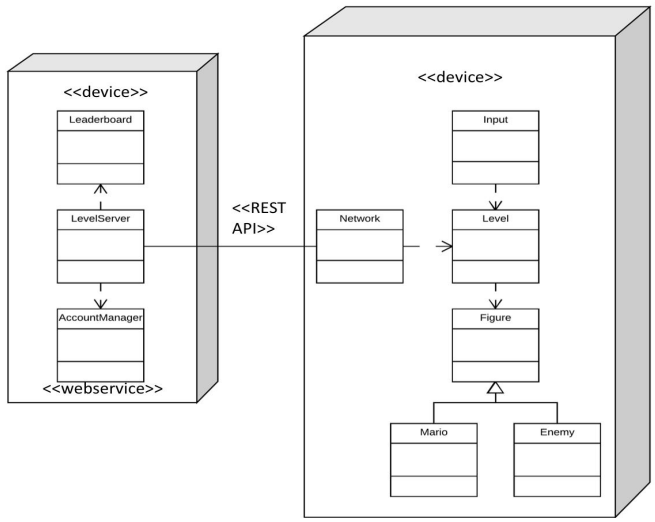
State Machine Diagram

Helpful for modeling event-based systems (e.g., user interfaces)

Deployment Diagram

Focus on identifying physical devices and what software components they hold (where system parts will execute).

Example



A few comments:

- Differentiate between component and deployment diagram: a component shows the structural relationships (connections, interactions) between the components of a system, so the focus is on the components, not on the physical deployment of the system (don't care about where the software actually executes). A deployment diagram instead has a focus on the physical deployment of the system on hardware infrastructure (on which physical device software is deployed)
- For deployment: identify all physical devices and associate each component with a device. Make sure to specify who requires and provides something.

Decomposition

Describes the process of breaking down a complex description of a system into more manageable pieces, The goal is to make common tasks simple and difficult tasks possible. It is an iterative process. Find entities, link them together, figure out actions of the story and bind them to the user stories, prototype, formalize (design with UML), implement.
2 approaches:

- Top-down: from general (high levels of abstraction) to particular (low levels of abstraction)
- Bottom-up: from particular to general. Low level implementation errors affect the higher ones.

Quiz on Design Modularity and High Level Design

- A java interface is a good example of information hiding, since an implementation o fit might change while the interface itself remains the same -> True
- The more information hiding used when designing a software system, the harder it is to understand -> False
- Decomposing a system can either be done bottom-up or top-down. Are the following bottom-up approaches? 1) Splitting large system parts into smaller, more manageable parts -> False. Given two classes Mario and Luigi, creating a common superclass PlayerCharacter -> True
- Information hiding is meant to force unprofessional programmers into using only public (safe) methods -> False

- There can be multiple diagrams for one system -> True
- Deployment diagrams are only taken into account after implementing the application -> False

Summary

- Design consists of multiple views (static and dynamic)
- Diagrams are communication tools
- Designing is an iterative refinement process: start rough, flexible then clean up

DECOMPOSING USER STORIES

Decomposition is not neatly sequential.

Activities when decomposing:

THIS IS NOT AN ORDERED SET OF STEPS

NOUNS	Identify actors (objects/classes)
VERBS	Identify behaviours and associate those with actors
OWNERSHIP/ KNOWELDGE	Identify relationships between actors (fields)
SEQUENCES	Identify calls between behaviours

Reid's video: find entities, link entities, bind actions, [prototype, formalize, implement]

Quiz:

- One part of decomposing a user story is to step through a scenario, also called prototyping, to ensure that the in-progress design matches the user story -> True

3. You are given a user story for a social networking app for mobile phones called WhatUPNet: *As a user I want to search for other users by their name so that I can add them to my network.*

   When decomposing this user story (solely based on this user story), which are the relevant entities that a design should have?

User, users, name, network, WhatUPNet
User, users, name, network
User, name, network
User, network

- There should be an entity network that has a method to add a user -> True
- There should be an entity user that has a method to add the user to a network -> False
- In a class diagram for a pizza ordering application, there is a composition between a class Pizza and a class Ingredient, representing the ingredients of the pizza.
  An object of type Pizza is composed of objects of type Ingredient, and the ingredient objects of a pizza can exist even after the pizza object ceases to exist -> False, composition
  If there is no object of type Pizza in the application, there should also be no object of type Ingredient -> True

DESIGNING APIs AND REST APIs

Application Programming Interfaces:

- Provide a predictable mean for programmatic interaction
- Specify the protocols, names, parameters, and data formats for interacting with software components
- Hide implementation details (e.g., algorithms)
- Provide a lot of value
- Once clients start to use them, they are difficult to change (APIs are forever)
- Spend time designing APIs "right"

Usability Principles:

- Good visibility: possible actions and states must be expected
- Good conceptual model: helpful, consistent, and complete abstractions clarify the correct model of the system
- Good mapping between actions and results: natural (save method should save and not delete)
- Good feedback about results of actions: full and continuous (if something fails during the performance of an action provide information about what failed and why)

Characteristics of a good API:

- Easy to learn
- Easy to use, even without documentation
- Hard to misuse

- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to understand

Principles for designing good APIs:

- Do one thing and do it well (similar to SRP): functionality should be easy to explain, users should not be surprised by the behavior of the API
- APIs should be as small as possible but no smaller: when in doubt, leave it out (APIs are forever)
- Implementation should not impact API: leaking details into API is a fundamental mistake
- APIs should be easy to use (usability)
- Minimize access, maximize information hiding
- Names matter: should be self-explanatory
- APIs should be flexible and documented: a flexible API can be used by many different clients in many ways. Good documentation makes it easier for the clients to understand how to use the API effectively
- Avoid long parameter lists
- Return descriptive objects (e.g., user type and not string)
- Avoid exceptional returns; do not return null values or empty lists
- Handle exceptional circumstances
- Favor immutability: client should not be able to change code

- Nouns are good (for resources), verbs are bad
- Plurals are better
- Error handling is important
- Specify what is needed and return format
- Favor private classes, fields, and methods

Address two questions:

1) What is the goal (use cases) of the API?
2) Who will be using the API?

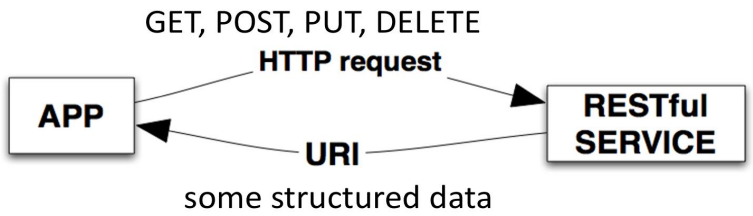
Technical debt: concept in SE that reflects the cost of additional rework caused by choosing an easy solution instead of using a better approach that would take longer

REST

- Representational State Transfer
- Definition: a network of web pages, where the user progresses through an application by selecting links, resulting in the next page being transferred to the user and rendered for their use
- Stateless (usually based on HTTP): every request is self-contained, clients maintain their own state; server does not retain state about clients but only keeps resources
- Resource-Oriented: key elements of any RESTful service are resources. Every resource is identified by a Uniform Resource Identifier (URI). Every Server maintains resource structure and passes back relevant links to related resources -> connectedness

- Simple, reliable, scalable, extensible
- Constraint 1: rest based services communicate using intermediate representations, this means that when a request is made, the service responds by generating a representation of its internal data in XML or JSON format
- Constraint 2: rest services are based on self descriptive hypermedia documents (when you get a response from a service it doesn't just send a small amount of data, but also other forwarding links that help you understand other actions you can do with this response data you received)
- When a response is returned, it contains a header and a body: the header contains all kinds of metadata, where one important piece is the status code.
- Constraint 3: statelessness, which means that rest clients are responsible for maintaining all of their own state as they proceed through their transaction (ask server for one specific data not for whole set)

REST Methods



HTTP methods (subset) to manipulate resources

- GET: get a resource
- PUT: update an existing resource
- POST: create a resource
- DELETE: delete a resource

Quiz:

2. You are developing a service for a university where users can access a variety of textbooks. You are currently working on a draft of the API calls, which you want to ensure is RESTful. Mark which of the following are well designed REST API calls. Note: multiple can be true.

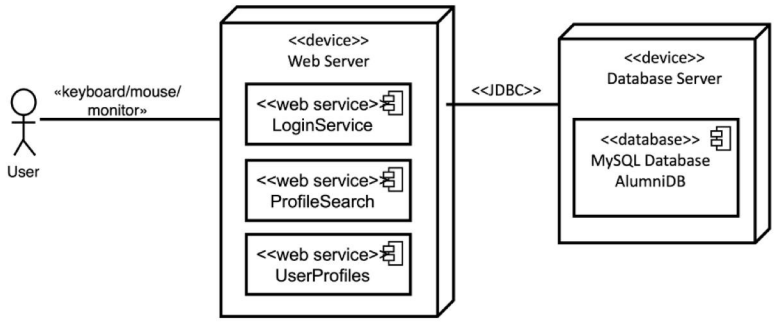
- GET /books/physics?author=Goldstein
- GET /books/chemistry/organic/getAllBooks
- DELETE /books/chemistry

- **The statelessness property of a REST based system is enabled through connectedness -> True, it allows to know where to go next**

Last year's Midterm:

- **The scrum master is in charge of picking development tasks for the sprints and organizing the daily scrum -> False (product owner)**

- The sales team states that switching to Scrum can be difficult if customers don't want to be involved other than in beginning and end of the process -> True
- XP has pair programming as a core principle while Scrum does ot explicitly say anything about it -> True



The diagram shows that the ProfileSearch component has to provide an interface for searching the user profiles. [True / False]

The diagram lacks a device that contains the web browser that a user interacts with to access the web services on the web server. [True / False]

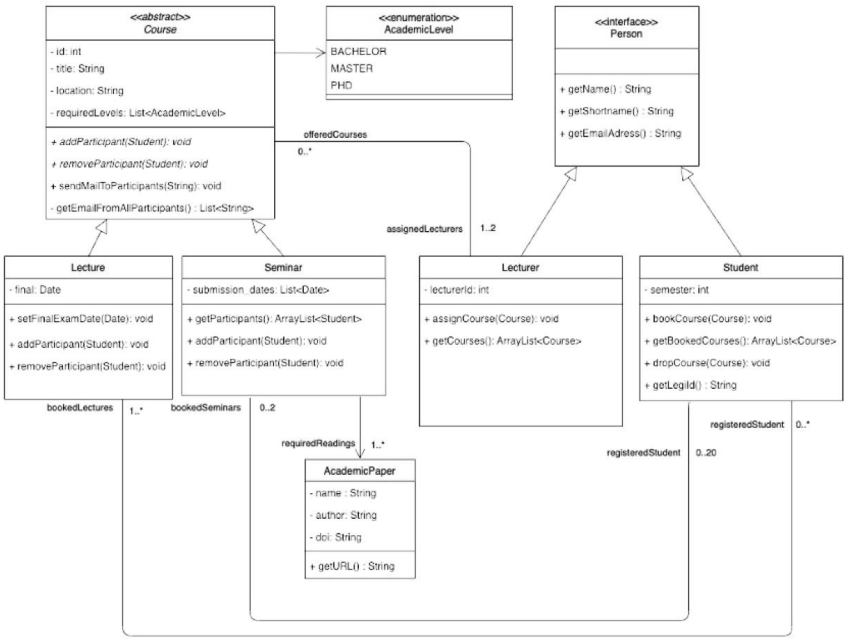
Section C – Diagrams

… implement a new feature: alumni and students should be able to be matched and connected with each other for mentoring. Alumni that want to become mentors and students that want to become mentees must both fill out an online application. **An elaborate matching process** then matches mentors with mentees taking into account the past mentoring history, specified preferences, availabilities, background, interests, location and more.

Since the matching feature will result in a lot of **new and complex functionality to be added to your system, you decide to depict the matching algorithm** in a UML diagram.

Please select which of the following diagram(s) are best suited for this purpose. (Multiple can be true)

[Class / state machine / sequence / deployment diagram]



state for each of the following requirements if they are supported by the class diagram

The association “assignedLecturers” should be implemented in the abstract class Course as a field/attribute with that name (i.e. assignedLecturers). [True / False]

By dividing up the system into a Course, a Person, a Lecturer and a Student class, we decompose the system into more manageable pieces [True / False]

The class student is an abstraction of the concept of a student that hides irrelevant details. [True / False]

The public methods of the abstract class Course represent an interface that indicates how the class interacts with other elements. [True / False]

You and a group of fellow students have noticed that supermarkets in Switzerland often sell the exact same products but at varying prices, sometimes allowing you to make substantial savings by simply comparing the prices …

U1. As a user, I want to be able to select which supermarkets will be included in the scope of my searches, so that my search results do not contain supermarkets that are inaccessible or not of interest to me.

U2. As a user, I want to be able to search for a product and find out for which price each supermarket in my search scope offers it, so that I can make a good decision on which supermarket to go to for buying the product.

- In a class diagram, there should be a class Scope, a class Supermarket and a composite aggregation between the Scope and the Supermarket. [False]
- In a sequence diagram for U2, there should be a message (drawn as an arrow) sent from a Scope object instance to a Supermarket object instance, and the arrow should be labelled with a method name (similar to) “getPrice()”. [False]

...decide to build a service called MeetNow to allow users to meet up for events. The application shall allow people to create events, register for events, and search for events close to a specified location, as well as based on the type of event, such as going for drinks, playing sports, or going for a walk. Finally, the app should allow users to invite other users to an event based on their name.

To allow others to also extend your idea, you decide to provide a public REST-API for your service.

At the moment, you are designing the backend of your application and the REST interface. You have to come up with a first draft of the REST API, in particular its resources and the behaviour for certain methods. After you noted down some of the specifics, you sketch out examples of REST API

- "PUT /events" with the data {"type":"sports", "name":"Triathlon Swim Practice", "location":"47.372372729077014, 8.5329314693058", "date":"15.04.2022", "time":"16:00"} is a well-designed REST API call that creates a new event of type "sports" with the name "Triathlon Swim Practice" for the specified location, date and time. [False]
- "POST /users/JesseS" with the data {"type":"sports", "name":"Forest Run", "location":"47.38240375091224, 8.573676957552193", "date":"18.04.2022", "time":"06:00"} is a well-designed REST API call that creates a new event for the user JesseS with the event of type "sports" with the name "Forest Run" for the specified location, date and time. [False]

```
public class User {
    private String userName;
    public User(String name) {...}
    public String getName() {...}
    public void setName(String name) {...}
}
```

- The API for the class User (that is presented in the code) is well-designed according to the principles covered in the course [True]

MODULAR DESIGN AND DESIGN PRINCIPLES

Benefits of a good (modular) design: easier to maintain afnd resolve. Is reusable and if we need to make a change we do not affect the rest of the code.

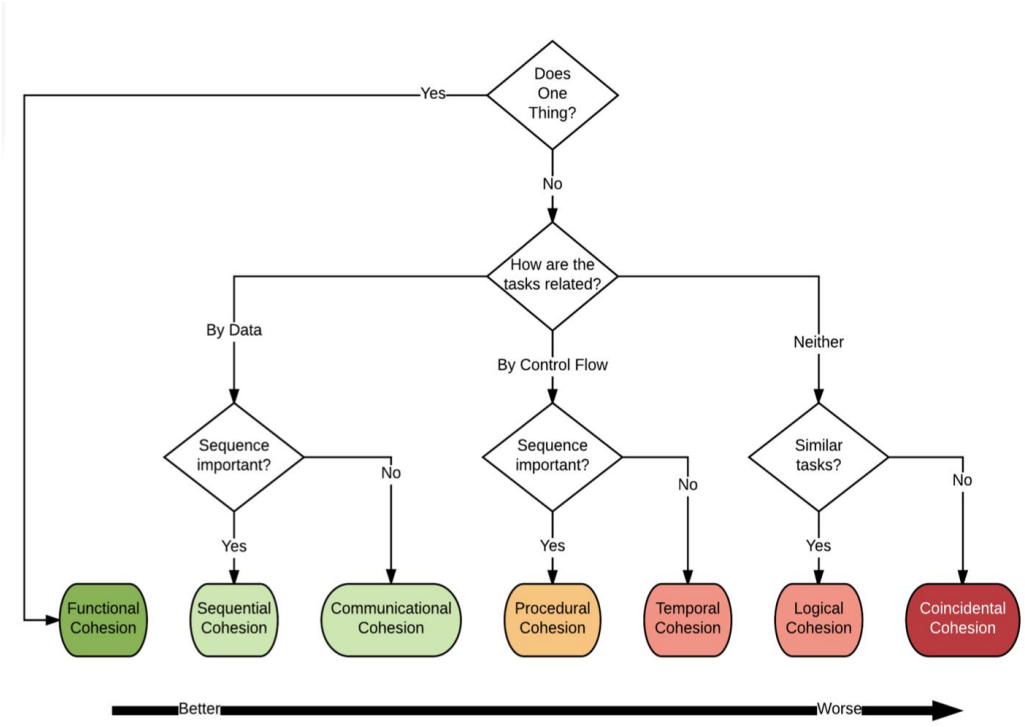
Properties of a good design:

- **High cohesion: functions in a module should be closely related**
- **Low coupling: modules should only depend on as few modules as possible**

- Encapsulation: restrict access to some of a module's components (hide internal details / information). Only the necessary things should be provided to the outside

Cohesion

Classes with low cohesion are harder to reason about as they often have many competing maintenance problems because changes to fix one defect within a class may actually be by design for another feature provided by the class (the larger the class, the more likely this problem is to be encountered). Cohesive classes are generally small, and this makes the number of classes within a system to be high. Cohesive classes have a small set of private fields that make sense to the majority of public methods within the class (might be hard to find the right class in the system)



Functional cohesion (best) – Module does a single well-defined task

Sequential cohesion (very good) – output from one part is the input to next part (e.g. a function which reads data from a file and processes the data).

Communicational cohesion – operate on the same data (e.g. a module which operates on the same record of information)

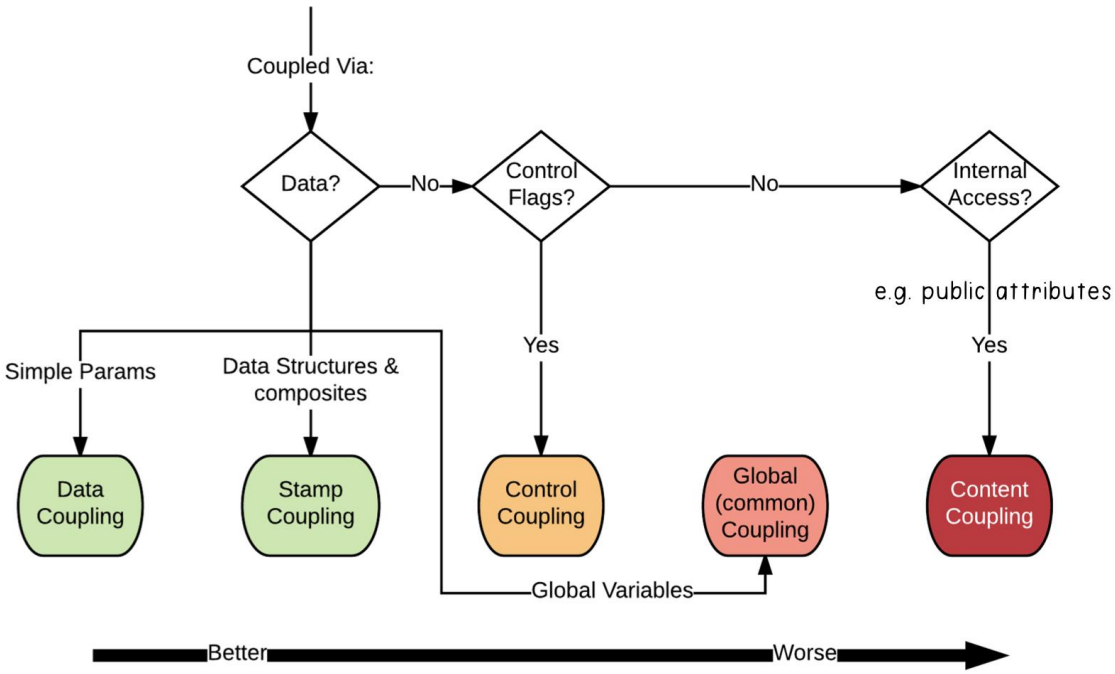
Procedural cohesion – always follow a certain sequence of execution (e.g. a protocol).

Temporal cohesion – processed at a particular time in program execution, but not the same thing (e.g. a function which is called after catching an exception which closes open files, creates an error log, and notifies the user).

Logical cohesion (bad) – logically related but otherwise different.

Coincidental cohesion (bad) – grouped arbitrarily; the only relationship between the parts is that they have been grouped together (e.g. a "Utilities" class).

Coupling



Data coupling – interacting through simple data types (e.g. parameters)

Stamp coupling – interacting through data structures or composite objects and only using parts of it (e.g. passing a whole record to a function that only needs one field of it)

Control coupling – modifying execution by sending control flags (e.g. passing a what-to-do flag)

Global/Common coupling – global variables (changing the shared resource implies changing all modules using it)

Content coupling – internal modification of other classes (e.g accessing local data of another module)

Coupling cannot be avoided entirely, there will always be coupling, but aim for low coupling

SOLID Design Principles:

- **S: Single Responsibility: classes should do one thing and do it well**
- **O: Open/Closed: classes should be open to extension and closed to modification. No need to modify a class for extending the behavior -> encourages developers to think about what parts in the system should enable future change and which parts should not**
- **L: Liskov substitution: an object of a superclass should always be substitutable by an object of a subclass. Subclass has same or weaker preconditions, but same or stronger postconditions. It shows that a design can be structurally consistent but behaviourally inconsistent -> we must so verify whether the pre and postconditions in properties will gold when a subclass is used. Derived methods should not assume more or deliver less.**
- **I: Interface segregation: clients should not be forced to depend on interfaces they do not use / need**
- **D: Dependency Inversion: depend on abstractions not implementations. High-level modules should not depend on low-level**

modules, both should depend on abstractions. Attractions should not depend on details, details should depend on abstractions

Quiz:

- When making a change to a method methodA in class A also requires the change of a field/attribute in another class B then there is a coupling between these two classes -> True
- Classes with high cohesion often cause maintenance problems, as changes made to one feature might conflict with other ones within the class -> False
- A system with a lot of coupling is automatically also more rigid and more difficult to evolve -> True
- If classes adhere to the SOLID principles, they generally tend to be more modular and easier to evolve -> True
- Which design principle is violated in a file with 52 import statements? Low coupling

DESIGN PATTERNS AND ARCHITECTURAL STYLES

A design pattern is a general repeatable solution to a commonly encountered problem. It is a template for how to solve a problem in a good modular way. It leverages design expertise, names design structure explicitly and eases communication -> not a complete / finished design.
Components of a DP: name, intent (what problem it solves), solution (participants and structure), consequences (known uses, related patterns, pros and cons). Design patterns are language-independent and

cannot be mechanically applied (must be translated to a context by the developer)

Design Pattern Classification

- By purpose:
  - Creational patterns: concern the process of object creation
  - Structural patterns: deal with composition of classes or objects
  - behavioral patterns: characterize the ways in which classes or objects interact and distribute responsibility
- By scope:
  - Class: deal with relationships between classes and their subclasses
  - Object: deal with object relationships that can be changed at runtime and are more dynamic

To apply a design pattern, one must know it, recognize the problem, apply the design pattern and integrate it in the system.

Observer Design Pattern

Definition: the Observer pattern defines a one-to-many dependency between a set of objects so that when one object changes state, all of its dependents are notified and updated automatically.

Use cases: when you want multiple objects to get notified of changes in the state of another object. The notifying object (publisher) send an event (publication) to all its observers (subscribers).

Advantages: 1) The Observer Pattern provides an object design where subjects and observers are loosely coupled. Observer nicely isolates subsystems, since the subjects in the subsystems don't need to know anything about each other except that they implement a certain interface (the Observer interface). This isolation makes the code much more reusable. Subject only knows the observer interface, new observer can be added at any time.

2) New observers can be added at any time. Because the only thing the subject depends on is a list of objects that implement the Observer interface, we can add new observers whenever we want. We can replace any observer at runtime with another observer or remove it and the subject will keep working.

3) Subjects or Observers can be reused independently of each other, since they are not tightly coupled.

4) Changes to either the subject or an observer will not affect the other. Because the two are loosely coupled, we are free to make changes to either, as long as the objects still meet their obligations to implement the Subject or Observer interfaces.

5) We never need to modify the subject to add new types of observers. If we have a new concrete class that needs to be an observer we don't need to make any changes to the subject; all we have to do is implement the Observer interface in the new class and register as an observer. The

subject doesn't care, it will deliver notifications to any object that implements the Observer interface.

6) Because the subject is the sole owner of the data, the observers are dependent on the subject to update them when the data changes. This leads to a cleaner Object Oriented design than allowing many objects to control the same data.

Disadvantages: 1) Implementing the Observer Pattern might make the code too complex.

2) Notifying all the observers whenever the subject changes state can be computationally intensive, especially if there are a large number of subjects and observers.

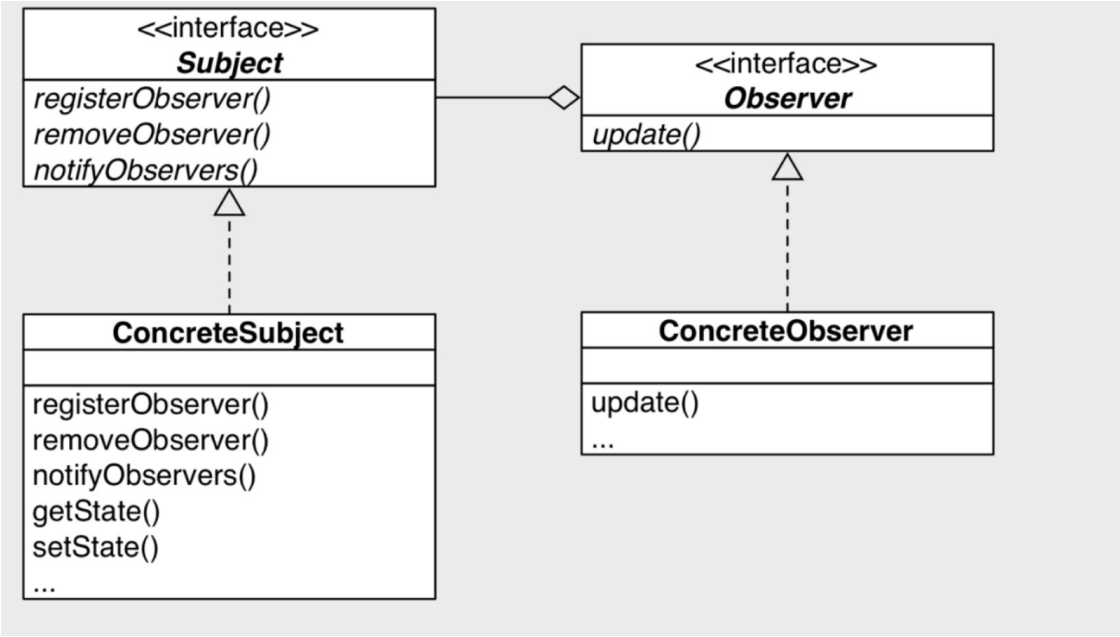
3) The order of notifications is important, but the Observer does not guarantee it. If the order is important for the correct functioning of the program, then the program might fail.

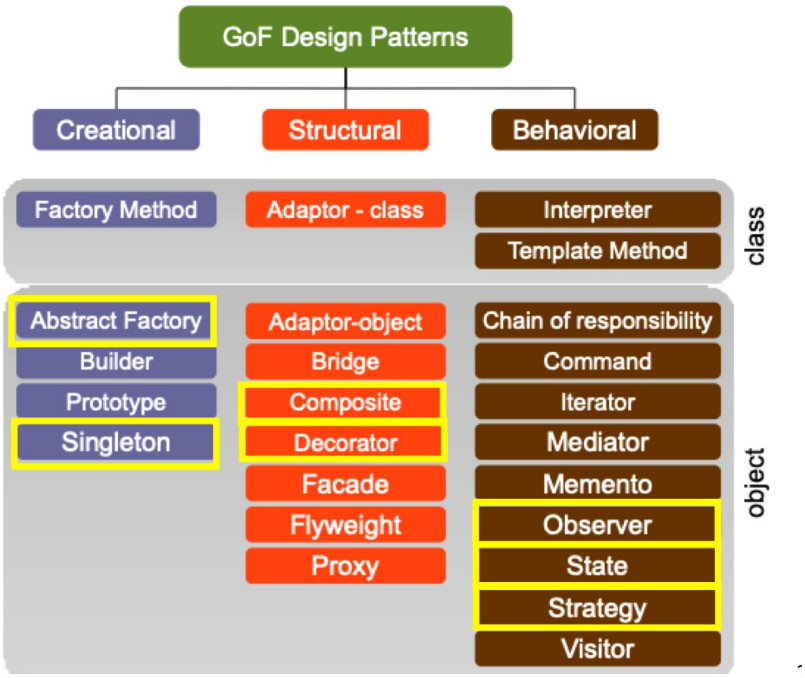
4) There is no guarantee that a subscriber won't be notified of an event after the subscriber cancels its subscription. Memory leaks are easily created by passing references to observers, and they can

happen if the observer fails to unregister from the subject when it no longer needs to receive notifications.

5) Publication events can propagate alarmingly when observers are themselves publishers.

Class Diagram

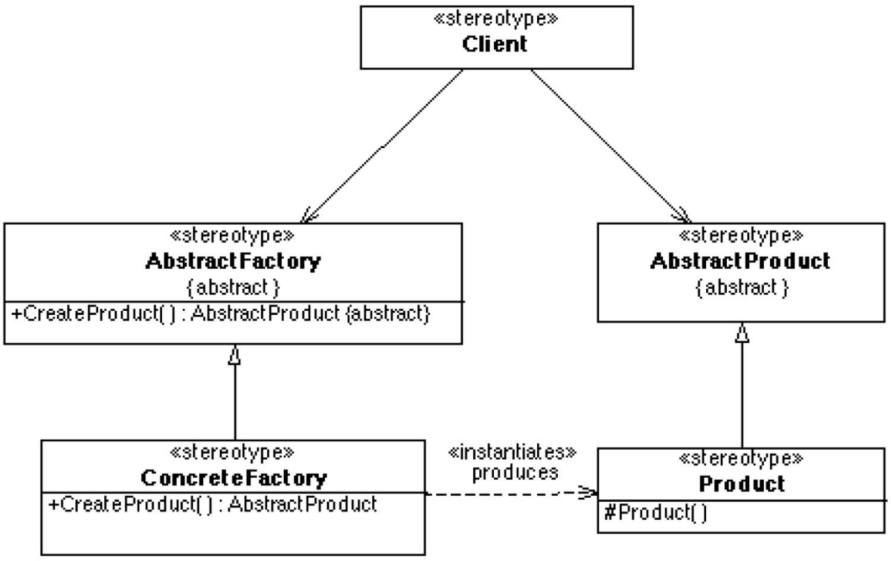




Abstract Factory Design Pattern

Intent: interface for creating families of related or dependent objects without specifying their concrete classes. The factory object has the responsibility for providing creation services for the entire platform family. Clients never create platform objects directly, but instead they ask the factory to do that for them.

Class Diagram



Singleton Design Pattern

Definition: the Singleton Pattern ensures that only one object is instantiated for a class, and provides a global point of access to it.

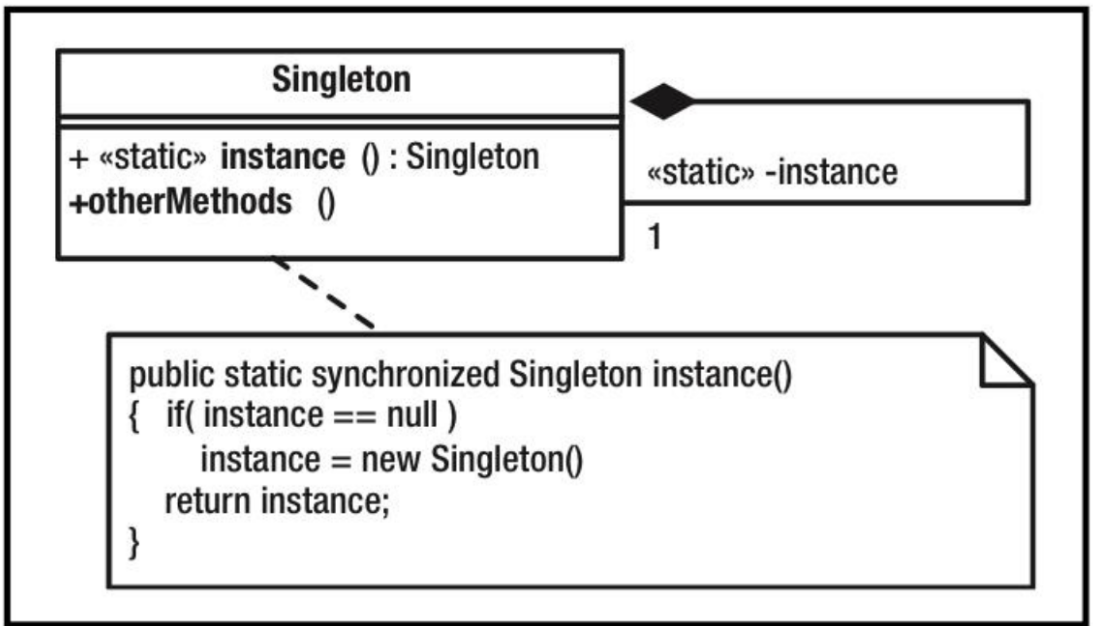
Use cases: When you need to ensure that only one instance of a class is created, and a global point of access to it is provided. The global point of access to the instance can be useful when you want to ensure that other parts of the system can access the shared resource without having to pass the instance around as a parameter.

Disadvantages: a) it can introduce tight coupling between the code that uses the singleton and the singleton itself, which can make it difficult to change the implementation of the shared resource, and if you make a change to the Singleton you will likely have to make a change to every object connected to it.

b) It might break the Single Responsibility Principle (SRP): The singleton is not only responsible for managing its one instance and providing global access but also for whatever is the role of the Singleton class in the complete program; so it could be argued that it is taking on two responsibilities.

c) A class with a singleton cannot be subclassed since the constructor of the Singleton is private and a class cannot extend a class that has a private constructor. To be able to extend it the constructor would need to be changed to protected or private but then it's not really a Singleton anymore because other classes can also instantiate it. If we change the constructor then all the subclasses will share the same instance variable since the implementation is based on a static variable.

Class Diagram



Strategy Design Pattern

Definition: the Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it. The Strategy Pattern encapsulates interchangeable behaviors an uses delegation to decide which behavior to use.

Use cases: 1) When the code has multiple algorithms that are similar but differ in their implementation, the Strategy pattern allows you to select the appropriate algorithm at runtime based on the specific needs of your application. When we would like to change the behavior of an object at runtime without modifying the object itself.

2) When we need different implementations of an algorithm for different types of objects without creating a separate subclass for each implementation.

Advantages:

1) The Strategy pattern allows you to encapsulate each algorithm or behavior in a separate class, which makes it easy to add new algorithms or behaviors without changing the existing code. The strategy pattern is flexible because it makes use of object composition, and clients can change their algorithms at runtime by (simply) using a different strategy object.

2) It is a good alternative to subclassing. Rather than overriding methods from the superclass, an interface is created.

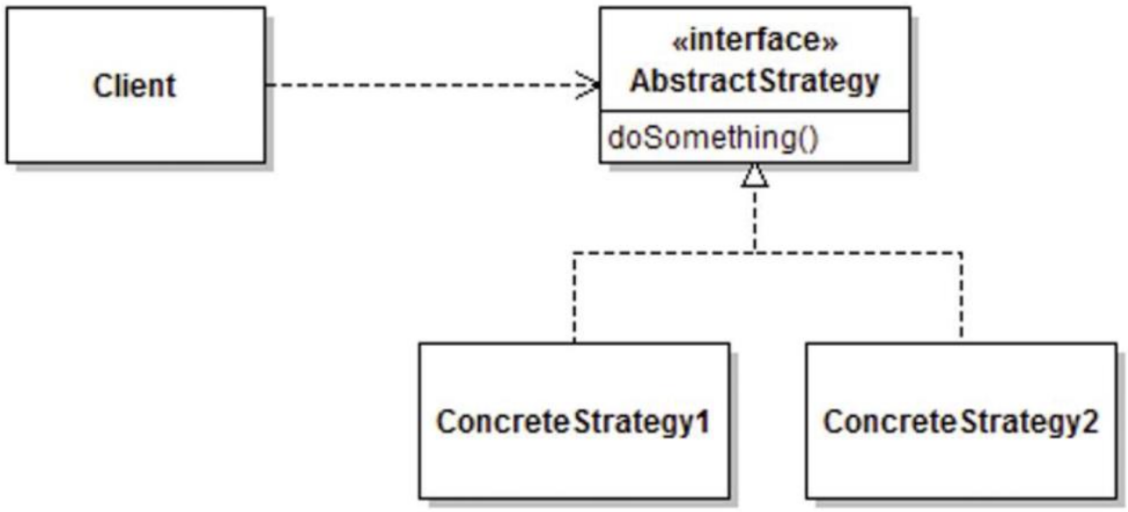
3) An object can take on multiple forms -> Polymorphism.

Disadvantages: 1) The application must be aware of all the strategies to select the right one for the right situation.

2) Context and the Strategy classes normally communicate through the interface specified by the abstract Strategy base class. Strategy base class must expose interface for all the required behaviours, which some concrete Strategy classes might not implement.

3) communication overhead is small. Some of the arguments passed to the Strategy objects may not be used.

Class Diagram



The strategy pattern does not change the behavior of an object like the State DP does, but the calculating algorithm changes.

State Design Pattern

Definition: the State pattern allows an object to alter its behavior when its internal state changed. The object will appear to change its class. It encapsulates state-based behavior and delegates it to the current state.

First sentence: since the State pattern encapsulates state into separate classes and delegates to the object representing the current state, we know that behavior changes along with the internal state.

Second sentence: from the client perspective if an object can completely change its behavior, then it appears that it is actually instantiated from another class. In reality, composition is used to give the appearance of a class change by simply referencing different state objects.

Use cases: the State Pattern is used when we want to implement finite state machines (systems that have a finite number of states), or when we need to handle a large number of conditional statements in a correct way (avoiding the switch statement antipattern).

Advantages: 1) prevents the Switch statement antipattern, which consists of using a large switch statement or a series of if-else statements to execute different code paths based on the value of a particular variable; and if a change request comes, code is really hard to be modified without changing it and causing bugs in code that was working before. The State Design Pattern eliminates these long and hard-to-maintain switch statements when an object changes its state.

2) State machines are easier to maintain since all the behavior for a given state is in one place. 3) The behavior of each state is localized into its own class.

4) Closes each state for modification, and lets the Context object open to extension by adding new state classes -> Open/Closed Principle.

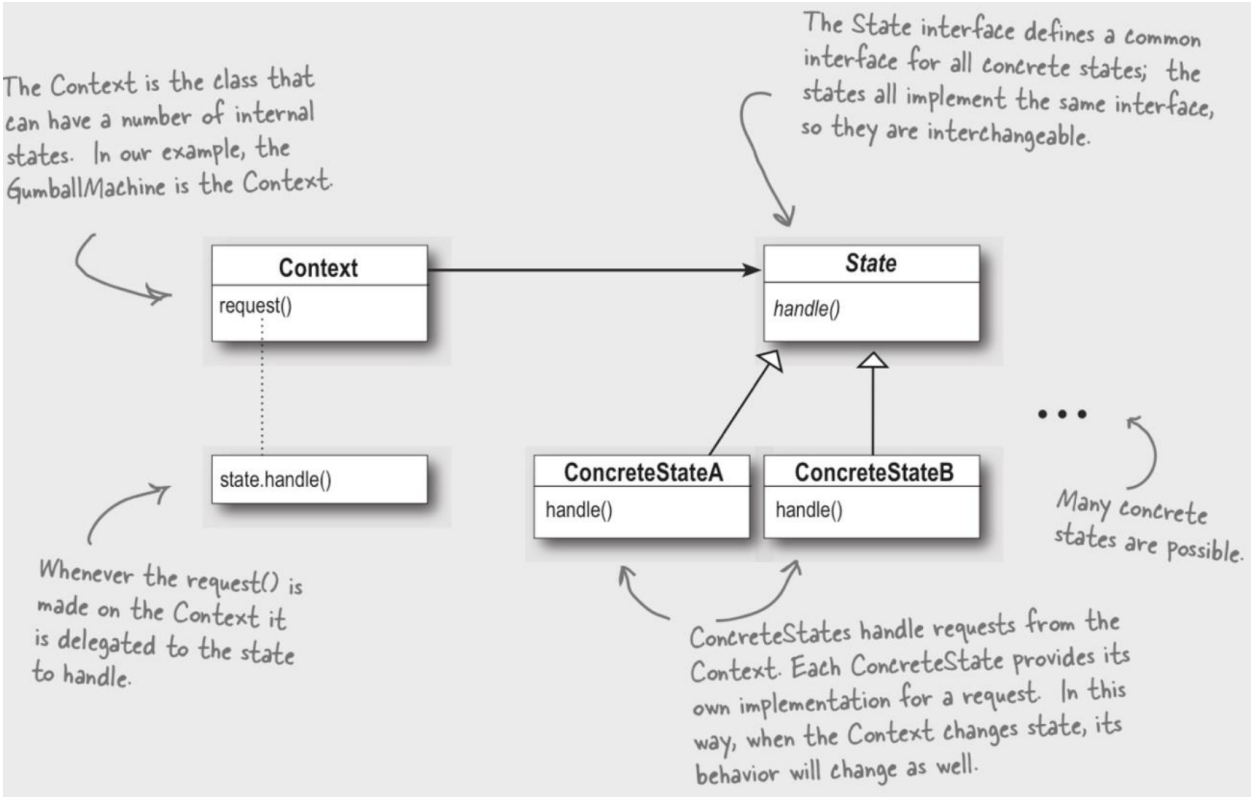
5) These class structure maps more closely to the object (e.g. Gumball Machine) diagram and is easier to read and understand.

Disadvantages:

1) It increases the number of classes in the system and maintenance problems might arise.

2) Code may become too complex, especially if only a few methods change behavior with state. But if we choose to have a program that has a lot of state and you decide not to use separate objects, you'll instead end up with very large, monolithic conditional statements. This makes your code hard to maintain and understand.

Class Diagram



Composite Design Pattern

Definition: The Composite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly (with common methods to be called on composites and leaves).

Use cases: used where we need to treat a group of objects with similar properties as a single object, for example when representing directories, which are files that contain other file (file system).

Advantages: 1) the Composite Pattern allows to build structures of objects in the form of trees that contain both compositions of objects and individual objects as notes. Using a composite structure, we can apply the same operations over both composites and individual objects. In most cases we can ignore the differences between compositions of objects and individual objects. The children of a composite can either be leaves or other composites theirselves.

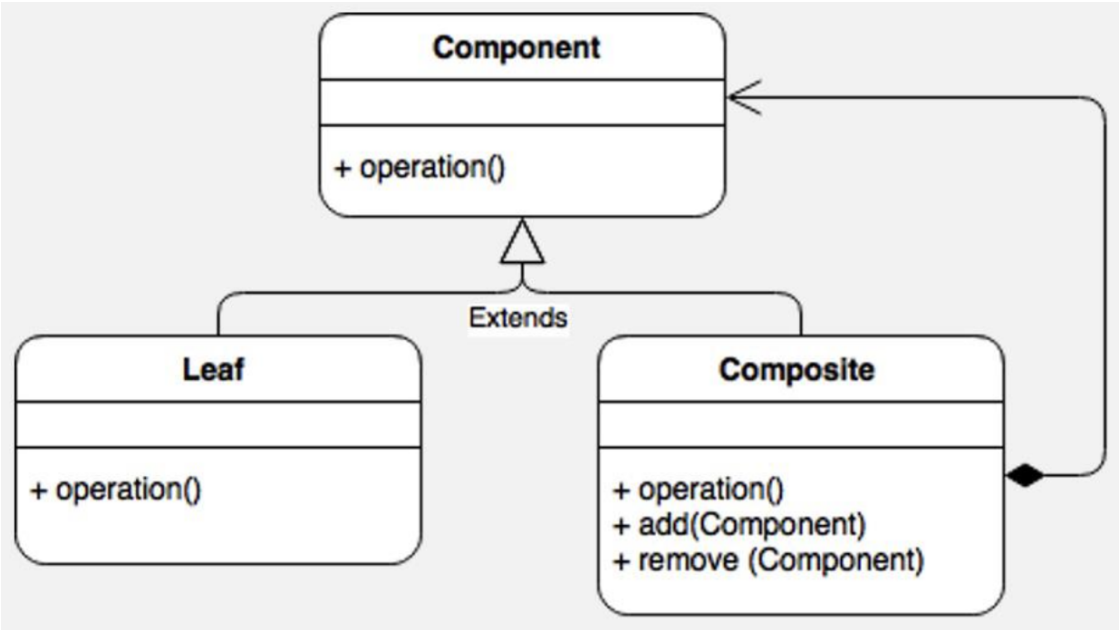
2) The container is simple to implement because it treats all contents uniformly.

3) It is simple to add or remove objects from the tree structure and allows to work with objects in the tree as a single unit.

4) Individual objects in the tree structure do not need to be aware of their position in the tree.

Disadvantages: It's not always meaningful or appropriate for every Composite or Leaf to implement every method of the Component. It's an awkward runtime error if an unimplementable method throws an exception.

Class Diagram



Decorator Design Pattern

Definition: the Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Use cases: when we want to decorate some object with additional features (modify their functionality) at runtime, while being able to treat the decorated objects like any other object of the undecorated type and not modify the code of the Object class.

Advantages: 1) decorators can simplify class hierarchies by replacing subclassing with containment. If we relied on inheritance, we would have to change existing code anytime we wanted a new behavior.

2) Adheres to Open-Closed Principle

3) Decorator allows to assemble (at runtime) a composite object that contains exactly the mix of capabilities needed without having to know which of these capabilities will be needed when the code is being written.

4) Extends functionality of an object without affecting any other object

5) Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like. You can wrap a component with any number of decorators.

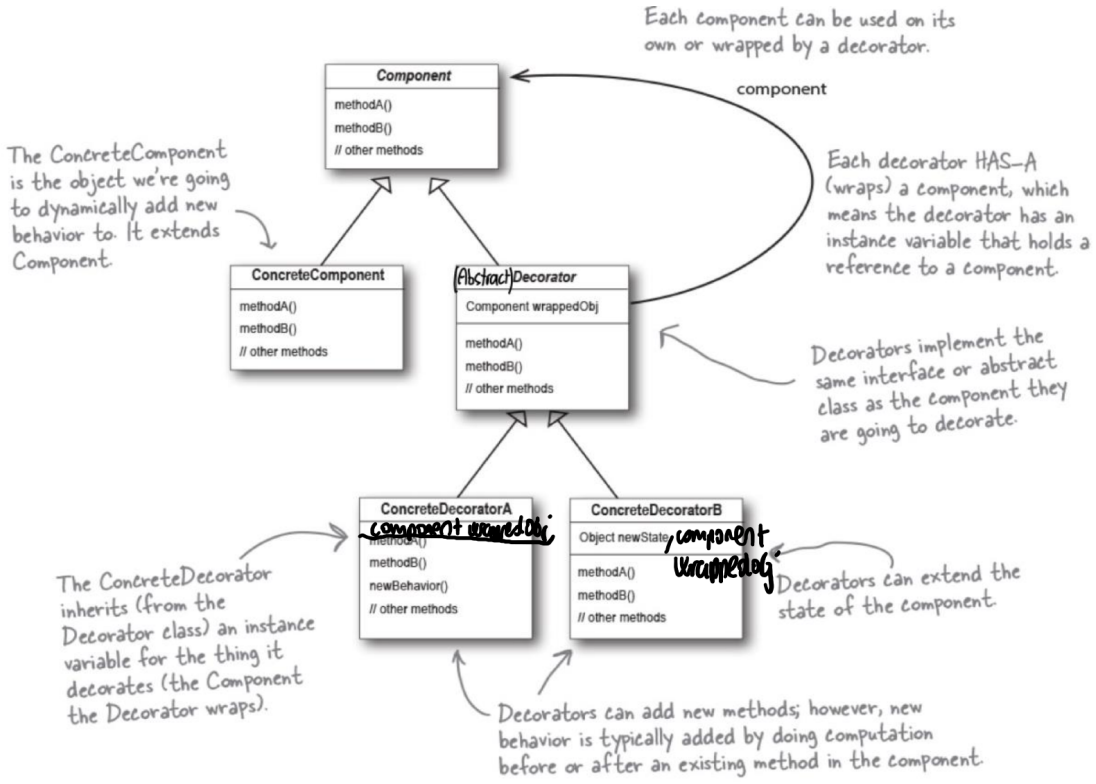
6) Decorators can be inserted transparently, and the client never has to know it's dealing with a decorator.

7) Prevents class explosion, which occurs if inheritance is used instead of composition with the Decorator pattern.

Disadvantages: 1) Can create lots of small classes, and overuse can be complex

2) Increases complexity of code needed to instantiate a component

Class Diagram

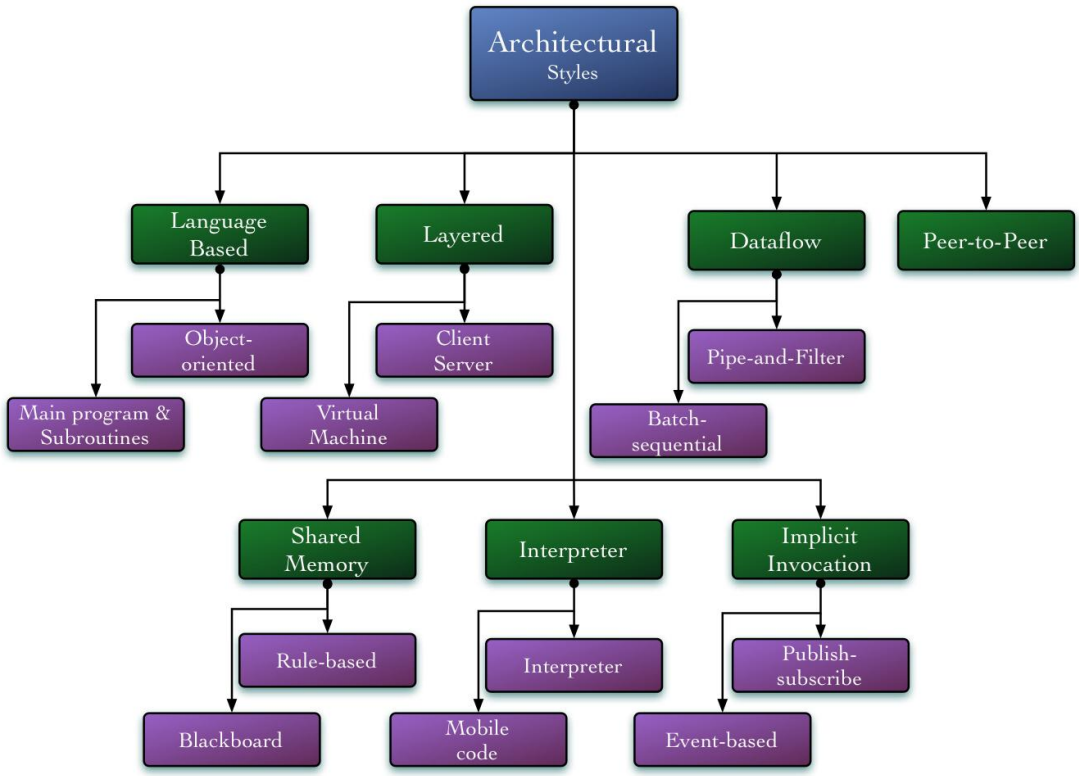


Architectural Styles

An architectural style is a named collection of good architectural design decisions applicable to a recurring design problem (based on experience). This family of architectures are constrained by component/connector vocabulary, topology, and semantic constraints

A good architectural style results in a consistent set of principled techniques, is resilient in the face of changes, it the source of guidance through product lifetime and allows the reuse of established engineering knowledge.

Systems in practice deviate from pure styles, and feature many different architectural styles.



Language-based

- Influenced by the languages that implement them
- Lower-level, very flexible
- Often combined with other styles for scalability
- Examples: main and subroutine, object-oriented

Dataflow

A data flow is a system in which the availability of data controls computation, the design structure is determined by order motion of data

between components and the pattern of data flow is explicit. Examples: batch-sequential (one component has to finish before the next one stars), pipe-and-filter

Pipe and Filter

Data is passed between components, and transformed and filtered along the way

Components: filters

Connectors: pipes

Constraints: filters must be independent entities and should not share state with other filters. Filters do not know about other filters

Layered Systems

Layered systems are hierarchically organized providing services to upper layers and acting as clients for lower layers. Lower levels provide more general functionality to more specific upper layers. In a strict layered system, programs at a given level may only access adjacent layers. Examples: virtual machine, n-tier systems, client-server

Microservices

Monolithic applications put all its functionality into a single process, whereas microservices put each element of functionality into a separate service

Components: services, organized around capabilities.

Connectors: network protocols

Benefits: independently deployable (if a service changes only that one can be deployed instead of deploying the entire application each time), easier to scale, improved fault isolation

Drawbacks: additional complexity of deployment, testing service interaction is more difficult, increased memory consumption

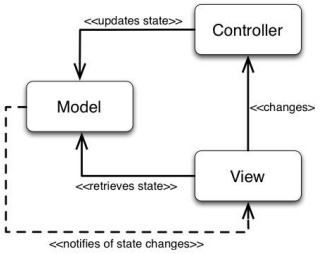
Model View Controller

Goal: decouple models and views -> increase maintainability/testability system and permit new views to be developed

MVC stands for Model View Controller. It is a software design pattern that is typically used for building graphical user interfaces and helps separate the responsibility of the applications State (Model), State Visualization (View) and State Manipulation (Controller). The model contains all of the data within the application and knows all to validate its own integrity and correctness. Models are domain-independent, so they can be easily reused between different applications. Model is the subject in Observer pattern: whenever the model is updated, it notifies the view that its state changed. The view is the observer, and calls model object and asks for its state. It is responsible for rendering the model for the presentation to the user. The view does not store data nor maintain state, and is updated independently from controller or the model. The

controller responds to the changes in the view and updates the model accordingly; it contains the main application logic for the system and binds the view of the model. The model, view, and controller are not to be seen as a single object but rather as a collection of time. Too much functionality is often put in the view, and this functionality should rather be put into the controller which is easier to test, and helps keep the views lightweight and easier to reuse.

Model View Controller



- Model
  - contains application data (often persisted to data store)
  - often the Subjects in the Observer design pattern
- View
  - presents model to user
  - allows user to manipulate data (does not store data)
  - configurable to display different data
- Controller
  - glues model and view together; updates model when user manipulates view
  - houses application logic
  - loose coupling between model and others; view tightly coupled/cohesive with controller

MVC decouples the view from the rest of the system -> benefit: new views are easier to add without changing the other parts of the system.

Drawbacks of MVC: performance issues if there are many objects in the system, deconstruction of a small application into MVC can be overly complicated, it is tempting to make views large (easy to put functionality into view instead of controller)

Model View Presenter

Modernization of MVC which decreases the amount of code that is in the view and forces the model and the present to contain more features. Model is the same as in MVC, it contains the data objects of the system. The model interacts with the Presenter (middle layer) via observer or an event bus (third party component that moderates the interactions between these two different layers). View in MVP is not coupled in the model, in fact no model objects are passed to the view but it instead interacts with the presenter using primitive types (they are easy to create assertions for). Many tests are written for the presenter and the model, but the view is validated only by inspection (by looking at its code). Never sees model objects.

The view only deals with primitive data types because:

- Since the view only sends and receive primitive data types it is hard to put complex logic into it, and this ensures that the presenter is the main location for all program logic
- Views become similar and are validated by inspection since they are just rendering primitive data types

Presenter contains the application logic and mediates all the interactions between model and view. Presenters tend to be large and complex but they are often split up into multiple presenters and can easily be tested with unit testing. View and Presenter interfaces are tightly bound to

each other, this is because each needs to know the public interface of the other to enable communication.

Model View View Model

Considers view and controllers a single unit, The view uses data binding to allow data values to automatically update UI representations.

SOFTWARE QUALITY AND TESTING

Software quality is the degree to which a system, component or process meets the specified requirements /customer or user needs and expectations

Quality Management: includes quality planning, quality assurance, quality control, and quality improvement.

Quality assurance activities are focused on the process used to create the deliverables.

Quality control activities are focused on the product itself.

Process quality influences code quality; code is not the only element of software quality.

Mechanisms to improve code quality: testing, code reviews (formal or informal, good to find defects, especially the early ones), pair programming (good for review and sharing of knowledge), frequent demonstration of working software to stakeholders, code smell detection and refactoring, and many more.

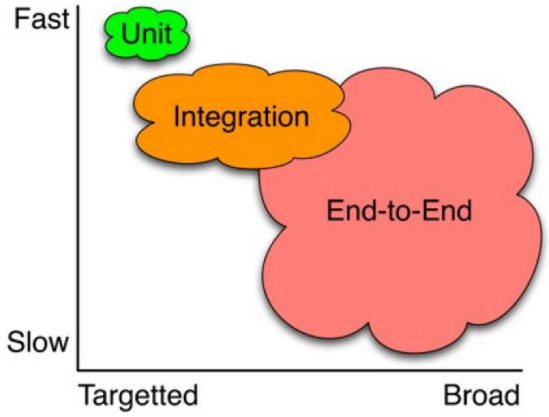
Testing

Verification: is the system being built right? -> discover situations in which the software behavior is incorrect or undesirable

Validation: is the right system being built? -> demonstrate that the software meets its requirements

Testing can show the presence, but not the absence of errors.

Types of Testing: unit, regression, integration, system, acceptance



	Unit	End-toEnd
Fast		
Reliable		
Isolates Failures		
Simulates a Real User		

Unit test

A method of testing that verifies that the individual units of source code are working properly. A unit is the smallest testable part of an application. They are good for ensuring that widely used low-level functionality is correct.

-> process of testing individual components in isolation (units may be: functions / methods, classes, composite components with defined interfaces used to access their functionality). Unit tests are typically written by the developer and are usually fully automatic

To reveal a fault, a test must:

- Reach some code
- Trigger a defect
- Propagate an incorrect result, which must be observed and interpreted as incorrect

Each test is made of a name that indicates the point of the test (just one thing), a setup portion, invocation of the code to be tested and a check of the results against expectations (usually through assertions)

Test Suite: contains the collection of tests for one project, sometimes broken down by component

Regression test: testing the system to check that changes have not broken previously working code. They are not new tests, but instead a repetition of the existing tests -> do not write throwaway tests

Integration test: exercise groups of components to ensure that their contained units interact correctly together. They touch larger parts of

the system (major subsystems). Since these tests validate many different units at the same time, identifying the root cause of a test failure can be difficult. Good for ensuring that important tasks that users will perform are correct.

System Test: test large parts of the system, or whole system. It involves integrating components to create a version of the system

Acceptance Test: formal testing with respect to user needs and requirements to determine whether the software satisfies the acceptance criteria

Code Coverage

Intuition: the more parts are executed the higher is the chance that defects are uncovered. Code coverage measures the proportion of the system that is executed by the test suite.

Types of coverage criteria (all measured in percentage):

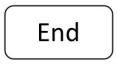
- Line coverage: measures the number of lines executed by the test cases divided by the total number of lines present in the code
- Statement coverage: measures the number of statements in the code that have been executed at least once during testing. Each line of code that can be executed i s considered a statement. If a line of code is var x = 10; console.log(x); it has 2 statements in one line. Statement coverage and line coverage are often the same.
- Branch coverage (each condition): number of branches that have been executed at least once. Two different branches are created when there are decision points (if else, switch statements, while

loops). Branch coverage looks if you took true and false branches for each conditional for the written tests. There are twice as many branches as conditionals. Ternary operators have 2 branches but one statement.

- Path coverage (each possible path through the code): number of executed paths (can be unlimited, e.g., while loops)
- Multiple condition coverage (MCC): ensures that all statements are independently evaluated with respect to the outcome of a function (hard to use)

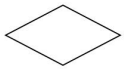
Control Flow Graphs

Indicates the start of the control-flow



Indicates the end of the control-flow

Indicates a processing step, text in the box is the code to execute

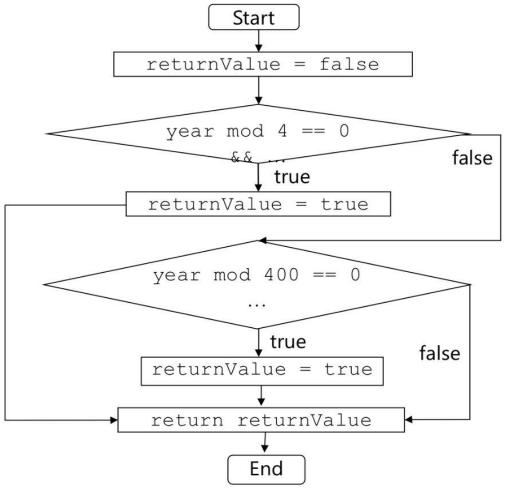


Indicates a conditional representing a yes/no question or true/false test. Two arrows emanate (one for yes/true and one for no/false) and must be labeled. The yes/true arrow typically comes out the bottom and the other out one of the sides.

Example

```
boolean isALeapYear( int year ) {
   // Declare a variable for the return value
   // of the function
   boolean returnValue = false;

   // If the year is divisible by 4 and
   // not by 100 it is a leap year
   if ( ( year mod 4 == 0 ) &&
           ( year mod 100 != 0 ) )
      returnValue = true;
   else if ( year mod 400 == 0 )
      returnValue = true;
    return returnValue;
}
```



Branch Coverage

`int eval(int x, boolean c1, boolean c2){`
`if (c1)`
`x++;`
`if (c2)`
`x--;`
`return x;`
`}`

Test Suite

```
eval(0, true, true)
eval(0, false, false)
```

100%

Path Coverage

```
int eval(int x, boolean c1, boolean c2){
    if (c1)
        x++;
    if (c2)
        x--;
    return x;
}
```

Test Suite

```
eval(0, true, true)
eval(0, false, false)
eval(0, true, false)
eval(0, false, true)
```

100%

Multiple Condition Coverage (MCC): all combinations of conditions inside each decision are tested (i.e., all combinations of true and false for the conditions in the decision have to be tested)

Advantages of Code Coverage: actionable, cheap (except path coverage that can become infeasible e.g., while loop), intuitive.

Coverage as stopping criteria is good, but smart testing is always better

White Box Testing: tests based on code, examines system internals, covers as much implemented behavior as possible. Test criteria: cover execution paths

Black Box Testing: tests based on specification, treats system as atomic, covers as much specified behavior as possible. Test criteria: cover input space

Equivalence class partitioning:

- Convert a continuous input (output) space into something manageable. ECF involves dividing input into equivalence classes

based on their behavior and attributes. The goal is to reduce the number of test cases required to test a software system while ensuring that each test case is effective in revealing faults.

- Identify / guess which types of inputs are likely to be processed in a in a similar way
- Each test should exercise one and only one equivalence partition

Example

- System asks for numbers between 100 and 999
- Equivalence partitions:
  - Less than 100
  - Between 100 and 999
  - More than 999
- Three tests:
  - 50, 500, 1500

Boundary testing: tests inputs starting from known good values and progressing through reasonable but invalid to known extreme and invalid (e.g., max / min, just inside / outside boundaries, typical values and error values). It's like ECP but looks specifically at edge cases.

Mutation Testing: has the goal to ensure that a test suite is able to detect faults (test suite quality). Involves modifying a program in small ways and examining if the test suite is able to find errors or mutants. It tries to simulate the real human behavior of making small mistakes. Effectiveness of the test suite is determined by the percentage of mutants killed.

Pros: correctness focus, programmatic oracle

Cons: synthetic, time to execute is low

Quiz:

- In Red Green Refactor, we first write the failing test case, then we refactor and extend the code, and finally we create the implementation to make the test pass -> False, refactoring is done later, not in between
- If you want to ensure that the application satisfies your customer's expectations, acceptance tests are the best option -> True, acceptance tests are often run by software developer (QA) itself but it can also be done with customers as well. Unit tests are about testing correctness and do not have to do with customers.
- System tests typically run faster than unit tests -> False, system tests are bigger, so they run slower
- It is generally better to use a monolithic architecture instead of a microservice architecture because microservices tend to have low cohesion and high coupling -> False, microservices usually have high cohesion and low coupling.
- Suppose you have a simple web blogging application that consists of the blog post data (e.g., title, author, and content), which is then represented in a web interface in the browser. You decided to apply the MVP pattern. The blog post data represents the model in the MVP pattern -> True.

CODE SMELLS AND REFACTORING

Every module has three functions: to execute according to purpose, to afford change, and to communicate to its readers. If it does not do one or more of these, it is broken.

Code smell: a recognisable indicator that something structural may be wrong in the code. It does not imply bugs or that code does not prevent code from running. Can occur in the product code as well as the test code. Code smells indicate weaknesses in design that may be slowing down development or increasing the risk of bugs or failure in the future. Bad code smells can be an indicator of factors that contribute to technical debt.

Within-class smells:	Between-class smells:
• Duplicated Code • Long Parameter List • Long Method / Large Class • Massive Switch Statements • Magic Numbers • Comments (or lack thereof)	• Duplicated Code • Divergent Change • Message Chains • Shotgun Surgery

Bloaters: size of program elements make them hard to understand / change. Examples: long methods, large classes, long parameter lists

OO Abusers: failure to leverage OO design. Examples: switch statements, refused bequest

Change Preventers: make it harder to evolve code. Examples: divergent changes, shotgun surgery
Dispensables: unnecessary complexity. Examples: duplicate code, dead code, speculative generality
Couplers: unnecessary coupling. Examples: feature envy, middle man
Duplicated code is bad because you modify one instance of duplicated code but not the others (not all versions fixed). Long parameter lists are hard to understand and become inconsistent and difficult to use. Long methods and large classes are most difficult to understand. Switch statements are a problem of duplication. The problem of magic numbers is that their meaning will be forgotten (for example a number that has not a variable assigned to it). Comments are not necessarily bad themselves but may indicate areas where the code is not as clear as it could be -> do not write comments for things that do not need them, it is not good to write too many comments. Divergent change occurs when making changes to a class, and you have to change many unrelated method (dependency issue). Message chains occur when a client asks an object for another object and then asks that object for another object etc. It's bad because the client depends on the structure of the navigation. Shotgun surgery is present if a change in one class requires cascading changes in several related classes -> consider refactoring so that the changes are limited to a single class
Refactoring: is a predictably meaning-preserving code transformation. A small, behavior-preserving, source to source transformation. It does not

involve bug fixing, nor adding new features, but just for the ease of better understanding

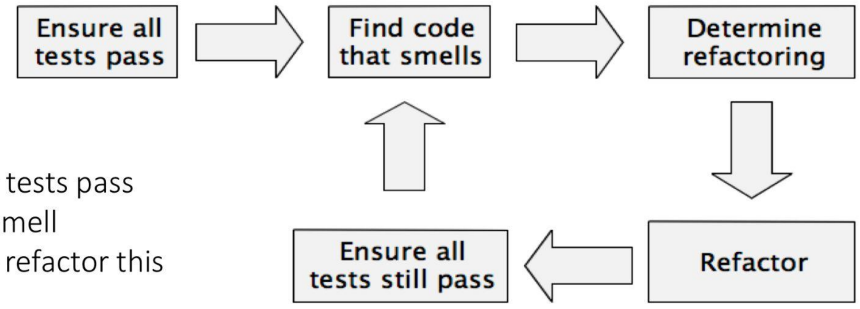
When to refactor? If it appears more than three times, or affects the ability of making a change or the code is no longer understandable

When not to Refactor:

- 2 weeks every 6 months
- After the main functionalities are implemented
- When tests are failing
- When you should just rewrite the code
- When you have impending deadlines

Code Linting: it is the automated checking of your source code for programmatic and stylistic errors using a Linter

How to Refactor:



1. Make sure all your tests pass
2. Identify the code smell
3. Determine how to refactor this code
4. Apply the refactoring
5. Run tests to make sure you didn't break anything
6. Repeat until the smell is gone

Without tests you don't know whether refactoring is correct or not -> write tests before you refactor

Rule of three: code the feature, code it but take note, refactor code first

Refactoring – Extract Method: pull code out into a separate method when the original method is long or complex. Name the new method to make the method cleared, and follow the SRP.

Refactoring – Extract Class: one class is doing work that should be done by two classes -> solution: create a new class and move the relevant fields and methods from the old class into the new class

Refactoring – Pull Up Method: if there are identical methods in more than one subclass, move the method to the superclass

Refactoring – Pull Up Field: if there are identical fields in more than one subclass, move the field to the superclass

Refactoring – Extract Subclass: if a class has features that are used only in some instances, you can create a subclass for that subset of features

Refactoring – Extract Interface: if several clients use the same subset of a class's API / interface, or two classes have part of their interfaces in common you can extract the subset into an interface.

Refactoring – Replace Data Value with Object: if you have some data that needs additional data or behavior, you can turn the data into an object.

Refactoring – Hide Delegate: if a client is calling a delegate class of an object, you can create methods on the object to hide the delegate

Quiz:

- One way to reduce the duplicated code smell is to create a method containing the duplicate code and then call it in places where the duplicate code previously was -> True

- Switch statements are a good example of the OO Abuser smell since they often fail to leverage OO design -> True

You find the following piece of code in your codebase that calculates the n-th fibonacci number in the fibonacci sequence. Which code smells are exhibited? Select all that apply.

```
// Calculates the n'th fibonacci number recursively
// Does so by first checking whether the input number "n" is 0 or 1 (if it is, fibonacci number is just
the number itself)
// Otherwise returns the sum of the two previous fibonacci numbers "n-1" and "n-2".
public static long fibonacciAtIndex(long index) {
   if ((index == 0) || (index == 1))
      return index;
   return fib(index - 1) + fib(index - 2);
}
```

- Explanatory comments
- Literal values
- Large class

SOFTWARE EVOLUTION

Software maintenance / evolution is the process of producing new (versions of) software under the constraints of the existing software -> similar to development, but harder.

Software change is inevitable:

- New requirements emerge when software is used
- The business environment changes
- Errors must be repaired
- New computers and equipment is added to the system

A key problem of organizations is implementing and managing changes to to their existing software systems

Evolution is hard: systems are not robust under chance. There is often a poor documentation of code, of design process and of system evolution

Reasons for Evolutionary change:

- Corrective: correct faults in the system behavior -> caused by errors in coding, design or requirements
- Adaptive: due to changes in operating environment (e.g., different hardware or operating system)
- Perfective: due to changes in requirements, often triggered by organizational, business, or user learning

API Contract Compatibility

Method pre-conditions	Strengthen	Breaks compatibility for callers
	Weaken	Contract compatible for callers
Method post-conditions	Strengthen	Contract compatible for callers
	Weaken	Breaks compatibility for callers

A change is binary compatible with pre existing binaries if pre existing binaries that previously linked without error will continue to link without error

Centralized version control system: commits become available to everyone else on the project immediately

Centralized version control system: commits are pushed to some central copy, and then are pulled by others to have their copy locally.

Issue trackers: used to track issues that are present in the system. Useful to know what the problem is, how important it is, who is responsible for fixing it, and who caused it. Example: Git.
Issue tracking systems maintain a list of bugs in the software, in a database called the bug repository. Responsibility is assigned for each bug, feature, or task.
A bug report is made of a title (summary), description, status, assignee, priority, target milestone, and comments. A good summary (title) should quickly and uniquely identify a bug report, explain the problem and not the suggested solution. Examples: good: canceling a file copy dialog crashes file manager, bad: software crashes or browser should work with my website.
A good description should include enough context, overview, steps to reproduce, actual results and expected results.
Bug report comments have the role of discussion among reporters and developers. They provide a way of documenting the history of the development process and the rational (e.g., why it was decided to fix a bug in a certain way)

Working on a change task

Problems / Difficulties: too much code to understand and read, language mismatch (bug reports (NL) != source code). To localize and edit relevant code identify good search terms, take advantage of tool support, and take advantage of information provided in bug reports (such as stack traces, and on Q&A forums like StackOverflow)

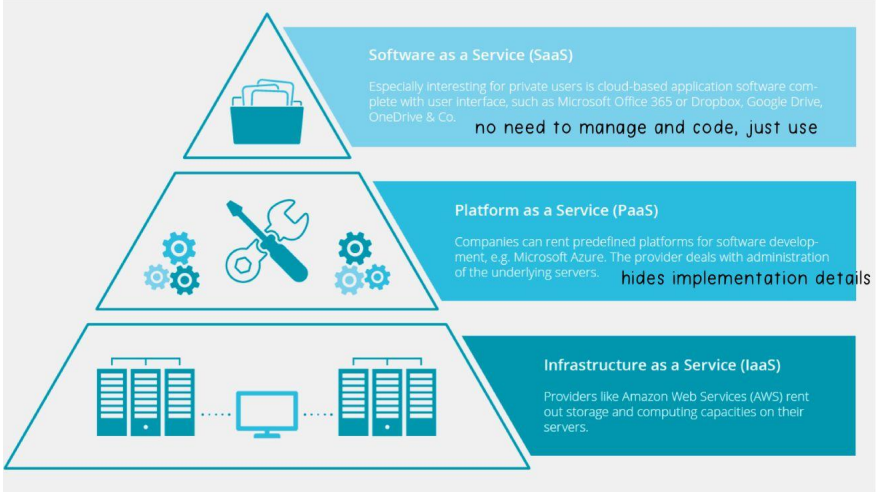
Quiz:

- Both adding and deleting a package that contains classes and methods that are part of the API are binary compatible -> False, deletion is not binary compatible
- When evolving API methods, changing a method name breaks the compatibility -> True
- Operations on non-API packages (adding, deleting) are all binary compatible -> True
- Git is a centralized version control system where users make commits to a single repository that become immediately available to everyone else working on the project -> false, it's decentralized, and to have commits stored locally one has to send pull request to repository first
- Version control systems are tools that facilitate collaboration among developers working on a project, and track and manage changes to source code over time.

CLOUD DEVELOPMENT CI/CD

Cloud Computing: model for enabling convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction -> promotes availability

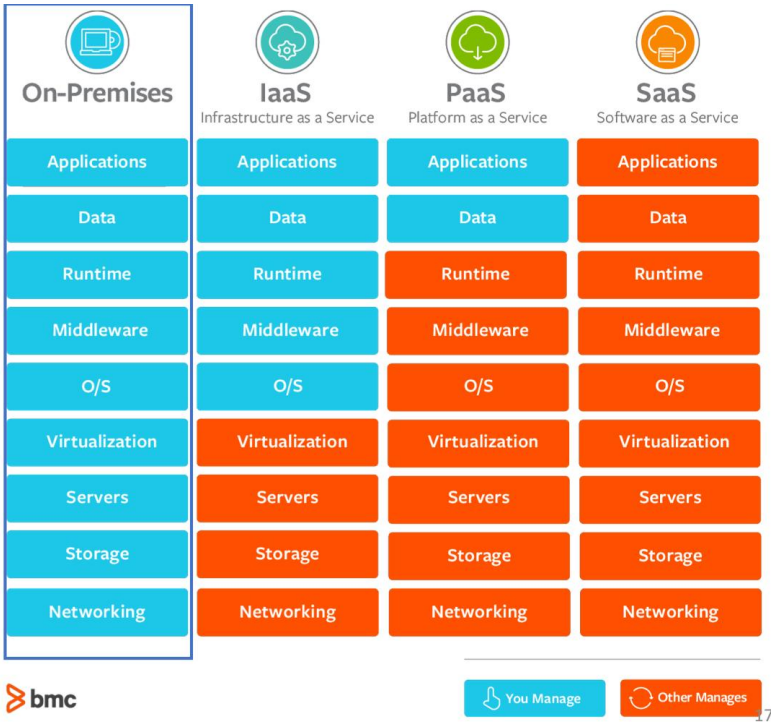
IaaS vs PaaS vs Saas



- SaaS - End-users
  - Example: AWS Workmail
- PaaS – Software Developers
  - Example: AWS Elastic Beanstalk
- IaaS - IT Administrators
  - Example: AWS EC2

On-Premise

- Buy a Server / barebone PC
- Add RAM & SSD/HDD
- Add VMware/docker
- Install OS
- Setup middleware
- Setup runtime
- Setup application
- Connect database
- Configure network routing
- Configure network firewall



Source: https://kinsta.com/de/blog/arten-von-cloud-computing/

Waterfall: disconnection between the different stages

Agile: disconnection between deployment and operating the application (operating: fix problems, make sure it scales; deploy: pushing to cloud and making sure that it works there)

Continuous Integration (CI): a practice where developers automatically build, test, and analyze each software change in response to every software change committed to the source repository. Tests are run to ensure that the changes didn't break the application. It's the solution to the problem of having too many branches at once that might with each other.

Principles of CI: frequent commits, only commit code that compiles, fix broken builds immediately, only pull code from known good configurations, all tests must pass.

Benefits of CI: detect and fix problems faster, measurable software health, developers trust their changes (global failure feedback).

Risks of not doing CI: inability to generate testable build, late defect discovery, low quality awareness, long integration phases, fear of making changes (e.g., refactoring), little confidence that tests will pass after integration

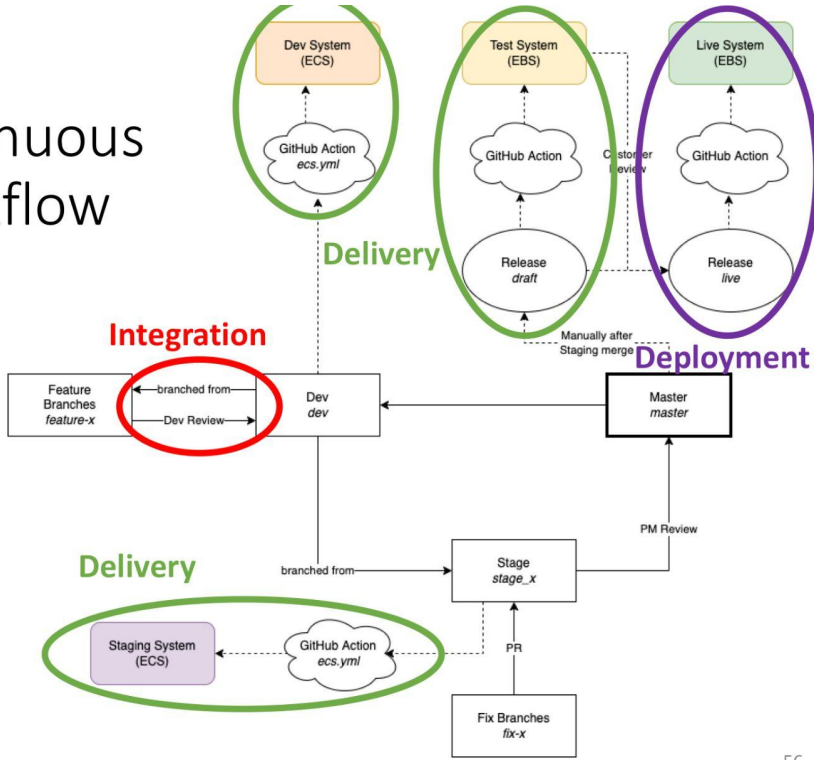
Continuous Delivery: a practice that ensures that a software change can be delivered and is ready for use by a customer in production-like environments. The purpose of continuous delivery is to ensure that it takes minimal effort to deploy new code. -> automatically release to repository

Benefits of CD: can deploy at any time (e.g., security fixes), even better feedback and more confidence

Continuous Deployment: a practice where incremental software changes are automatically tested, evaluated, and deployed to production environments -> automatically deploy to production

Example for Continuous Deployment Workflow

- Small test-deployments for long-lived branches (dev) and staging branches
- Manual trigger for test system deployment (customer facing)
- Manual trigger for live system deployment (end-user facing)



Principles of low-risk releases

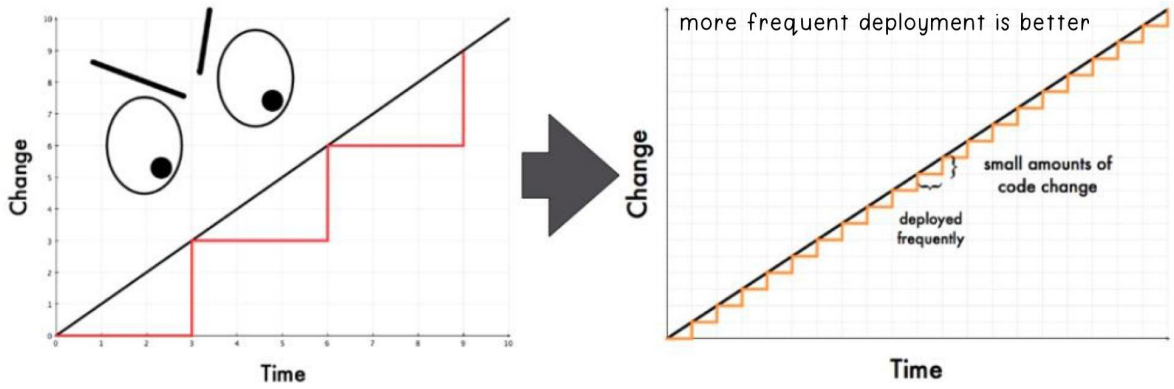
1) Favor incremental changes:

   - Green/Blue Deploy -> two production environments, big flip (router that manages user migration). The fundamental idea behind Blue-Green-Deployment is to have two environments that one can switch between, where one is always live and the other is used for testing new features

Benefits of Green/Blue Deploy: incremental, stress- free deployment, preparation of blue environment, supports rapid rollback (since it can easily be switched between the blue and green environment), minimized down-time. But does not make it easier to deploy one feature at a time instead of entire application versions.
Drawbacks of Green/Blue Deploy: extra cost in infrastructure overhead, data migration issues (switch cost might be high if migration is slow and not mirrored), database is still not handled
- Canary Releasing: change is only seen by employees, and then by the rest of users later on. If it fails the system is being corrected otherwise it gets expanded to more users.
Benefits of Canary Releasing: rolling back is easy, stop routing users, can combine with A/B testing, can check if capacity requirements can be met by gradually increasing the load
Drawbacks of Canary Releasing: any shared resource (cache, services) need to work with all versions in production. May be supporting multiple versions in production

2) Decouple deployment from releasing:
    - Deployment is what happens when you install some version of your software into a particular environment (often the production environment)
    - Releasing is when a system or some part of it (for example, a feature) is made available to its users

3) Focus on reducing batch size: deploy new features to users quickly, enable responsive defect resolution, smaller delta = smaller faults



4) Optimize for resilience: the ability to restore your system to a baseline state in a predictable time is vital not just when a deployment goes wrong, but also as part of your disaster-recovery strategy

Quiz:

- Building the application and running tests can be automated but deployment cannot be automated -> False, deployment can be automated
- Continuous Delivery is the practice of building and deploying software regularly, bringing updates faster to the end-customer -> False, delivering cannot be seen by the end-customer, just by the internal system

USER INTERFACE DESIGN, SUABILITY, AND BIAS

Good design is a lot harder to notice than poor design, because good designs fit humans' needs so well that the design becomes invisible.

Why (UI) design is important:

- Many human errors and machine misuse are in reality design errors
- Designers help things work by providing a good conceptual model
- Designers decide on a range of users as the design audience
- Good design avoid wasting time of users

Design is not just what it looks and feels like, but it is also how it works.

User-Centered Design: process of designing a tool from the perspective of how it will be understood and used by a human user. It's cost saving, useful for user expectations. Consists of abilities, memory, color, ergonomics.

Usability: the extent to which a product can be used by specified users to achieve specific goals with effectiveness, efficiency and satisfaction in a specified context of use. It's closely related to good UI design and part of the ussr experience. Usability is not utility / functionality. Utility is how useful something is, functionality is concerned with functions/features of the product. Usability is how easy it is to use. There can be products that are usable but have no utility (a good product / software should have both)

Usability Methods:

- Prototyping: involves creating a scaled-down or incomplete version of a system to demonstrate or test aspects of it. Prototyping aids UI design, provided a basis for testing, and allows interaction with the user to ensure satisfaction

- **Surveys:** used to collect quantitative data, helps to find information on who the users are, how they use the product and what their needs and opinions are. Questions in surveys should be well-planned (usually no follow-up questions possible) and surveys should be short.
- **Contextual inquiry / task analysis:** involves interviews with people in their natural work environment. It helps to understand the task procedures that users follow to reach their goals.
- **Cognitive walkthrough:**

Group of usability experts *walk* through a set of typical user tasks, one-step-at-a-time

At each step, answer 4 questions, e.g.

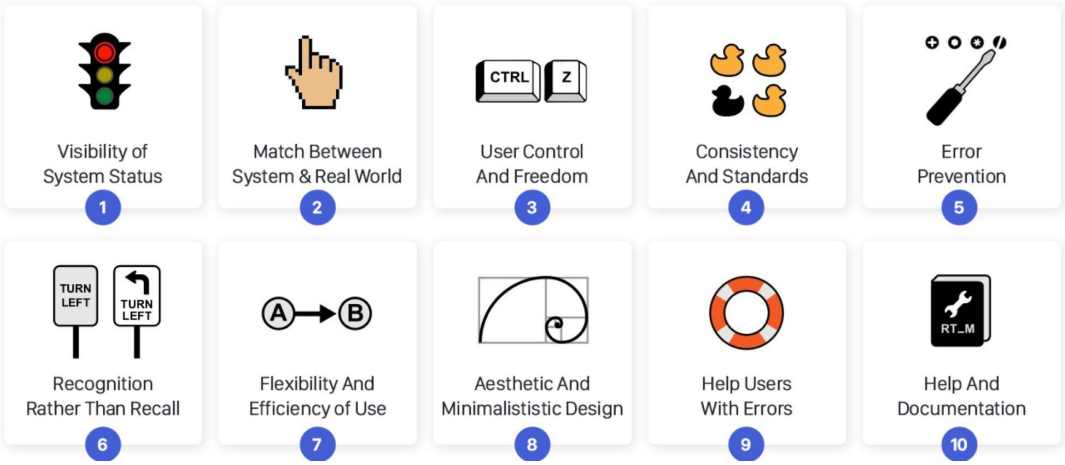
- Will the user try to achieve the right effect?
- Will the user notice that the correct action is available?
- Will the user associate the correct action with the effect to be achieved?
- If the correct action is performed, will the user see that progress is being made toward solution of the task?

Define tasks and sub-tasks to be analyzed

Establish persona: who is the user of the system, specify experience or technical knowledge for persona

e.g., very computer-savvy teenage kid / senior without tech experience

- **Heuristic evaluation (Nielsen's 10 Principles):**



Visibility of system status: keep users informed about what is going on, through appropriate feedback within reasonable time

Match the real world: the system should speak the users' language, with phrases and concepts familiar to them, rather than system-oriented terms. The system should follow real-world conventions and make the information appear in natural and logical order.

User Control and Freedom: provide an emergency exit without having to go through extended dialogue, support undo and redo operations.

Consistency and Standards: users should not have to wonder whether different words, situations, or actions mean the same thing (follow conventions)

Error Prevention: preventing errors is better than good error messages -> eliminate error-prone conditions or check and present users with a confirmation option before executing irreversible actions (e.g., deleting account or discarding changes)

Recognition rather than recall: minimize user's memory load by making objects actions, and options visible

Flexibility and efficiency of use: provide accelerators (shortcuts)

Aesthetic and minimalist design: dialogues should not contain information which is irrelevant or rarely needed

Help users recognize, diagnose, and recover from errors: error messages in plain language (no HTTP status codes), precise and constructive

Help users recognize, diagnose, and recover from errors: help should be searchable, focuses on user's task, concrete and short

Usability methods should be applied early and often, prototype and test often (before and during development, best to have an iterative testing cycle instead of a one-off phase)

Cognitive Bias: a systematic pattern of deviation from norm or rationality in judgment

Cognitive Biases:

- Affinity Bias: describes people's tendency to favor people that share similar interests with us
- Anchor Bias: describes people's tendency to rely too heavily on the first piece of information they receive
- Confirmation Bias: describes people's tendency to seek for information that is consistent with out beliefs

- Hyperbolic Discounting: describes people's tendency to choose smaller rewards rather than delayed gratification
- Negativity Bias: describes people's tendency to remember negative experiences better, because they tend to affect people much more than the positive ones

Biases with respect to gender can also lead to issues in design (gender not binary, but most research focuses on male/female). Example: smartphones too big for most women's hands -> GenderMag

K-anonymity: an individual in a database is linked to k other entries when linked to other datasets.

Quiz:

- A website asks for the phone number in international standard, and restricts the country code to two digits. THis restriction on the number of digits for the county code is a good example of the effectiveness principle in UX and specifically usability -> True
- An address form on a website asks for the postal code of a Swiss address. Restricting the input to the postal code field of the form to 4 digits is a good example of the error tolerance principle in UX and specifically usability -> False: not about error prevention, but rather effectiveness.
- Showing a user that the phone number they inputted has an invalid format by marking the according input field in red color after the user fills it out is a good example for the error prevention principle -> True

- Anonymising data helps keeping the utility of the data while minimizing the risk of disclosing private user information -> True

- To enhance the level of K-anonymity, one option is to consider omitting the specific birth date and instead disclose only the birth year -> True.
-