Konstantin Kuchenmeister

Einführung in die Softwaretechnik

Summary-UML-Pattern

TUM-Wirtschaftsinformatik

Gliederung

Vorlesung 1: Einführung

→The impossible trident.

Software development is more than just writing code:

- It is problem solving
  - Understanding the problem
  - Proposing a solution and plan →prototype
  - Engineering a system based on the proposed solution
- It is about dealing with complexity
  - Creating abstractions and models
  - Notations for abstractions: is a graphical or textual set of rules for representing a model
- It is about dealing with change
  - Requirements elicitation, analysis, design, implementation, validation of the system, delivery and maintenance.

Abstractions:

- Complex systems are too hard to understand →human memory can only store 7|+-2 pieces
- Chunking: Group collection of objects to reduce complexity →phone number
- allows to ignore unessential details
- Def. 1: Abstraction is a thought process where ideas are distanced from objects→activity
- Def. 2: Abstraction is the result of a thought process →Abstraction as entitiy
  → Abstractions can be expressed with a model.

Models: A model is an abstraction of a system.

- Object model: What is the structure of the system?
- Functional model: What are functions of the system?
- Dynamic model: How does the system reacts to external events?
- System model: object model + functional model + dynamic model

Why is software development difficult?

- The problem is usually ambiguous
- Requirements are usually unclear and change when they become clearer
- The problem, application and solution domains are complex
- The development process is difficult to manage
- Software is a discrete system→hidden surprises (David Lorge Parnas)

Software engineering: a problem solving activity:

- Analysis: Understand the nature of the problem and break the problem into pieces
- Synthesis: Put the pieces together into a larger structure

What one uses for problem solving:

1. Techniques: Formal procedures for producing results using some well defined notation → algorithms
2. Methodologies: Collection of techniques applied across software development and unified by a philosophical approach.→Functional Decomposition, O-O Analysis and Design
3. Tools: Instruments or automated systems to accomplish a technique → Compiler, Editor

Software engineering definition:

Software engineering is a collection of techniques, methodologies and tools that help with the production of a high quality software system developed with a given budget before a given deadline while change occurs.

Dealing with complexity:

- Modeling, Notations (UML)
- Requirements Elicitation
- Analysis and Design
- Implementation
- Testing

Dealing with Change:

- Release Management → Software Configuration Management
- Delivery → Continuous Delivery
- Software Life Cycle Modeling
- Rationale Management
- Project Management

Phenomenon: An object in the world of a domain as you perceive it → real world object (watch)
Concept: Describes the common properties of phenomena → all watches

- Is a 3 tuple
  - Name: distinguishes the concept from other concepts
  - Purpose: Properties that determine if a phenomenon is a member of concept
  - Members: The set of phenomena which are part of the concept.

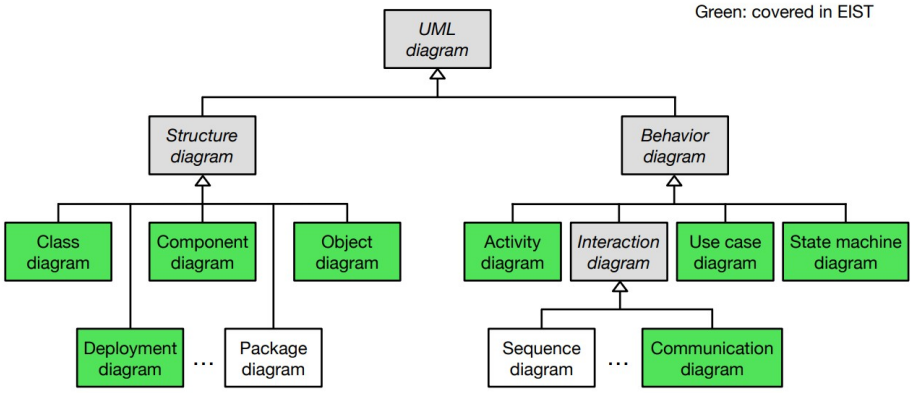
Abstraction: Classification of phenomena into concepts
Modeling: Development of abstractions to answer specific questions about a set of phenomena while ignoring irrelevant details

Systems

- A system is a organized set of communicating parts
  - Natural system: A system whose ultimate purpose is not known
  - Engineered system: A system which is designed by engineers for a specific purpose
- The parts of the system can be considered a system again → subsystems

Models: A model is an abstraction describing a system
View: A view depicts certain aspects of a model
Notation: A set of graphical or textual rules for depicting models and views
→ Informal notations (“napkin design”)
→ Formal notations (UML)

Overview of UML Diagrams



Vorlesung 2: Model-Based Software Engineering

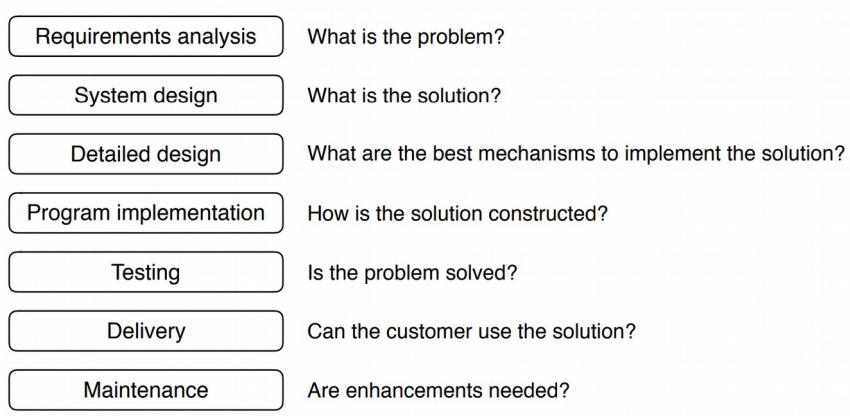
Software lifecycle:

- Set of activities and their relationships to each other to support the development of a software system.
- Activities: Requirements elicitation, Analysis, System Design, Implementation, Testing, Delivery
- Relationships: Testing must be done before implementation

Software lifecycle model:

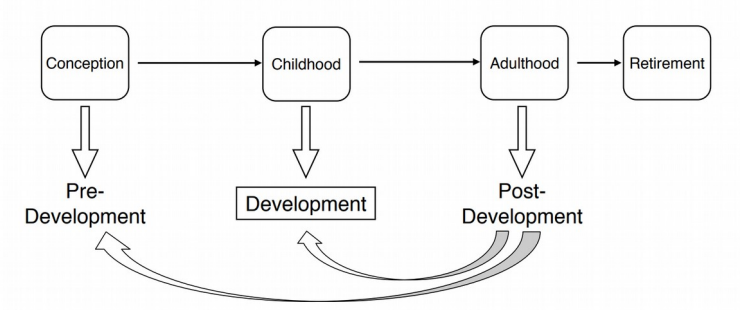
- An abstraction representing the development of software for the purpose of understanding, monitoring, or controlling the development of software.

Software Development Activities:



1. Requirements elicitation: the client and developers define the purpose of the system.
    1. Functional requirements
    2. Nonfunctional requirements → use case diagram
2. Analysis: developers aim to produce a model of the system that is correct, complete, consistent and unambiguous.
    1. Transform the use cases into an object model that completely describes the system.
    2. Class diagram
    3. Dynamic Model: → state machine diagram, → sequence diagram
3. System design: developers define the design goals of the project and decompose the system into smaller subsystems that can be realized by individual teams.
    1. Subsystem decomposition
    2. system design object model
    3. design goals
4. Object design: developers define solution domain objects to bridge the gap between the analysis model and the hardware/software platform defined during system design
    1. Object design model → class diagram
    2. more detailed design
5. Implementation: developers translate the solution domain model into source code.
6. Testing: developers find differences between the system its models by executing the system with sample input data sets.

Software lifecycle:



Tailoring:
There is no one size fits all software lifecycle model that works for all possible software engineering projects. → adjusting a lifecycle model to fit a project

- Naming: adjusting the naming of activities
- Cutting: removing activities not needed in the project
- Ordering: defining the order the activities take place in

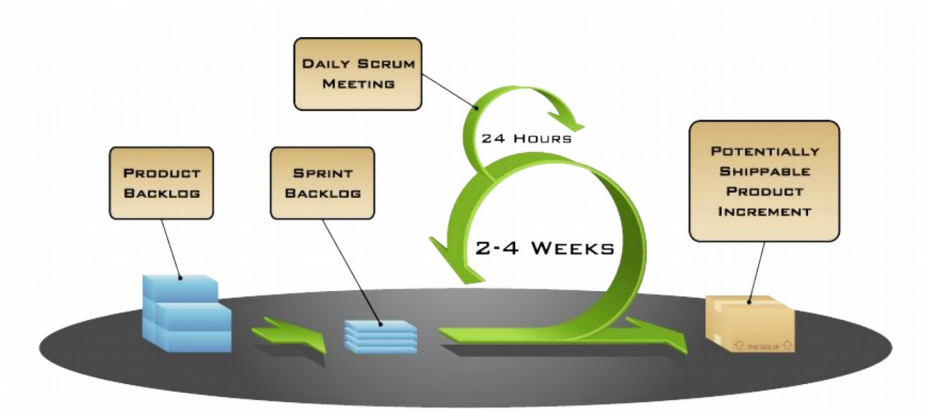
Controlling software development with a process:

1. Through organizational maturity: Repeatable process, CMMI
2. Through agility: Large parts of software development is empirical in nature

Defined vs. empirical process:

1. Defined: Planned, follows strict rules, avoids deviations
2. Empirical process: Not entirely planned, inspect and adapt

SCRUM: Example of an empirical process control model



Problem Statement:

- A description of the problem addressed by the system
- Synonym: Statement of work
- A problem statement describes:
  - current situation, functionality of new system, environment of new system, deliverables expected by client, delivery dates, set of acceptance criteria

Unified Modeling Language Overview:

- reduces complexity by focusing on abstractions
- can be seen as high level programming language → generation of source code
- Communication: provides a common vocabulary for informal communication
- Analysis and design: UML models enable developers to specify a future system
- Archival: UML models provide a way for storing the design of an existing system

Application domain: The environment in which the system is operating: represents all aspect of the user's problem. This includes the physical environment, the users and the other people, their work processes and so on. → Changes over time as work changes.

Solution domain: The technologies used to build the system. Is the modeling space of all possible systems. Modeled in more detail than the application domain.

→ Use case diagrams (UML-Zusammenfassung)
→ Class diagrams (UML-Zusammenfassung)

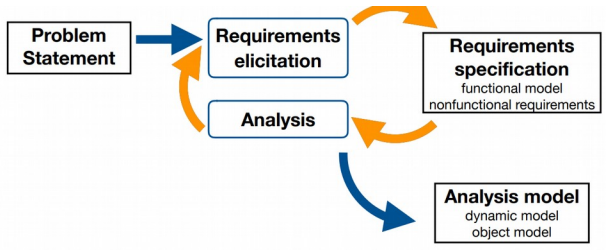
Analysis: finding application domain objects

- Syntactical investigation with Abbot's technique (creating class diagrams in natural language)
  - Flow of events in use cases
  - Problem statement form the customer
- Other knowledge sources:
  - Application knowledge: End users and application domain experts know the abstractions of the application domain
  - General world knowledge: You can also use generic knowledge and intuition
- During system design and object design we use another knowledge source:
  - Solution knowledge: Solution domain experts know abstractions in the solution domain

Vorlesung 3: Requirements Analysis

Requirements engineering:

1. Requirements elicitation (is a development activity)
   1. Definition of the system in term understood by a customer or user → Requirements specification
2. Analysis:
   1. Definition of the system in terms understood by a developer → Analysis model
3. Requirements engineering
   1. Combination of the two activities requirements elicitation and analysis
   2. defines the requirements of the system under construction



Activities during requirements elicitation:

- Identify actors: different types of users
- Identify scenarios: a set of detailed descriptions in natural language
- Derive use cases: generalize and abstract the scenarios to represent the functionality
- Refine use cases: detail each use case and describe behavior of the system
- Identify relationships among use cases to allow for better reuse
- Identify nonfunctional requirements to agree on aspect of system not related to functionality

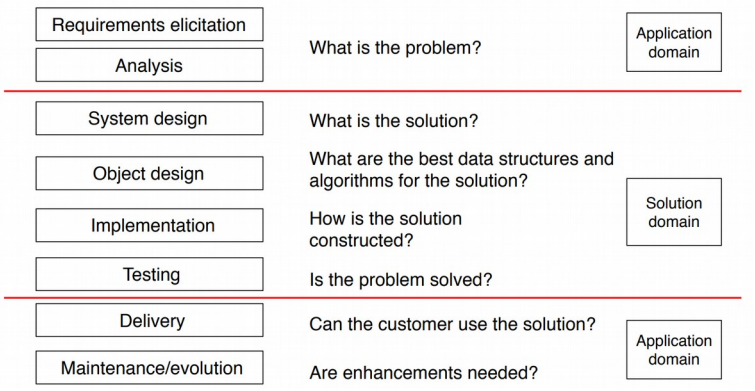
Requirements:

- A feature that the system must have or a constraint it must satisfy to be accepted by client
  - describe the user's view of the system
  - identify what of the system, not how

Part of requirements	Not part of requirements
• Functionality • User interaction • Error handling • Environmental conditions (interfaces)	• System design • Implementation technology • Development methodology

Difficulties: Requirements eliciation

1. How can be identify the purpose of a system
2. How can we identify the system boundary

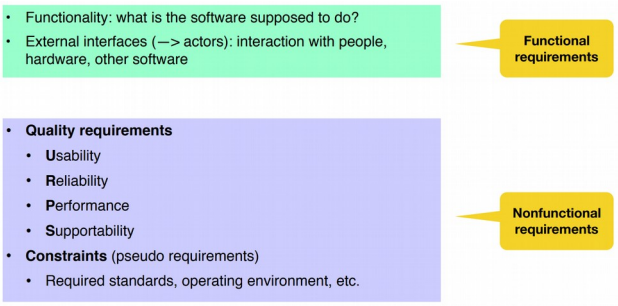


Types of requirements elicitation: (should all start with problem statement)
Greenfield engineering: Development from scratch, no prior exists, requirements are extracted from client and users, triggered by users needs
Re-engineering: Re-design or re-implementation of an existing system, triggered by new technology
Interface Engineering: Provide services of existing system in new environment , triggered by new market needs || technology

Functional requirements: (What is the software supposed to do?)
Includes:
- Relationships of outputs to inputs, response to abnormal situations
- Exact sequence of operations
- Validity check on the inputs

→ Should be phrased as a verb or action

Nonfunctional Requirements: (URPS)
- **U**sability
- **R**eliability: Robustness, Safety
- **P**eformance: Response time, Availability…
- **S**upportability: Adaptability, Maintainability



Performance requirements:
- Number of simultaneous users supported
- Amount of information handled
- Number of transactions processed within certain time periods

→ Usability: The ease of which actors can perform a system function
→ Robustness: The ability of a system to maintain a function, when user enters a wrong input or when there are changes in the environment
→ Availability: The ratio of expected uptime of a system to sum of expected downtime
→ Adaptability: The ability of the system to adapt to itself to changed cirucmstances
→ Maintainability: The ease with which a system can be modified by a developer to correct defects, deal with new requirements or cope with a changed environment.
→ Safety: Protection against unwanted incidents.
→ Security: Protection against intended incidents.

Constraints (pseudo requirements):
- Compliance to standards: report format, audit tracing
- Implementation requirements: tools, programming language
- Operations requirements: Administration and management of the system
- Legal requirements: Licensing, regulation, certification

Constraint definitions:
- Packaging requirements: Constraints on the actual delivery of the system
- Interface requirements: Constraints imposed by external systems
- Legal requirements: The system must comply with federal law.

Model verification: an equivalence check between two models, one of the generated from the other one
Model validation: is the comparison of the model with reality.

Techniques to describe requirements: (bridging the gap between end users and developers)
1. Scenario: describes the use of the system as a series of interactions between a specific end user and the system → very specific, includes names, numbers and instances
    1. Describes a single instance of a use case
2. Use case: describes a set of scenarios of a generic end user, called actor, interacting with the system
    1. Describes all possible instances and is more generic and more abstract
3. User story: Describes a functional requirement from the perspective of an end user.

Scenario: a concrete, focused, informal description of a single feature of the system used by a single actor

- Central is the textual description of the usage of a system. The description is written from an end user's point of view
- can also include video, pictures
- contain details about the work place, social situations and resource constraints

→ Scenario-based design: scenarios are used as the basis for the design of the hypothetical interaction of the end user with a new system.

Use of scenarios in development activities:

Scenarios can be used in many activities during the software lifecycle

- Requirements elicitation: as-is scenario, visionary scenario.
- Client acceptance test: evaluation scenario.
- System deployment: training scenario.
- Natural or formal language.

1. As-is scenario: Describes a current situation, describes the usage of an existing system, commonly used in re-engineering projects.
2. Visionary scenario: Describes a future system, used in all types of projects, aren't usually formulated by user or developer alone. → Greenfield engineering.
3. Evaluation scenario: Description of a user task against which the system is to be evaluated.
4. Training scenario: A description of the step by instructions that guide a novice user through a system. System delivery.

How do you find scenarios?

- Do not expect the client to be verbose → does only understand application domain not solution domain
- Do not wait for information if the system exists
- Engage in a dialectic approach

Finding Scenarios:

- Ask yourself or the client questions → do not rely on questions alone
- Insist on task observation if the system already exists (interface engineering or re-engineering)
  - also speak to the end user, not just the client

After the scenario is formulated:

- Find functions in the scenario where an actor interacts with the system
- Describe each of these use cases in more detail

Requirements validation: (6 criteria)

1. Correctness: the requirements represent the client's view
2. Clarity: the requirements can only be interpreted in one way
3. Completeness: all possible ways of using the system are described
4. Consistency: there are no requirements that contradict each other
5. Realism: the requirements can be implemented and delivered
6. Traceability: system components and behavior can be traced

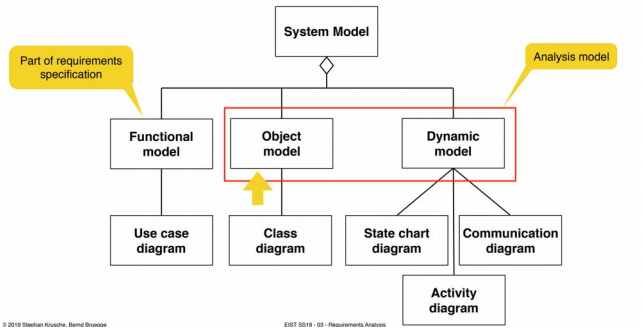
What should not be in the requirements?

- A description of the system structure, use of a specific implementation technology
- The development methodology
- A description of the development environment
- A specific implementation language
- Reuseability

Analysis

Analysis concepts:

- Analysis model: the object model and the dynamic model of a system to be developed
- Entity, boundary and control objects: objects can be divided into three major categories describing their use inside the system
- Generalization and specialization: hierarchies can be detected in two different ways to adopt object-oriented programming principles like inheritance/polymorphism and abstraction.



Object modeling:

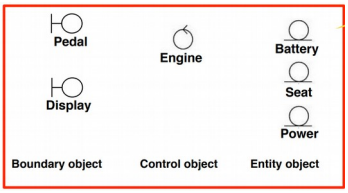
→define the structure of the system by identifying objects, attributes, operations(methods) and associations:

How to model:

1. brainstorming, finding classes, review names, attributes and methods
2. Associations: label generic associations, determine multiplicity of the associations
3. find taxonomies
4. simplify and organize

Stereotypes/Objects supported by UML:

- Entity: represent persistent information tracked by the system
- Boundary objects: represent the interaction between the user and the system.
- Control objects: represent the control tasks to be performed by the system.



Pros of stereotype graphics:

- UML diagrams are often easier to understand when they contain graphics and icons
- Graphics increase the readability of the diagram, especially for a non trained person

Cons of stereotype graphics:

- If developers are unfamiliar with symbols it is much harder for them to understand what is going on
- Additional symbols add to the burden of learning to read the diagrams

Other stereotypes:

- use case relationships: <<extends>>, <<includes>>
- Subsystem interface: <<interface>>
- Stereotype for classifying method behavior: <<constructor>>, <<getter>>, <<setter>>

Actor: Any entity outside the system, interacting with the system.
Class: A concept from the application domain or in the solution domain.
Object: A specific instance of a class.

Dynamic modeling:

- Describes the components of the system that have interesting dynamic behavior
- State chart diagrams: model the states of one class with interesting dynamic behavior
- Activity diagrams: model workflows in use cases and complex workflows in operations of object
- Communication diagrams: model the interaction between multiple classes/objects

→ UML communication diagrams

Generalization

- Identifies abstract concepts from lower-level ones
- Identify common features among different concepts, we create an abstract concept
- Common speech: “from low level to high level”, “from subclass to superclass”

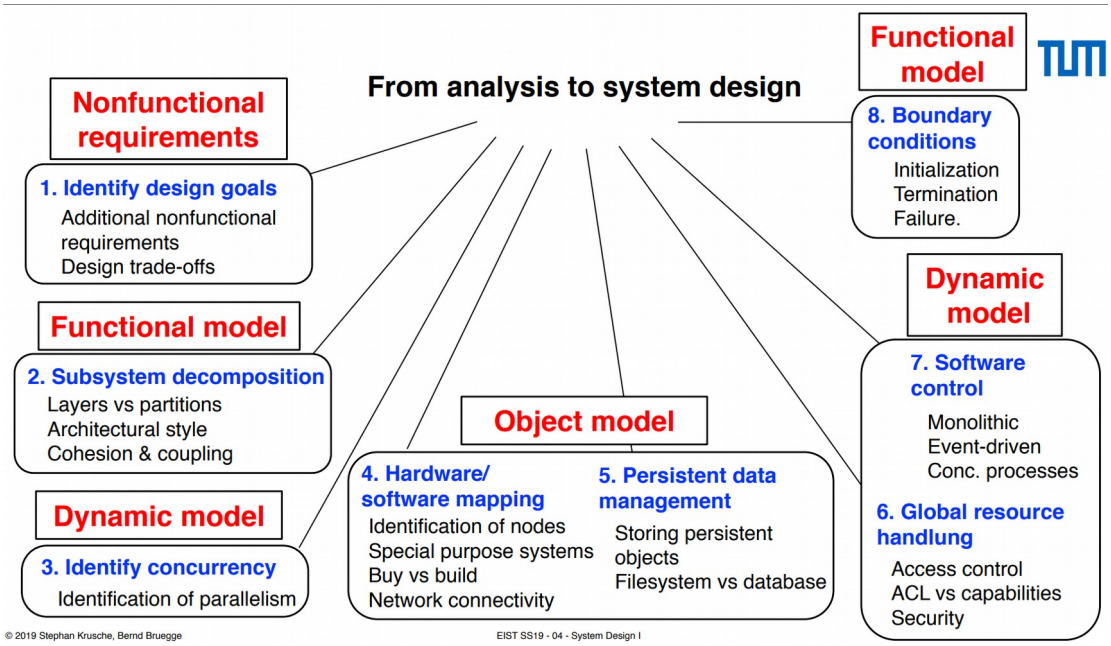
Specialization

- Identifies specialized concepts from higher level ones
- Identify more specific concepts from a high-level one

Vorlesung 4: System Design I

Why is design so difficult?
Analysis: focuses on the application domain
Design: focuses on the solution domain → The solution domain is changing very rapidly
Design window: time in which design decisions have to be made.
The Scope of System Design: Bridge the gap between a problem and an existing system

The 8 issues of System Design:

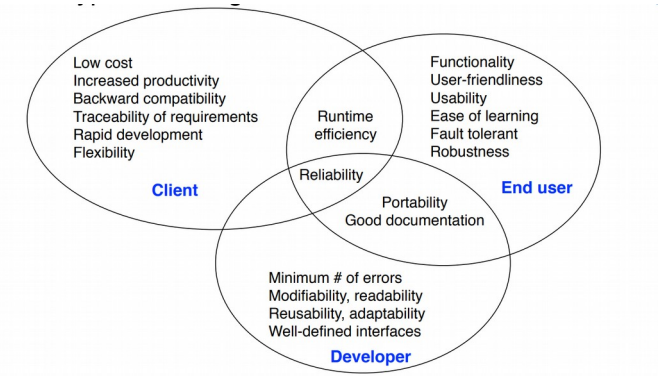


How the analysis model influence system design

- Nonfunctional requirements: definition of design goals
- Functional model: subsystem decomposition
- Object model: Hardware/software mapping, persistent data management
- Dynamic model: Identification of concurrency, global resource handling, software control
- Hardware/software mapping: Boundary conditions

Design goals:

- Design goals govern the system design activities
- As a starter: Any nonfunctional requirement is a design goals
- Additional design goals are identified with respect to
  - Design methodology
  - Design metrics
  - Implementation goals
- Design goals often conflict with each other → design goal trade-offs



Typical Design goal trade-offs

- functionality vs. usability → is a system with 100 functions usable?
- cost vs. robustness → a low cost design does not check for errors with wrong data
- efficiency vs. portability
- rapid development vs. functionality
- cost vs. re-useability → moving from 1-1 multiplicity to many-many multiplicity
- backward compatibility vs readability → using design patterns bc can be achieved

Subsystem decomposition: Identification of subsystems, services and their relationships
Subsystem:
Collection of classes, associations, operations, events that are closely interrelated with each other
The classes in the object model are the “seeds” for subsystems

Service:
A group of externally visible operations provided by a subsystem
The use cases in the functional model provide the “seeds” for services.

Subsystem interface: (Set of fully typed UML operations)
- Specifies the interaction and information flow from and to subsystem boundaries, but not inside the subsystem
- Refinement of service, should be well-defined and small
- Subsystem interface are defined during object design

Application programmer’s interface
- The API is the specification interface in a specific programming language
- are defined during implementation

The terms subsystem interface and API are often confused with each other

Subsystem interface object
- The set of public operations provided by a subsystem
- The subsystem interface object describes all the services of the subsystem interface
- A subsystem interface object can be realized with the Fa¢ade pattern.

Coupling and cohesion of subsystems: Reduce system complexity while allowing change
Coupling: measures dependency among subsystems
- High Coupling: Changes to one subsystem will have high impact on the other subsystem
- Low Coupling: A change in one subsystem does not affect any other subsystem.

Cohesion: measures dependencies among classes
- High Cohesion: classes in subsystem perform similar tasks, related through many associations
- Low Cohesion: many auxiliary classes, almost no associations

How to achieve high cohesion:
- High cohesion can be achieved if most of the interaction is within subsystems, rather than across subsystem boundaries
- Does one class from one subsystem always call another class from another subsystem for a specific service? → Consider moving the classes into one subsystem.

How to achieve low coupling:
- Low coupling can be achieved if a calling class does not need to know anything about the internals of the called class
- Can we make it possible that the calling class only calls operations of the lower level classes?
- Does the calling class really have to know the attributes of the classes in the lower layers?

High cohesion
- Operations work on same attributes
- Operations implement a common abstraction or service

Low coupling
- Small interfaces
- Information hiding
- No global data
- Interactions are mostly within the subsystem rather than across subsystem boundaries.

Fa¢ade pattern: Reduces Coupling

- A fa¢ade provides a unified interface for a subsystem
  - A fa¢ade consists of a set of public operations
  - Each public operation is delegated to one or more operations in the classes behind the fa¢ade
- A fa¢ade defines a higher-level interface that makes the subsystem easier to use
- A fa¢ade allows to hide design spaghetti from the caller

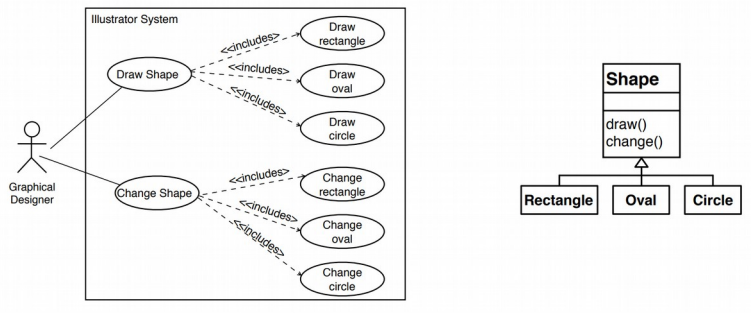
Closed architecture (opaque architecture) with a fa¢ade

- The subsystem decides exactly how it is accessed with the vehicle subsystem facade

Main advantages: reduced complexity, less recompilations
Additional advantages: A fa¢ade can be used during integration testing when the internal classes are not implemented, mock objects for each of the public methods in the fa¢ade

Decomposition: A technique used to master complexity ("divide and conquer")

1. Functional decomposition: the system is decomposed into functions → decomp- again
2. Object-oriented decomposition: the system is decomposed into classes ("objects") → decomp- again



Functional decomposition:

- the functionality is spread all over the system
  - source code hard to understand, complex and hard to maintain + Gui is awkward

Object-Oriented Approach:

1. Focus on the functional requirements
2. Find the corresponding use cases
3. Identify the participating objects
4. Use these participating objects to start the object model.

Architectural Style: a pattern for a subsystem decomposition

1. Layered architectural style: A layer is a subsystem that provides a service to another subsystem with the following restrictions:c
   1. A layer only depends on services from lower layers.
   2. A layer has no knowledge of higher layers.

Dijkstra layered architectural style: (1968) (T.H.E. system)

- A system should be designed and built as a hierarchy of layers.
- Each layer uses only the services offered by the lower layers.
- An operating system for single user operation → supporting batch mode

Hierarchical relationships between layers:

1. Layer A "depends on" layer B for its full implementation
2. Layer B "calls" layer B (runtime dependency)

Virtual machine:

- A subsystem connected to higher and lower level virtual by “provides services for” associations
- An abstraction that consists of a set of attributes and operations
- the terms layer and virtual machine can be used interchangeably

Closed architecture: (opaque layering)

- A layered architecture is closed, if each layer can only call operations from the layer below.
  - Maintainability, flexibility, portability

Open architecture: (transparent layering)

- A layered architecture is open if a layer can call operations from any layer below.
  - High performance, real-time operations support.

Properties of layered systems:

1. Closed architectures are more portable: low coupling
2. Open architecture are more efficient: high coupling
3. Layered systems have a chicken-and egg problem

→ 3 layered architec. style for web applications: web browser = gui, web server = requests, database = data

Layer vs. tier: (often used interchangeably)

3-layered architectural style: an application consists of 3 hierarchically ordered layers.
3-tier architecture: a software architecture where the 3 layers are allocated on 3 separate hardware nodes
→ a layer is a type(class, subsystem) and a tier is an instance (object, hardware node)
→ ISO’s OSI Reference model (defines 7 layers and communication protocols between the layers)

Client Server architecture:

- Often used in the design of database systems: front end: GUI, back end: database and manipulation
- Functions performed by the client: Input from the user, front-end processing of input data
- Functions performed by the server: Centralized data management, provision of data integrity and database consistency, provision of database security

Client server architectural style:

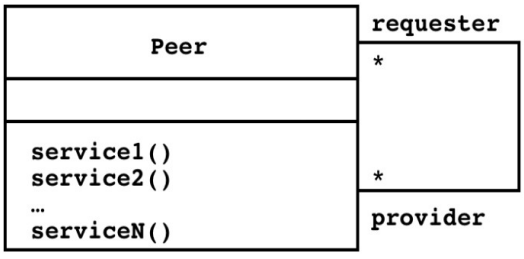
- Special case of the layered architectural style: servers provide services to instances of subsystems
- Each client calls a service offered by the server → server returns result to client
  - client knows the interface of the server, server doesn’t know the interface of the client
- The response is immediate
- End users only interact with the client.

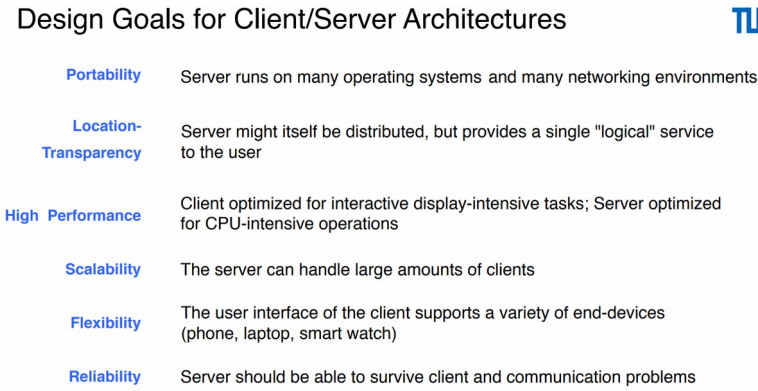
Problems with client server architectural style:

- use a request response protocol
- Peer-to-peer communication is often needed

Peer-to-peer architectural style:

- Generalization of the client server style
  - clients can be servers and servers clients
- Peer is new abstraction





Model view controller architectural style:
Problem: In systems with high coupling any change to the boundary objects(GUI) often force changes to the entity objects (data)

Solution: Decoupling! The model view controller (MVC) style decouples data access and data presentation
→View: A subsystem containing boundary objects (GUI)→displaying information to the user
→Model: A subsystem containing entity objects(data)→responsible for application domain knowledge
→Controller: A subsystem mediating between the views(data presentation) and the models (data access)
Pull notification variant: the view retrieves the data from the model
Push notification variant: the model updates the view after a change

Repository architectural style:
- support a collection of independent programs that work cooperatively on a common data structure called the repository
- in the architectural style these programs are called subsystems
- Subsystems access and modify data from the repository
- The subsystems are loosely coupled (interaction only through the repository)

Pipes and filters architectural style:
- consists of two subsystems called pipes and filters
  - Filter: A subsystem that does a processing step
  - Pipe: A pipe is a connection between two processing steps
- Each filter has an input pipe and output pipe
- Pipes and filters can be modeled with the UML activity diagram.

Vorlesung 5: System Design II

MVC benefits and challenges:

Benefits: Multiple synchronized views of the same model, pluggable views and controllers, exchangeability of look and feel, framework potential
Challenges: Increased complexity, potential for excessive number of updates, close connection between the view and controller, close coupling of views and controllers to model.

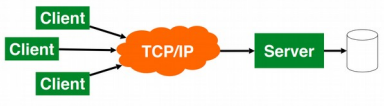
Steps for MVC

Step 1: simplify the object model (remove all attributes,methods andsubclasses from Class Diagram)
Step 2: identify object steroptypes
Step 3: Move the classes into packages

Elements of an architectural style:

1. Components(subsystems): Computational units with a specified interface (e.g. databases)
   1. Group of subtasks which implement an abstraction at some layer in the hierarchy
2. Connectors(communication): Interactions between the components(subsystems)
   1. Protocols that define how the layers interact

Client server architectural style: components and connectors



- Components: Subsystems are independent processes, servers provide specific services, client use these services
- Connectors: Data streams, typically over a communication network

Model view controller architectural style: components and connectors

- Components: 1 model, * view, 1 controller
- Connectors:
  1. Events: change propagation mechnanism via events ensures consistency between gui and model
  2. Method calls: readData(), initiateOperation(), update(), if the user changes the model through the controller of one view (the other views will be updated automatically)

Concurrency (3rd system design point)

- Used to address nonfunctional requirements such as: performance, response time, latency, availabilty
- Two objects are inherently concurrent if they can receive events at the same time without interacting
- Inherently concurrent objects can be assigned to separate threads of control
  - Objects with mutual exclusive activity can be folded into a single thread of control
  - Does the system provide access to multiple users?
  - Which entitiy objects can be executed independently from each other?
  - What kinds of control objects are identifiable
  - Can a single request to the system can be decomposed into multiple requests?
  - Can these requests be handled parallel?

Thread of control

- A thread of control is a path trough a set of states where only a single object is active at any time
- Concurrent threads can lead to race conditions
- A race condition is a design flaw where the output of a process depends on the specific sequence of other events

→ UML sequence diagrams

Implementing concurrency

1. Physical concurrency: Threads are provided by hardware (cores, processors, computer networks)
2. Logical concurrency: Threads are provided by software (usually provided by threads packages)
   →In both cases: Which thread runs when?
   →Today's operating systems provide a variety of scheduling mechanism (round robin, time slicing..)
   →Concurrency introduces problems such as starvation, deadlocks, fairness
   →Sometimes scheduling problems solved by you, sometimes by the virtual machine

4. Hardware software mapping:(4th in system design)

1. How shall we realize the subsystems: With hardware of with software?
2. How do we map the object model onto the chosen hardware and/or software?
   1. Mapping the objects: processor, memory, input/output devices
   2. Mapping the objects: network connections.

Mapping the objects:

Control objects → processor

- Is the computation rate too demanding for a single processor?
- Can we get a speedup by distributing objects across several processors?
- How many processors are required to maintauin a steady state load?

Entity objects → memory

- Is there enough memory to buffer bursts of requests?

Boundary → input/output devices

- Do we need an extra piece of hardware to handle the data generation rates?
- Can the desired response time be realized with the available communication bandwidth between subsystems?

Difficulties: (Hardware Software Mapping)

- Much of the difficulty of designing a system comes form addressing externally-imposed hardware and software constraints
- Certain tasks have to be at specific locations
- Some hardware components have to be used from a specific manufacturer

→UML component diagram(top level view of the system)
→UML deployment diagram(showing a design after system design decisions have been made)

5. Persistent data management(5th in system design)

- All classes of type entity in the system model need to be persistent:
  - class is persistent, if the values of their attributes have a lifetime beyond a single execution
- Realization of persistent classes:
  1. File system: data is used by multiple readers but a single writer
  2. Database system: data is used by concurrent writers and readers

→ Mapping an object model to a database

6. Global resource handling (6th in system design):

Access control: In multi-user systems different actors usually have different access rights to functions and data → modeled with use cases

Access matrix: Model the access of actors on classes(sparse matrix)

→ rows represent the actors, columns represent classes whose access control is wanted

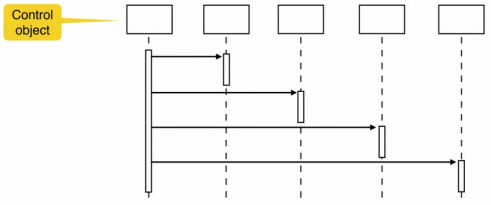
→ Access right: an entry in the access matrix

Software Control (7th in system design)

1. Implicit software control: Rule-based systems
2. Explicit software control: Centralized/Decentralized control

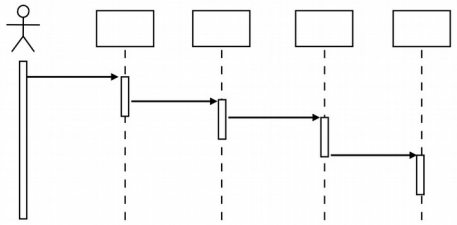
Fork diagram:

- dynamic behavior is placed in a single object, usually a control object
- It knows all the other objects and often uses them for direct questions and commands



Stair diagram:

- dynamic behavior is distributed
- objects knows only few of the other objects and knows which object can help with a specific behavior



Centralized explicit software control:

- Procedure-driven: control resides within the program code
- Event-driven: control resides within a dispatcher calling other functions via so called callbacks

Decentralized explicit software control:

- control resides in several independent objects
- Promises a possible speedup by mapping the objects on different processors, but require increased communication overhead.

Centralized design:

- one control object or subsystem controls everything
- → change in structure is very easy, ↓ the single control object is a performance bottleneck

Decentralized design:

- not a single object is in control, control is distributed, more than one control object
- → fits nicely into object-oriented development, ↓ additional communication overhead

Boundary conditions (8. Boundary conditions)

- Initialization: the system is brought from a non-intialized state to steady-state
- Termination: resources are cleaned up and other systems are notified upon termination
- Failure: bugs, errors, external problems

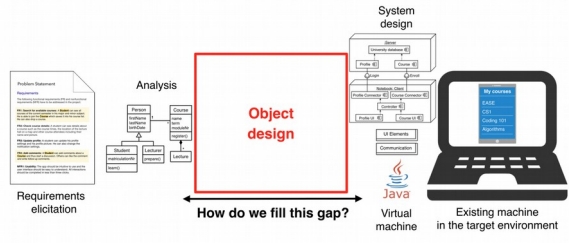
→ Good system design foresees fatal failures and provides mechanism to deal with them.
→ best modeled as use cases or administrative use cases
→ Actor is often the system administrator
→ use cases: start up/termination/error of a subsystem and full system

Lecture 6: Object Design I

Purpose of object design

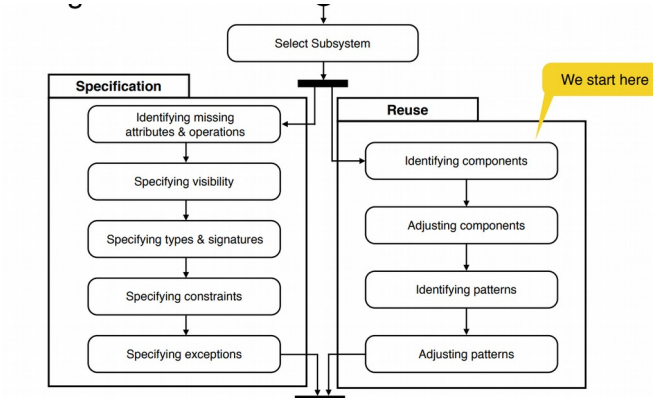
1. Prepare for the implementation of the system model based on design decisions
2. Transform the system mdoel (optimize it)
3. Investigate alternative ways to implement the system model → design goals
4. Serves as the basis of implementation

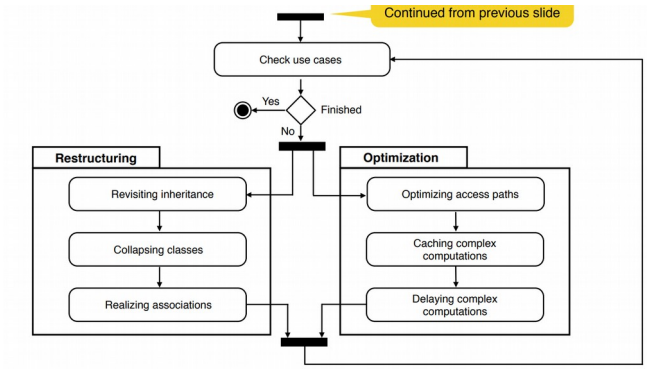
→ closes the gap between analysis and system design



Object design consists of 4 activities

1. Reuse: Identification of existing solutions
2. Interface specification: Describes precisely each class interface
3. Object model restructuring: transforms object design model to improve its understandability
4. Object model optimization: transforms object design model to address performance criteria





1. Identifying components:
   - Identify the missing components in the design gap
   - Make a build or buy decision to obtain the missing component

Reuse, reuse, reuse:

1. Reuse of design knowledge
2. Reuse of existing classes
3. Reuse of existing interfaces

Techniques to close the object design gap:

1. Composition(black box reuse): new class is created by aggregation of existing classes → offers aggregated functionality of existing classes
2. Inheritance(white box reuse): new class created by subclassing → reuses functionality of superclass

Inheritance in software engineering:

1. Description of taxonomies: used during requirements elicitation and analysis
   1. Activity: Identify application domain objects that are hierarchically related
   2. Goal: Make the analysis model more understandable
2. Interface specification: used during object design
   1. Activity: Identify the signatures of all identified objects
   2. Goal: Increase the reuseability, enhance the modifiability and the extensibility

Interface specification:

1. Implementation inheritance: subclassing from an implementation
   - A class is already implemented that does almost the same as the desired class
   - Extending a base class by a new operation or overriding an existing operation
   - Problem: The inherited operations might exhibit unwanted behavior

- Delegation:
  - A way of making composition as powerful for reuse as inheritance
  - catching an operation and sending it to another object
  - The existence of the receiver makes sure, that the client cannot misuse the delegate object
  - client calls receiver → receiver delegates to delegate

Implementation inheritance vs. delegation:

Delegation: flexible, because any object can be replaced at run time by another one
inefficient, because objects are encapsulated

Inheritance: straightforward to use, supported by many languages, easy to implement
inheritance exposes some methods of the parent class

Discovering inheritance associations:

1. Generalization: The discovery of an inheritance association between two classes, where the sub class is discovered first
2. Specialization: The discovery of an inheritance association between two classes, where the super is discovered first

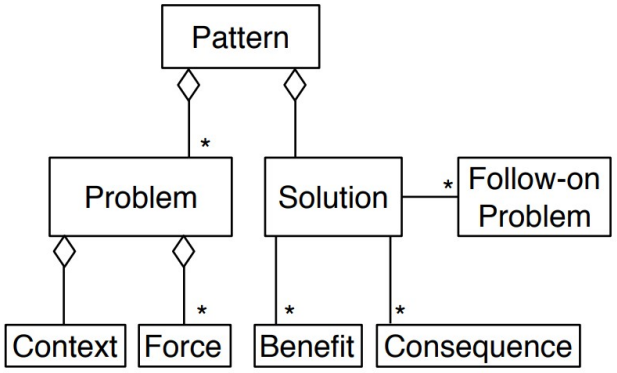
Patterns

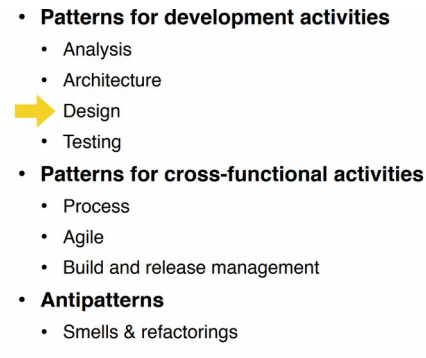
Algorithm: A method for solving a problem using a finite sequence of well-defined instructions for solving

Pattern: A pattern describes a problem which occurs over and over again in out environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Modeling a pattern in UML

- a pattern has a problem and a solution
  - problem class: context and set of forces
  - solution resolves these forces with benefits
- Solutions generate follow-on problems



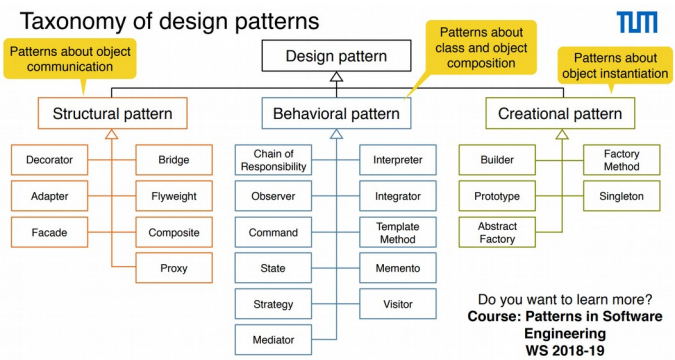


Why are design patterns good?

- There are generalizations of detailed design knowledge from existing systems
- They provide vocabulary
- They provide examples of reuseable designs

3 types of design patterns:

1. Structural patterns:
   1. reduce coupling
   2. introduce an abstract class to enable future extensions
   3. encapsulate complex structures
2. Behavioral patterns:
   1. allow a choice between algorithms
   2. Allow the assignment responsibilites to object
3. Creational patterns
   1. Allow to abstract from complex instantiation process
   2. Make the system independent



→Composite Pattern
→Bridge Pattern
→Proxy Pattern

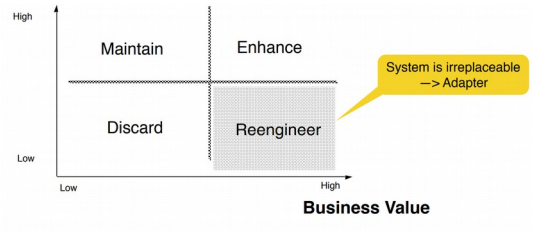
Lecture 7: Object Design II

Legacy System:

- An old system that continues to be used, even though newer technology or more efficient methods are now available
- Evolved over a long time, still actively used in a production environment
- high maintenance cost
- considered irreplaceable because a re-implementation is too expensive or impossible

Problems with legacy systems:

- System cost: The system still makes money, the cost of designing a new system with the same functionality is too high
- Poor engineering: The system is hard to change because the compiler is no longer available or source code has been lost
- Availability: The system requires 100% availability. It cannot be taken out and be replaced with a new system.
- Pragmatism: The system is installed and working



→ but change is required due to new functional, nonfunctional or pseudo requirements.

→Adapter Pattern(connect incompatible components and allows reuse of existing components)
→Observer Pattern(maintain consistency across redundant observers. Basis for MVC.
→Strategy Pattern(switch between multiple implementations of an algorithm at runtime.

Clues for the use of design pattern:

- “complex structure”, “must have variable depth and width” → Composite Pattern
- “must provide a policy independent from the mechanism”, “must allow to change algorithms at runtime”→Strategy Pattern
- “must be location transparent” → Proxy Pattern
- “changes state often” “must be extensible”, “must be scalable” → Observer Pattern (MVC Architectural Pattern)
- “connect incompatible components and must interface with an exisiting object” → Adapter Pattern
- “must interface to several systems, some of them to be developed in the future”, “an early prototype must be demonstrated”, “must provide backward compatibility” → Bridge Pattern
- “must interface to existing set of objects”, “must interface to existing API”, “must interface to existing service” → Fa¢ade Pattern

Lecture 8: Model transformation and refactoring

Model driven engineering

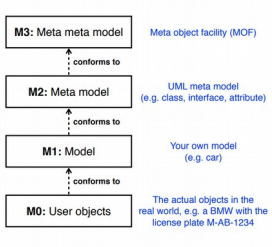
- A methodology which focuses on the creation of domain models
  - Domain models are abstract representations of the knowledge and activities
- The aim is to increase productivity through
  - Reuse of standardized models
  - Reuse of knowledge
  - Collaboration between developers working on the systems
  - Standardized terminology
  - Identification of best practices in the application domain

Model based software engineering (MBSE):

- Application of modeling to support requirements, design, analysis, verification, validation activities
- Advantages:
  - Better communication and knowledge management
  - Impact analysis of requirements and design changes
  - More complete representation
- A system engineering model contains several models addressing aspects of the participating systems:
  - Functional model, behavioral model, structural model, cost model, organizational model, development environment, target environment.

Model transformation:

- The goal in model-driven engineering is to automate model transformations
- Models conform to meta models
- A transformation can also be seen as model that conforms to a meta model



→ Input: A model conforming to a meta model
→ Output: Another model conforming to a meta model

Refactoring: definition

- Refactoring: A change made to the internal structure of source code to make it:
  - easier to understand, cheaper to modify → without changing observable behavior
  - restructure source code by applying a series of refactorings

→ Mapping UML to Java source code
→ UML state chart diagrams

State:

- An abstraction of the attributes of a class
- An equivalence class of all those attribute values that do need to be distinguished
- State has duration

→State Pattern
→ Mapping Models to contracts

Contract:

- Contracts are constraints on a class that enable class users, class implementors, and class extenders to share the same assumptions about the class
- specifies:
  - constraints that the class user must meet before using the class
  - constraints that are ensured by the class implementor and the class extender when used
- Contracts are modeled with OCL to define contracts for UML Models

→Mapping contracts to Java
→ Mapping models to tables

Source code transformation: code refactoring:

→ Tool support: IDEs provide support for refactoring → very limited though

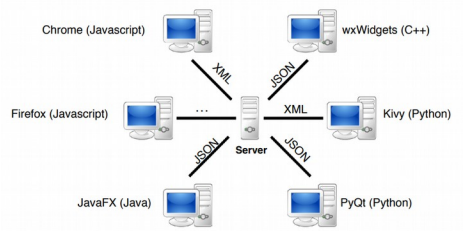
Refactoring principles:

- Why do we refactor?
  - To improve the design of software
  - To make model and source code easier to understand
- When should we refactor?
  - Refactor when you add functionality
  - Refactor when you need to fix a bug
  - Refactor as you do code reviews
  - Refactor when the code starts to smell
- What about performance?
  - Worry about performance only when you have identified a performance problem.

Object transformation:

- Dealing with data exchange often requires transformations: maybe server speaks other programming language

→ standardized data exchange formats: intermediate representation of data
→ serialization/deserialization with JSON

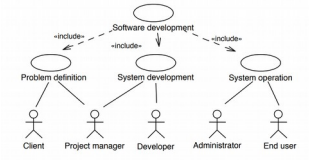


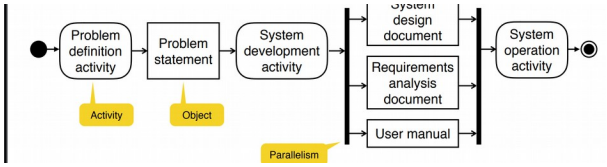
→ Reverse engineering, mapping source code to model

Lecture 9: Software Lifecycle Modeling

→ The management of software projects itself can be seen a complex system
→ We can use the same modeling techniques we use for software development

1. Functional model of a software lifecycle
    1. Scenarios, user stories
    2. Use case model
2. Structural model of a software lifecycle
    1. Object identification
    2. Class diagrams
3. Dynamic model of a software lifecycle
    1. Sequence diagrams, state chart and activity diagrams

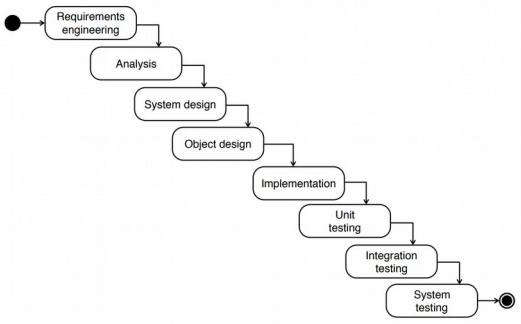




→ UML activity diagrams

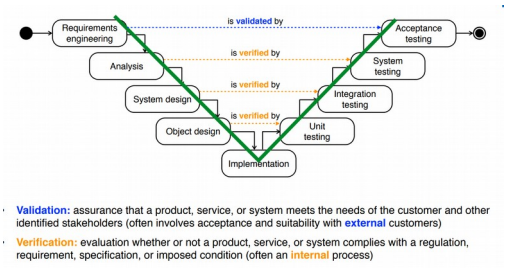
1. Linear Models: Waterfall model

- First described in 1970 by Royce
- Activity centered lifecycle model that prescribes a sequential execution of activites
    - Assumes that software development can be scheduled as a step-by-step process that transforms user needs into code
    - Never turn back once an activity is completed
- The key feature of his model is the verification activity that ensures that each development activity does not introduce unwanted or deletes mandatory requirements.
- Provides a simplistic view of software development that measures
- progress by the number of tasks that have been completed.



2. Linear Models: V-Model

- Horizontal arrows show information flow between activities of the same abstraction level
    → layout of the activities has no semantics in UML
- Higher levels of abstractions of the V-model deal with the requirements in terms of elicitation and operation
    → client acceptance activity validates the understanding of the problem into a software architecture
- The middle part focuses on mapping the understanding of the problem into a software architecture
    - Integration tests verify components and subsystems against the preliminary design
- The lower level focuses on details such as the assembly and coding of software components
    - Unit tests verify classes and methods against their description in the detailed design



Validation: Assurance that a product, service, or system meets the needs of the customer and other identified stakeholders. Often involves acceptance and suitability with external customers → Are you building the right thing?

Verification: Evaluation whether or not a product, service, or system complies with a regulation, requirement, specification or imposed condition. Often an internal process. → Are you building it right?

Properties of sequential models:

Managers love sequential models

- Nice milestones
- No need to look back (linear system)
- Always one activity at a time
- Easy to check progress during development

However software development is non-linear:

- During design: problems with requirements are identified
- During implementation: design and requirement problems are found
- During testing: coding errors, design errors and requirement errors are found

3. Iterative models: Spiral model

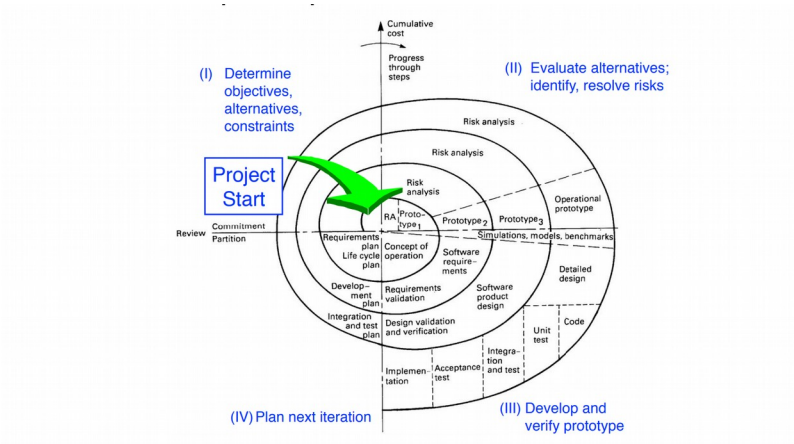
- The spiral model was proposed by Barry Boehm 1987 to deal with the problems of the Waterfall model
- It is an iterative model with 4 major activities:
  - Determine objectives, alternatives and constraints
  - Evaluate alternatives and identify risks, resolve these risks by assigning priorities to them
  - Develop a series of prototypes for the identified risks starting with the highest risk. Use a waterfall model for each prototype devlopment
  - If a risk has successfully resolved, evaluate the results of the iteration and plan the next iteration

→ If the risk could not be resolved, the project is terminated immediately

→ The 4 activities are applied in 9 iterations

Iterations:

1. Concept of operations
2. Software requirements
3. Software product design
4. Detailed design
5. Code
6. Unit test
7. Integration and test
8. Acceptance test
9. Implementation

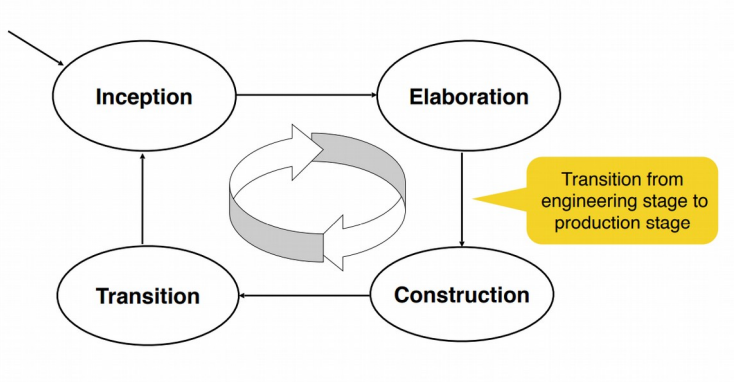


Another iterative model: Unified process

- Developed by Booch, Jacobson, and Rumbaugh
- Iterative and incremental lifecycle model built on the idea of cycles in the lifetime of a software system
- Each cycle consits of 2 stages and 4 phases: inception, elaboration, construction, transition
- Each phase can consist of several iterations
- Several workflows are performed in parallel: management, environment, requirements, design, implementation, assessment, deployment

The 2 stages in the unified process:

1. Engineering stage:
    1. Less predictable but smaller teams, focusing on design and synthesis activities
    2. The engineering stage consists of 2 phases
        1. Inception phase
        2. Elaboration phase
2. Production Stage:
    1. More predictable but larger teams, focusing on construction, test and deployment activities.
    2. The production stage also consists of 2 phases
        1. Construction phase
        2. Transition



An iteration creates an informal, internally controlled versions of artifacts:

- It leads to a “minor milestone”
- Iteration to iteration transition:
- Triggered by specific software development

A phase creates a formal, stake-holder approved version of artifacts :

- It leads to a major milestone
- Phase to phase transition:
- triggered by a significant business decision

Limitations of linear and iterative models:

- do not deal well with frequent change
    - The waterfall model assumes that once you are done with one activity, all issues covered in that activity are closed an cannot be reopened.
    - The spiral model can deal with change between activities, but does not allow change within an activity
- What do you do if change is happening more frequently?
    → Agile methods

How change influences choice of the software lifecycle model:

1. If change rarely occurs → Sequential model: Waterfall model, V-model
2. If change occurs sometimes: → Iterative model: Spiral model, unified process
3. If change is frequent: → Agile model: Spiral model, Unified process

Incremental: add onto something → helps you to improve your process
Iterative: redo something v rework something → helps you to improve your product
Adaptive: means react to changing requirements → improves the reaction to changing customer needs

Increments can cause iterations:

- Product iteration can result from incremental addition of functionality that had not been anticipated in a previous iteration
- Product iteration can also result from restructuring

Defined process control model:

- Requires that every piece of work is completely understood
    - Deviations are seen as errors
- Given a well-defined set of inputs, the same outputs are generated every time.
    - Precondition to apply this model: all the activities and tasks are well defined to provide their repeatability and their predictability
    - If the preconditions are not satisfied: lots of surprises, loss of control, incomplete or wrong work products

Empirical process control model:

- Imperfectly defined process, not all pieces of work are completely understood
  - Deviations, errors and failures are seen as opportunities that needed to be investigated
- Expects the unexpected: control and risk management is exercised through frequent inspection
- Apply when: Change is frequent and cannot be ignored, change of requirements, change of technology, change in the organization, change of people

Scrum is based on an empirical process control model

- Original definition(from Rugby): A scrum is a way to restart the game after an interruption
- Definition in agile processes: Scrum is a technique that deals with interruptions (change)
  - Improves risk management by improved communication, cooperation and the delivery of product increments

→ Antithetical to linear processes, where work is performed sequentially, work in agile processes is performed in parallel

History of Scrum

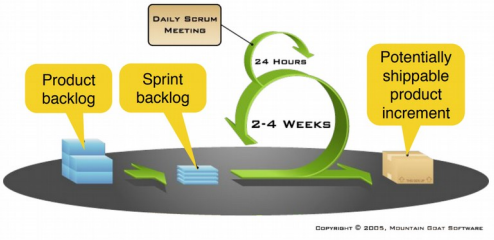
1. 1995: Jeff Sutherland and Ken Schwaber analyzed common software development processes
2. 1996: Introduction of Scrum at OOPSLA
3. 2001: Publication “Agile Software Development with Scrum” by Ken Schwaber&Mike Beedle
4. The Scrum founders are also members in the Agile Alliance which published the Manifesto for Agile Software Development.

Manifesto for Agile Software Development:

- Individuals and interactions are more important than processes & tools
- Working software is more important than comprehensive documentation
- Customer collaboration is more preferable than contract negotiation
- Responding to change is more preferable than following a plan

3 Scrum artifacts

1. Product backlog: List of requirements for the whole product
2. Sprint backlog: List of requirements and tasks for one iteration (“sprint”)
3. Potentially shippable product incremement: Release to the product owner that contains all results of the current sprint

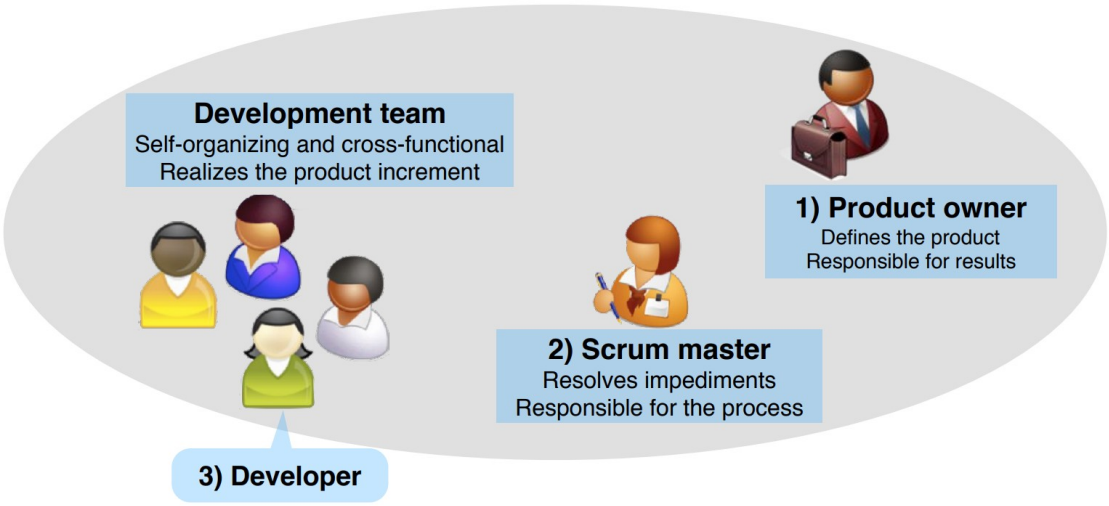


5 Scrum meetings:

1. Project kickoff meeting (start of the project): Create and prioritize the product backlog
2. Sprint planning meeting (start of each sprint): Create the sprint backlog
3. Daily scrum meeting (every day, 15 min): Share status, impediments and promises
4. Sprint review meeting (end of each sprint): Demonstrate the realized backlog items to the product owner
5. Sprint retrospective (after the sprint): Inspect the previous sprint and create a plan for improvements to be enacted during the next sprint.

→ Daily Scrum meeting:

- A short (15 minutes long) meeting, which is held every day before the Team starts working
- Main purpose: risk reduction by early information sharing and discussion
- Participants: Scrum master (moderator), Scrum development team
- Every team member should answer on 3 questions:
  - Status: What did you do since the last meeting?
  - Impediments: Are there any impediments in your way?
  - Promises: What do you promise to resolve until the next meeting?



Scrum Team

Lecture 10: Software Configuration Management

Why software configuration management?
Problems:

- Multiple people have to work on software that is changing
- More than one version of the software has to be supported: Released Systems, Custom configured systems(different functionality), Systems under development, Support for different machines&operating systems

→ Need for coordination: software configuration management

- Manages evolving software systems
- Controls the effort involved in making changes to a system.

What is software configuration management (SCM)?

- A set of management disciplines within a software engineering process to develop a baseline
- Encompasses the disciplines and techniques of initiating, evaluating and controlling change to software products during and after a software project.
- Standards (approved by ANSI)
  - IEEE 828: Standard for software configuration management plans(SCMP)
  - IEEE 1042: Guide to software configuration management
- Contains different activities and roles

Software configuration management activities:

1. Configuration item identification: Modeling the system as a set of evolving components
2. Change management: handling, approval & tracking of change requests
3. Promotion management: creation of versions for other developers
4. Branch management: management of concurrent development
5. Build and release management: creation of versions for client and users
6. Variant management: management of coexisting versions

Configuration management roles:

1. Configuration manager: responsible for identifying configuration items
2. Change control board member: responsible for approving or rejecting change requests
3. Developer: creates promotions triggered by change requests or the normal activities.
4. Auditor: Responsible for the selection and evaluation of promotions for release and for ensuring the consistency and completeness of this release.

1. Configuration item identification: Modelling the system as a set of evolving components

Terminology: configuration item

- An aggregation of software, hardware, or both, designated for configuration management and treated as a single entity in the configuration management process
- Software configuration items: source files, models, tests, binaries, documents
- Hardware configuration items: CPUs, sensors, actuators, physical infrastructure

Indentification of configuration items:

- Selecting the right configuration items is a skill that takes practice
- Very similar to object modeling
- Use techniques similar to object identification for finding configuration items

Terminology: baseline:

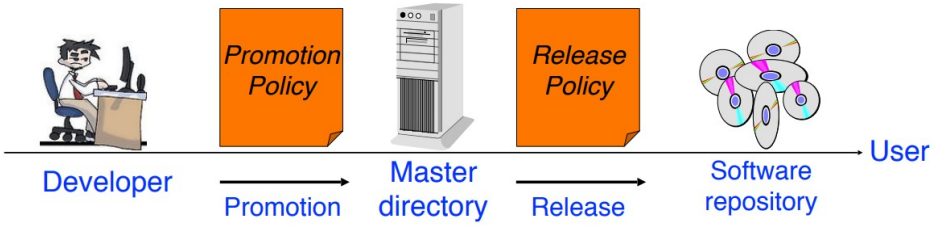
- A specification or product that has been formally reviewed and agreed to by responsible management
- It serves as the basis for further development

Examples:

1. Baseline A: The api has been defined, method bodies are empty
2. Baseline B: setter and getter methods are implemented
3. the gui is implemented

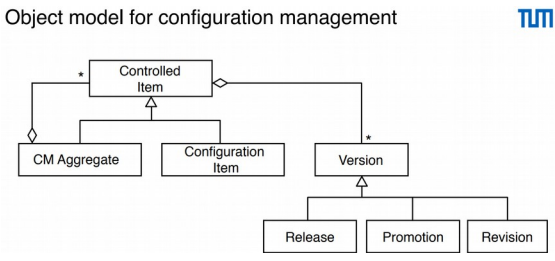
Change policies:

- Promotion: the internal development state of a software is changed
- Release: a changed software system is made visible outside the development organization



Terminology: SCM directories

- Programmers's directory: holding newly created or modified software entities, controlled by programmer only
- Master directory: Manages the current baseline and for controlling changes made to them, changes must be authorized
- Software repository: archive for the various baselines released for general use, copies of these baselines may be made available to requesting organizations



→ Version: specific instance of a configuration item, different versions have different functionality

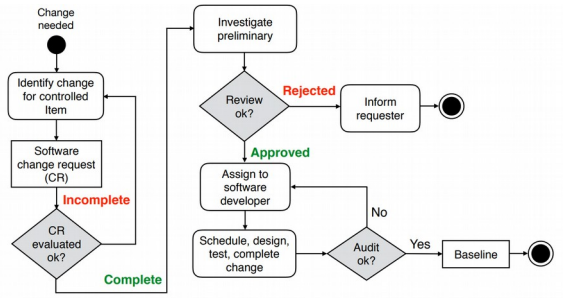
→ Release: The external distribution of an approved version

→ Promotion: a version that is made available to other developers

→ Revision: change to a version that corrects only errors in the design/code

2. Change Management: The handling, approval & tracking of change requests

- The handling of change requests
- Important: A baseline can only be changes through a formal change control procedure
- The process:
  - A change is requested
  - It is evaluated against requirements and project constraints and reviewed by the configuration control board
  - Following these assessments, the change request is approved or rejected
  - If it is approved, the change is assigned to a developer who will design, implement and test the change accordingly
  - The implemented change is audited to create another baseline



Change policy:

- The purpose of a change policy is to guaranteed that each promotion or release conforms to commonly accepted criteria
- Examples for change policies: Promotion policy: no developer is allowed to promote source code
- Release policy: No baseline can be released without having been beta-tested by at least 500 external people.

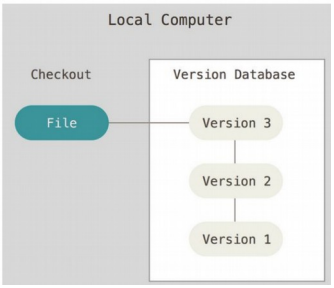
3. Promotion management: The creation of versions for other developers

Version control systems

- VCS allow many software developers to collaboratively work on the configuration items in a given project
- VCS store different versions of configuration items in a commit history and allow to restore previous versions
- The commit history allows developers to see how the configuration items changed over time and to see who changed a certain item
- Versions are stored in a repository and developers can check out a version into a working copy

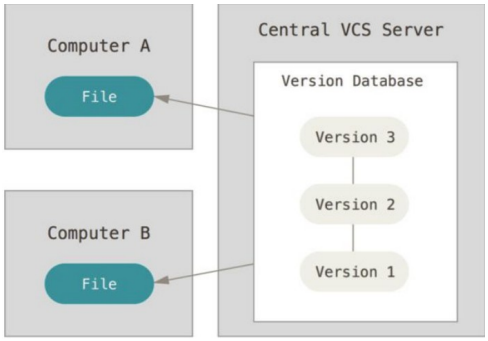
Monolothic architecture

- Developers have a simple local database that keeps all the changes to files under revision control
- Still distributed with many computers today



Repository architecture for version control:

- A single server contains all the versioned files
- Developers check out files from the server to their computer, change them and check them back into the central server
- Fine-grained control over who can do what
- Problem: single point of failure in the central VCS server: possibility of loosing all their versions and their history if the server crashes



Peer-to-peer architecture for version control:

- Addresses the single point of failure problem
- Each developer computer fully mirrors the repository
- Developers can work offline and create versions
- Not all versions are promoted to the master directory
- If the server dies and a programmer has a full copy of the repository, it can be recovered

Comparison of distributed vs centralized version control systems:

Advantages:

- ability to work offline
- ability to work incrementally
- ability to context switch efficiently
- ability to do exploratory coding efficiently

Disadvantages:

- high learning curve
- scaling issues
- less administrative control

Git:

- Open source project
- Add: select changes for the commit
- Clone: creates a copy of the remote repository
- Commit: add the changes to the local repository
- Push: upload local changes to remote repository
- Fetch: download changes from the remote repository into the local repository
- Merge: apply changes into the programmer's directory
- Pull: fetch followed by merge

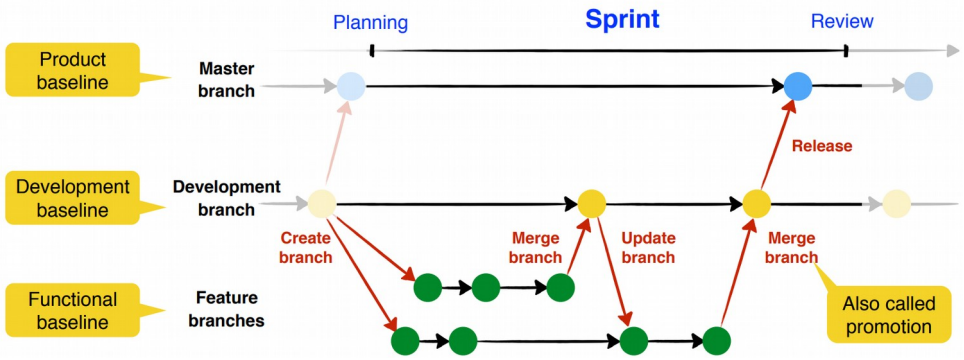
4. Branch Management

Types of baselines: As system are developed, a series of baselines is created, usually after a review.

- As systems are developed, a series of baselines is created, usually after a review
  - Product baseline
  - Developmental baseline
  - Functional baseline

Git branch management model:

- **Master branch:** external release (e.g. Scrum product increment)
- **Development branch:** internal release
- **Feature branches:** incremental development and explorations

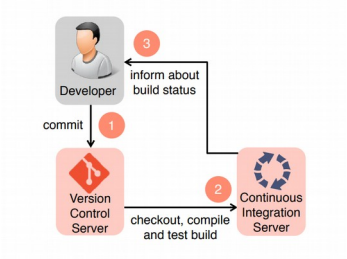


Best practices for branch management

- You can use branches to control concurrent development explorations, but you should agree on a branching model
- Only one person at once work with non merge-able files
  - Communicate with the team before changing them
  - Minimize the working time on such files

Continuous integration:

- A software development technique where members of a team integrate their work frequently



- Usually each person integrates at least daily, leading to multiple integrations per day
- Each integration is verified by an automated build which includes the execution of tests – regression tests – to detect integration errors as quickly as possible

Advantages of continuous integration:
- There is always an executable version of the system
- Developers and managers have a good overview of the project status
- Automatic regression testing

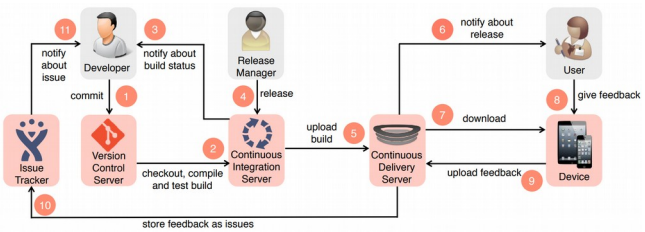
→ Regression testing:
Goal: verify that software developed and tested still performs correctly even after it was changed or interfaced with other software
→ finds errors in the existing source code immediately after a change is introduced
→ downside: can be very costly to execute a large test suite after each change

Techniques: Retest all, regression test selection, test case prioritization
When to execute the selected test cases? After each change, nightly, weekly?

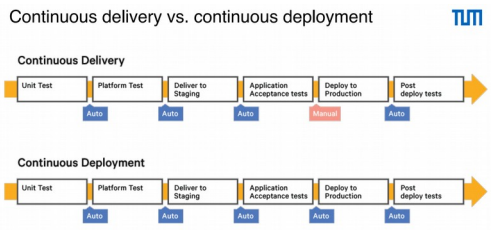
→ Build and dependency management example: Maven
Open source tool for java projects
stores libraries, frameworks and plugins in a central repository
POM = Project Object Model (main artifact and configuration file)
Support for multiple build lifecycle phases,

Examples of continuous integration systems:
Bamboo, Azure DevOps, Jenkins, Team City, GitLab CI, Travis CI

Release Management:
→ How it should not be done: delivery only after the implementation was finished, rare releases during project development, little feedback from clients and users during development
Requirements:
1. Large and distributed software projects need to provide a development infrastructure with am integrated build management that supports:
    1. Regular builds from the master directory
    2. Automated execution of tests
    3. E-mail notification
    4. Determination of code metrics
    5. Automated publishing of the applications and test results
2. The transition from source code to the executable application consists of these activities:
    1. Setting required paths and libraries
    2. Compiling source code
    3. Copying source files
    4. Setting file permissions
    5. Packaging the application



Terminology
- Continuous integration: see definition
- Continuous delivery: approach in which teams keep producing valuable software in short cycles and ensure that the software can be reliably released at any time.
- Continuous deployment: every change that passes the automated tests is deployed automatically
- Continuous software engineering: organizational capability to develop, release and learn from software in short cycles.



Continuous delivery: benefits and challenges

- Benefits
  - Accelerated time to market
  - Building the right product: improved product quality and customer satisfaction
  - improved productivity and efficiency
  - Reduced risk of a release failure
- Challenges
  - Organizational: Varying interests in different departments of an organization
  - Process: Traditional processes hinder continuous delivery
  - Technical: Maintainability of the source code & tailorable delivery workflows for heterogeneous project environments.

Lecture 11: Testing

Terminology:

1. Failure: Any deviation of the observed behavior from the specified behavior(crash)
2. Error: (erroneous state) The system is in a state such that further processing by the system can lead to a failure.
3. Fault: The mechanical or algorithmic cause of an error (bug)
4. Validation: Activity of checking for deviations between the observed behavior of a system and its specified behavior

Unit testing:
method where individual units in a program are tested

- in procedural programming, a unit is usually a function or procedure
- in object oriented programming, a unit us usually the class:L
  - a unit can also be an attribute, an individual method or the interface of the class
- unit tests are short code fragments created by programmers or occasionally by white box testers during the development process
- unit tests form the basis of integration testing

Guidance for test case selection:

- Use analysis knowledge about functional requirements(black box testing):
  - Scenarios and use cases, expected input data, invalid input data
- Use design knowledge about system structure, algorithms, data(white box testing):
  - Control structures, test branches, loops, classes and data structures, test methods, attributes, records fields, arrays…
- Use implementation knowledge about algorithms and data structures:
  - Force a division by zero
  - If the upper bound of an array is 10, then use 11 as index.

Junit overview:

- A java framework for writing and running unit tests
- Designed initially by Kent Beck and Erich Gamma with test first in mind
  - Test written before before implement the system
  - Observe those test cases that create failures
  - Write new code or fix existing code to make the test pass

Junit 4 uses Java annotations and Java assertions instead of inheritance
Annotations:

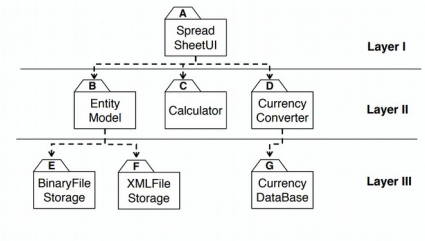
1. @Test: identifies that method is a test method
2. @Test (expected=Exception.class): Throws if the test method throws the named exception
3. @Test (timeout=100): Thes test fails if it takes longer than 100 milliseconds
4. @Before: Perform this method as the first test method
5. @After: Any test method must finish with call to this method
6. @BeforeClass: Perform the following method before the start of all tests, used to perform time intensive activities
7. @AfterClass: Perform method after all tests have finished. Used to perform clean-up activites
8. @Ignore(String) : Ignore the test method prefixed by @Ignore, print out the string instead.

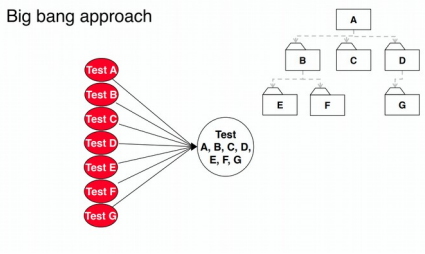
Assertions:

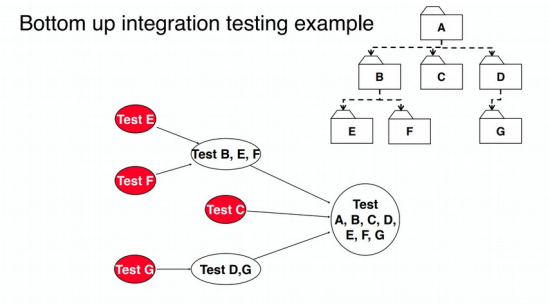
1. assertTrue: (message, Predicate): checks if predicate evaluates to true; otherwise throws an exception
2. assertFalse: (message, Predicate): checks if predicate evaluates to false; otherwise throws an exception
3. fail(String): let the method fail, useful to check that a certain part of the code is not reached
4. assertEquals (message, expected, actual): checks if the values expected and actual are the same; otherwise throws an exception including the message
5. assertEquals (message, expected, actual, tolerance): used for float and double; tolerance specifies the number of decimals which must be the same.
6. AssertNull (message, object): Check if the object is null; otherwise throws an exception
7. asserNotNull (message, object): Check if the object is not null; otherwise throws an exception including the message
8. assertSame (message, expected, actual): Check if both variables refer to the same object; otherwise throws an exception including the message
9. assertNotSame (message, expected, actual): Check that both variables expected and actual do not refer to the same object; otherwise throws an Exception including the message.

Integration testing:

- The entire system is viewed as a collection of subsystems (set of classes) determined during the system and object design
- Goal: Test all interfaces between subsystems and the interaction of subsystems
- The integration testing strategy determines the order in which the subsystems are selected for testing and integration
  - Big Bang integration
  - Bottom up testing: The subsystems in the lowest layer of the call are tested individually → then the subsystems above this layer are tested and call previous subsystems
  - Top down testing
  - Vertical integration

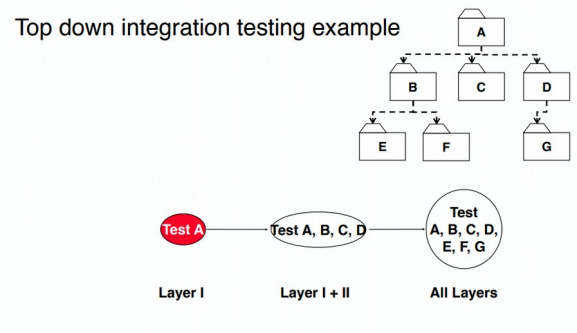






Pros and Cons: bottom up integration testing:
→ no doubles needed
→ useful for integration of oop systems and systems with performance requirements

→ tests an important subsystems last
→ test drivers are needed



Pros and Cons: top down integration testing
→ Test cases can be defined in terms of the functional requirements of the system
→ No test drivers needed
→ Doubles are needed
→ Writing doubles is difficult
→ Large number of doubles may required, especially in the lowest level of the system

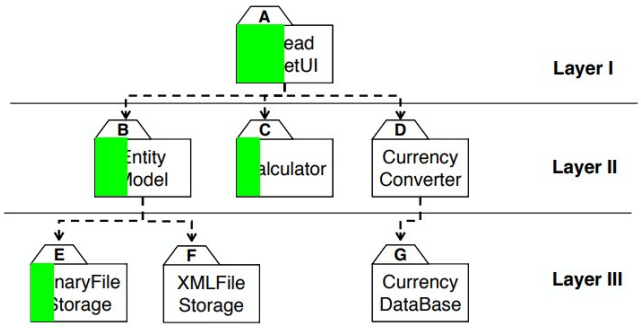
→ Some interfaces are not tested separately

Horizontal integration testing risks:

1. The higher the complexity of the software system, the more difficult is the integration of its components
2. The later integration occurs in a project, the bigger is the risk that unexpected failures occur

→ Vertical integration addresses these risks by building as early and frequently as possible

1. Used in scenario driven design: scenarios are used to drive the integration
2. Used in scrum: user stories are used to drive the integration



Advantages of vertical integration:

1. There is always an executable version of the system
2. All the team members have good overview of the project status.

System testing:
1. Functional testing: validates functional requirements
2. Structure testing: validates the subsystem decomposition
3. Performance testing: validates non-functional requirements.

→ Impact of requirements on system testing
→ Quality of use cases determines the ease of functional testing and acceptance testing
→ Quality of subsystem decomposition determines the ease of structure testing
→ Quality of nonfunctional requirements and constraints determines the ease of performance tests.

Functional testing: Essentially the same as black box testing
- test cases are designed from the requirements analysis document and centered around requirements and key functions
- The system is treated as black box

Performance testing: try to violate non-functional requirements

Acceptance testing: demonstrate the system is ready for operational use
→ Majority of all bugs in software is typically found by the client after the system is in use, not by the developers or testers
1. Alpha test: Sponsor or end use uses the software at the developer's site. → controlled setting
2. Beta test: Conducted at sponsor's or end user's site → software gets realistic workout in target environment, uncontrolled setting

Static analysis:

- Hand execution by the reading the source code
- Walkthrough by informal presentation to others
- Code inspection by formal presentation to others
- Automated tools that check for: syntactic and semantic errors, departure from coding standards
  → in Eclipse: Compiler warnings, metrix, SpotBugs, Checkstyle
- finds mistakes but some do not matter
- catches at best 5-10% of software quality problems

Dynamic analysis:

- Black box testing: test the input/output behavior
- White box testing: test the implementation of the subsystem or class

Black Box testing:

- Focus: I/O behavior. If for any given input, we can predict the output, then the unit passes the test.
- Goal: reduce number of test cases by equivalence partitioning:
  - Divide inputs into equivalence classes
  - Choose test cases for each equivalence class

White box testing:

- Focus: thoroughness(coverage). Every statement in the component is executed at least once
- Four types of white box testing
  1. Statement testing: tests each statement
  2. Loop testing: tests loops, number of executions...
  3. Path testing: makes sure all paths are executed
  4. Branch testing: ensures that each outcome in a condition is tested at least once

Comparison:

White box testing:

Potentially infinite number of paths have to be tested
Often tests what is done, instead of what should be done
Cannot detect missing use cases

Black box testing:

Potential combinatorial explosion of test cases
does not discover extraneous use cases

Both types are needed:

the are the end of the testing continuum
any test is in between these ends and depends on the following:
Number of possible logical paths
Nature of input data
Amount of computation
Complexity of algorithms

Observations:

- It is impossible to completely test any nontrivial module or system
  - Practical limitations: complete testing is prohibitive in time and cost
  - Theoretical limitations: e.g. halting problem
- “Testing can only show the presence of bugs, not their absence”
  →Define your goals and priorities.

Testing takes creativity:

1. Detailed understanding of the system
2. Application and solution domain knowledge
3. Knowledge of testing techniques
4. Skill to apply these techniques

Testing is done by independent persons:

- Developers often have a certain mental attitude that the program can behave in a certain way when in fact it does not
- Developers often stick to the data set that makes the program work
- A program often does not work when tried by somebody else.

Test model: consolidates all test related decisions and components into one package

- contains a test driver, input data, the oracle, the test harness and the test cases
  - test driver: program that executes the test cases
  - test case: a function usually derived from a use case
  - input data: consists of the data needed for the test cases
  - oracle: predicts the expected output data
  - test harness: frame work that runs the tests

Execution of test cases:

1. Manually: testers set up the data, run the test and examine the results themselves.
2. Automatically: running the test and compare the results against the oracle automatically
   1. everything tested automatically with a test harness
   2. Less boring for the developer
   3. better test thoroughness
   4. reduces the cost of test execution
   5. indispensable for regression testing.

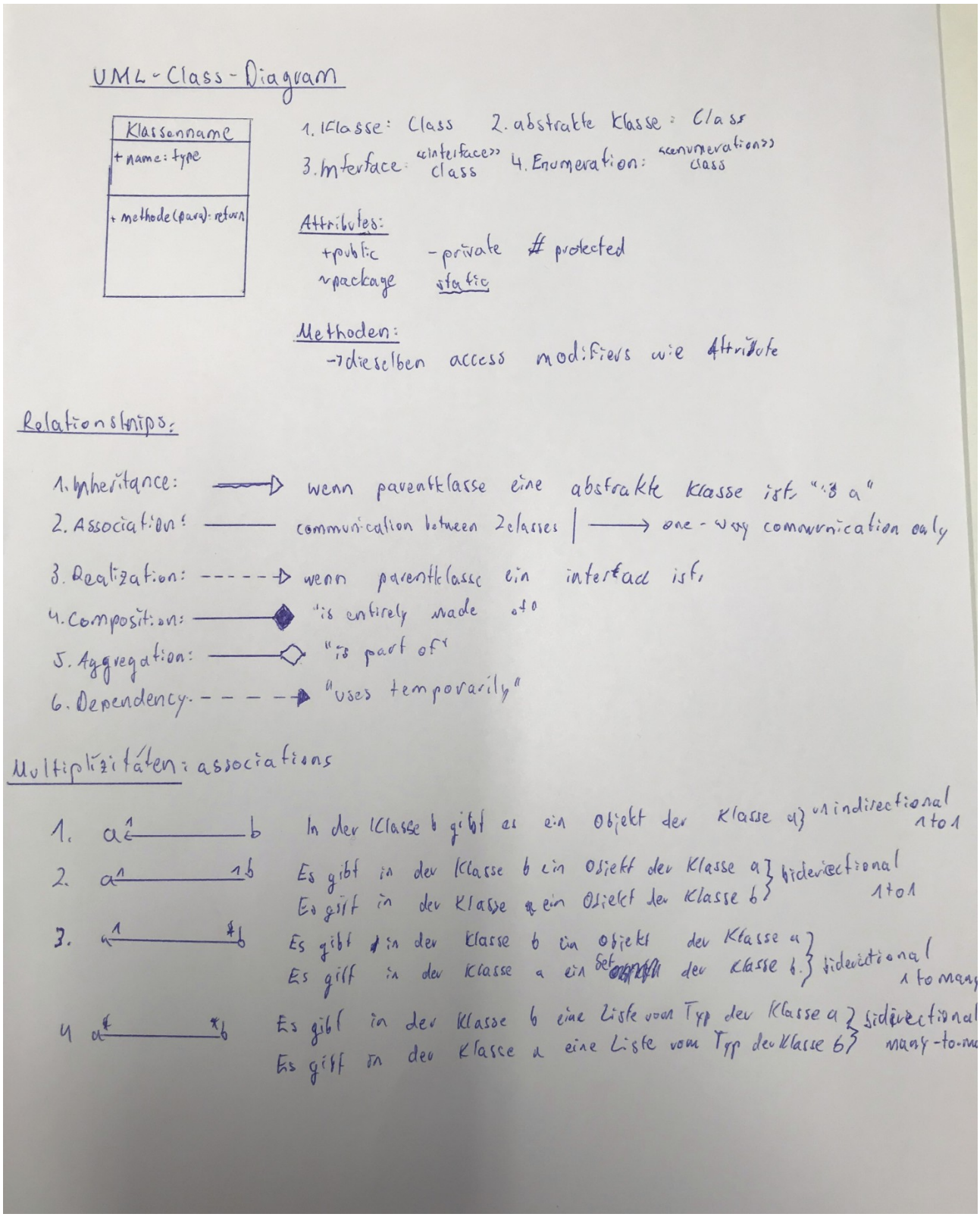
Model based testing:

- The system model is used for the generation of the test model
- Extreme programming variant:(XP)
  The test model is used for the generation of the system model
- System under test (SUT): The part of the system model which is being tested

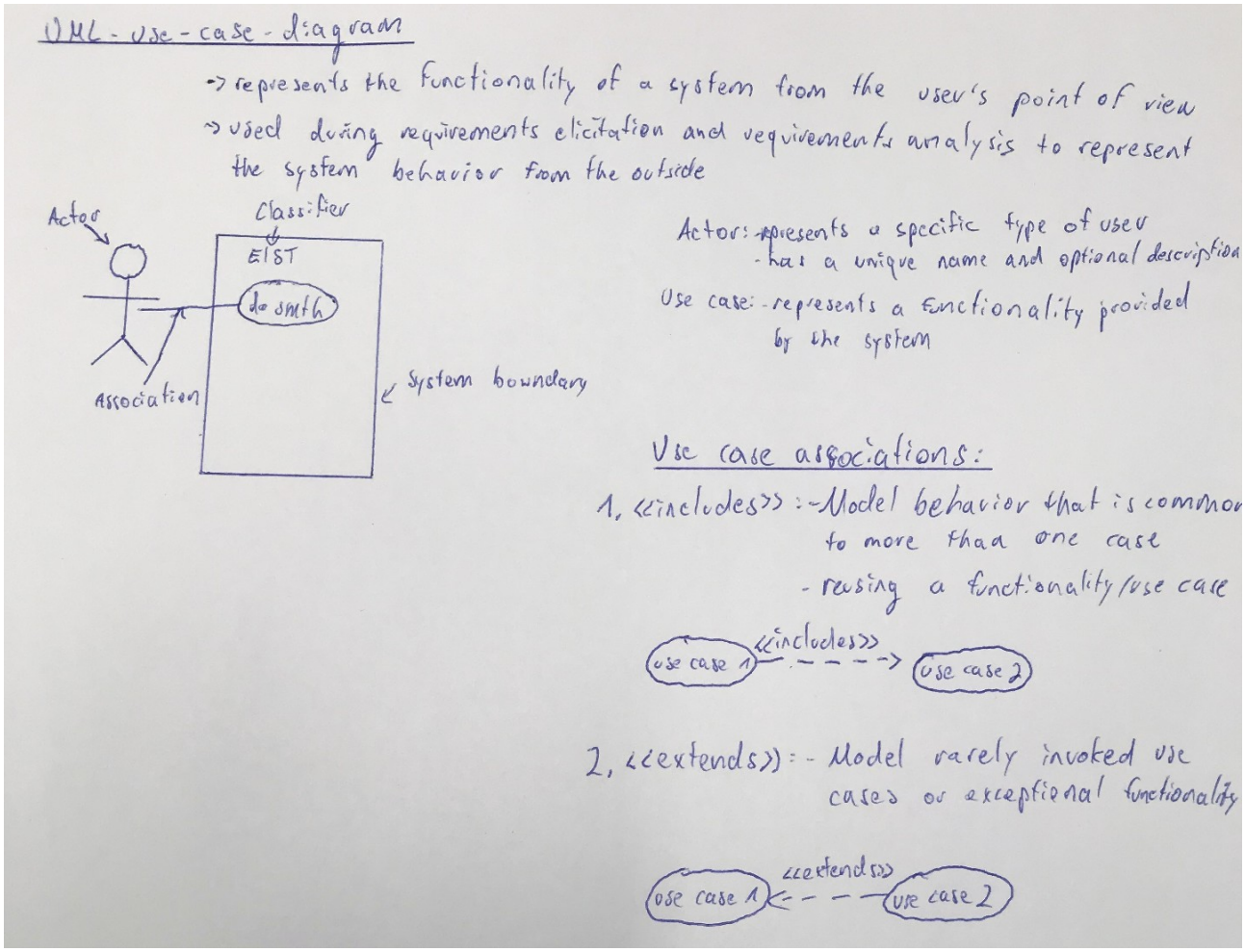
→ Mock Object Pattern

UML-Models/Diagrams

1. UML-Class-Diagram



2. UML-use-case-model



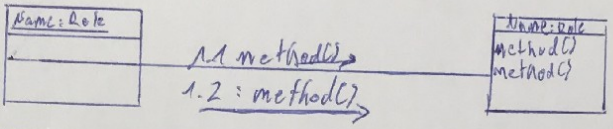
UML-communication-diagram

UML-communication-diagram

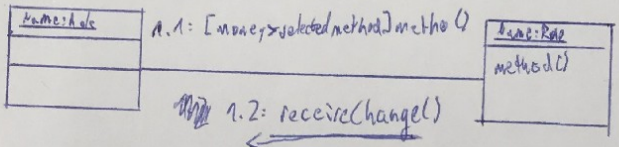
- -> visualizes the interactions between objects as a flow of messages.
- -> describe the state structure + dynamic behavior
- -> reuse the layout of classes and associations in the class diagram
- -> Messages between objects are labeled with a number and placed near the link the message is sent over
- -> describe the interaction between different objects by using method invocation
- -> shows message flow additional to classdiagram and associations

Types of messages:

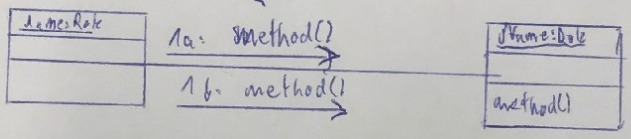
1. Sequential messages:



2. Conditional messages:



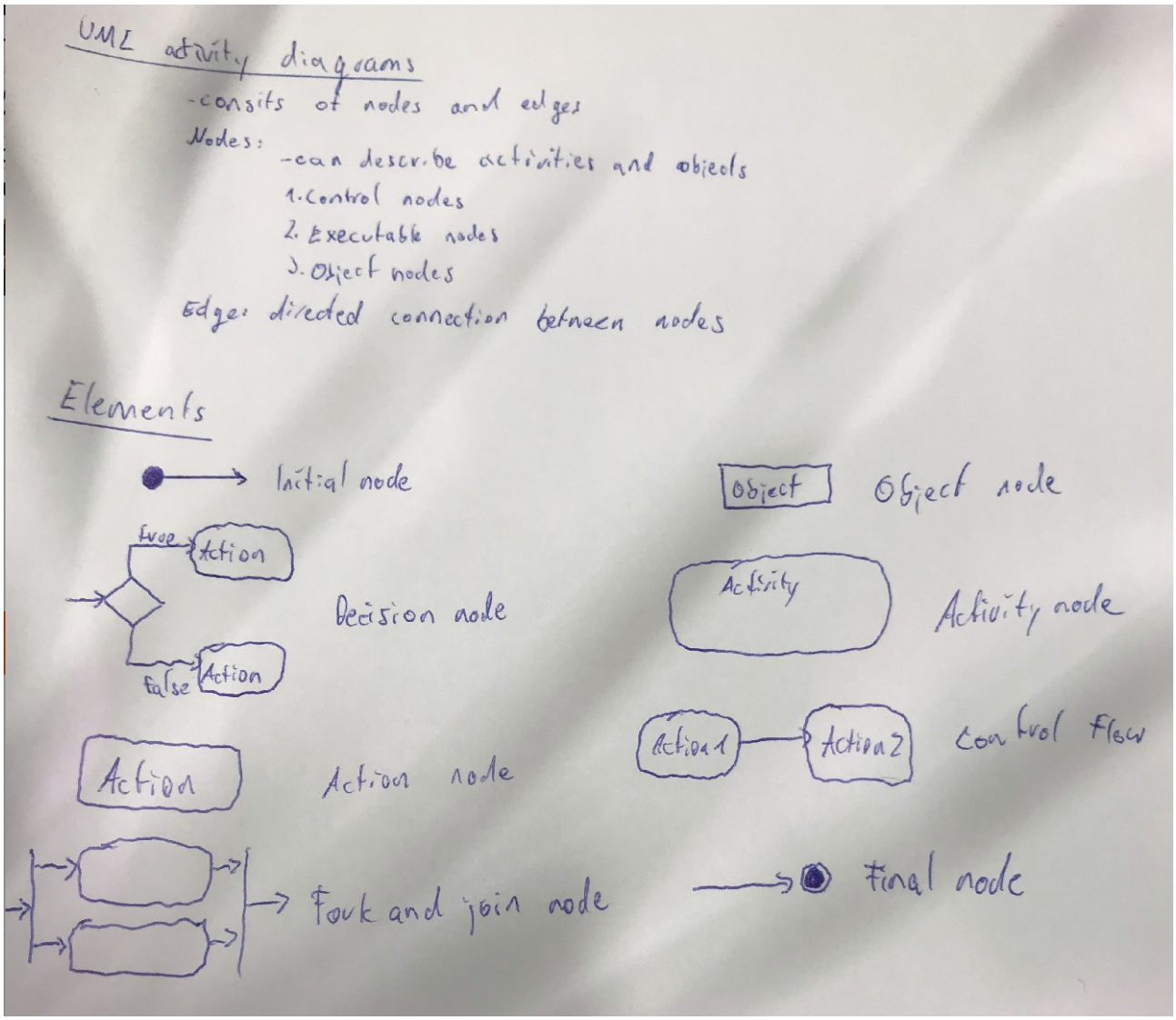
3. Concurrent messages:



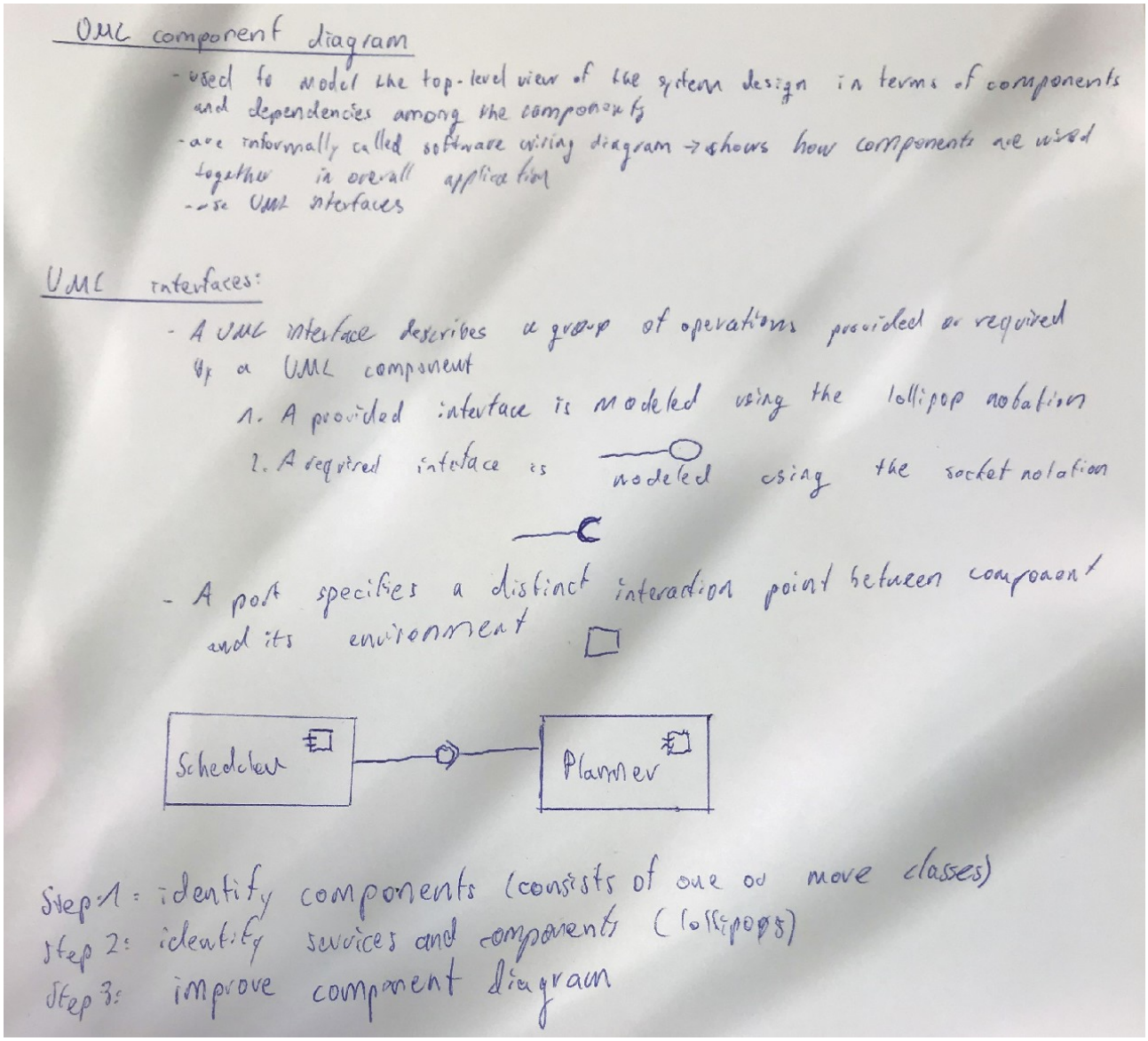
Recipe from class diagram to communication diagram:

1. Take all steps from the event flow of a use case
2. Instantiate the participating objects
3. Number the messages from each of the steps of the event flow
4. Is there a corresponding method in the receiver of the message?
5. Draw the message from sender to receiver

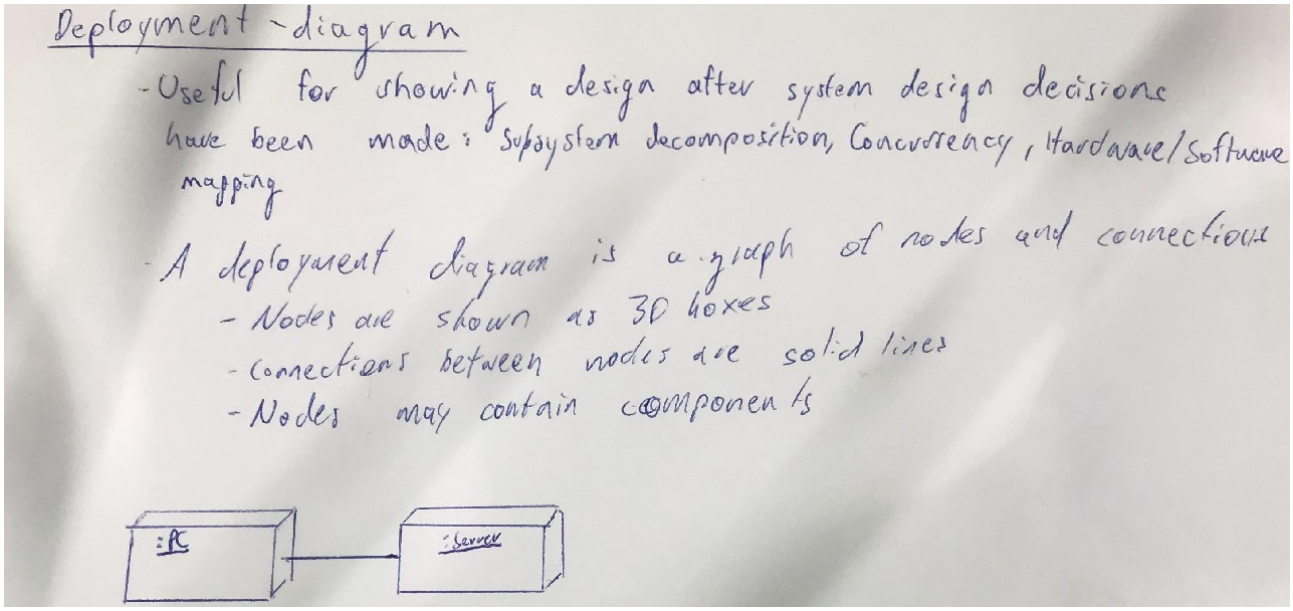
UML-activity diagram



UML-component-diagram

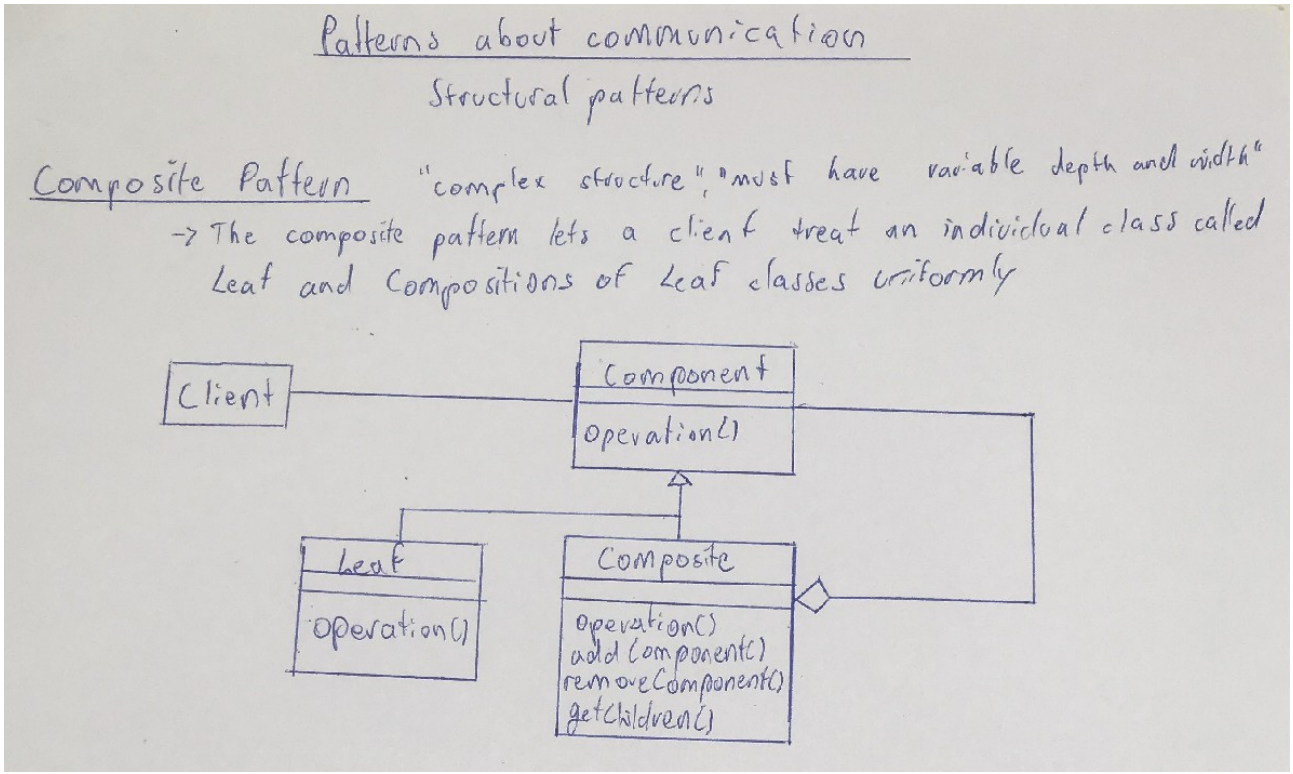


UML-deployment-diagram

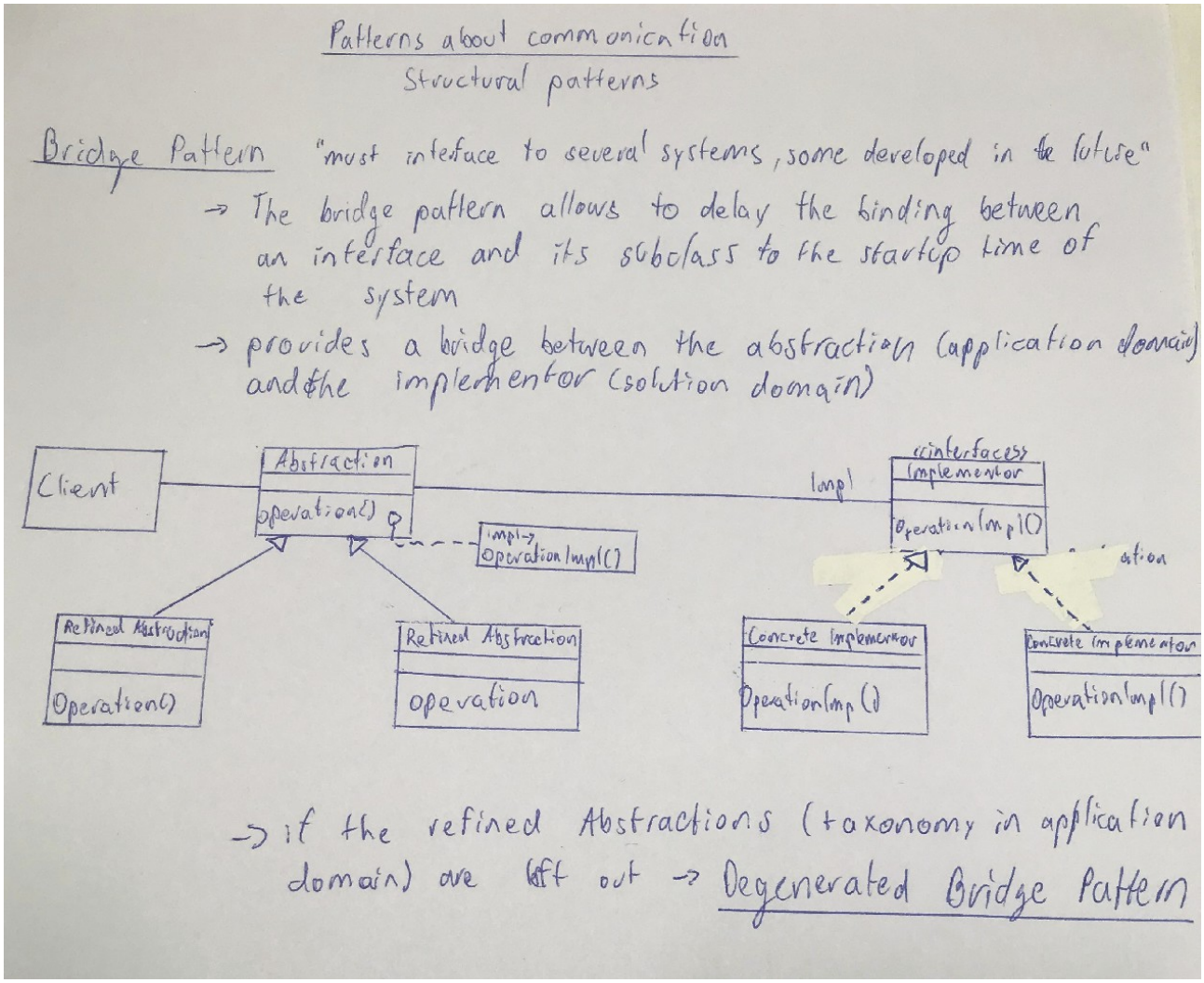


Patterns

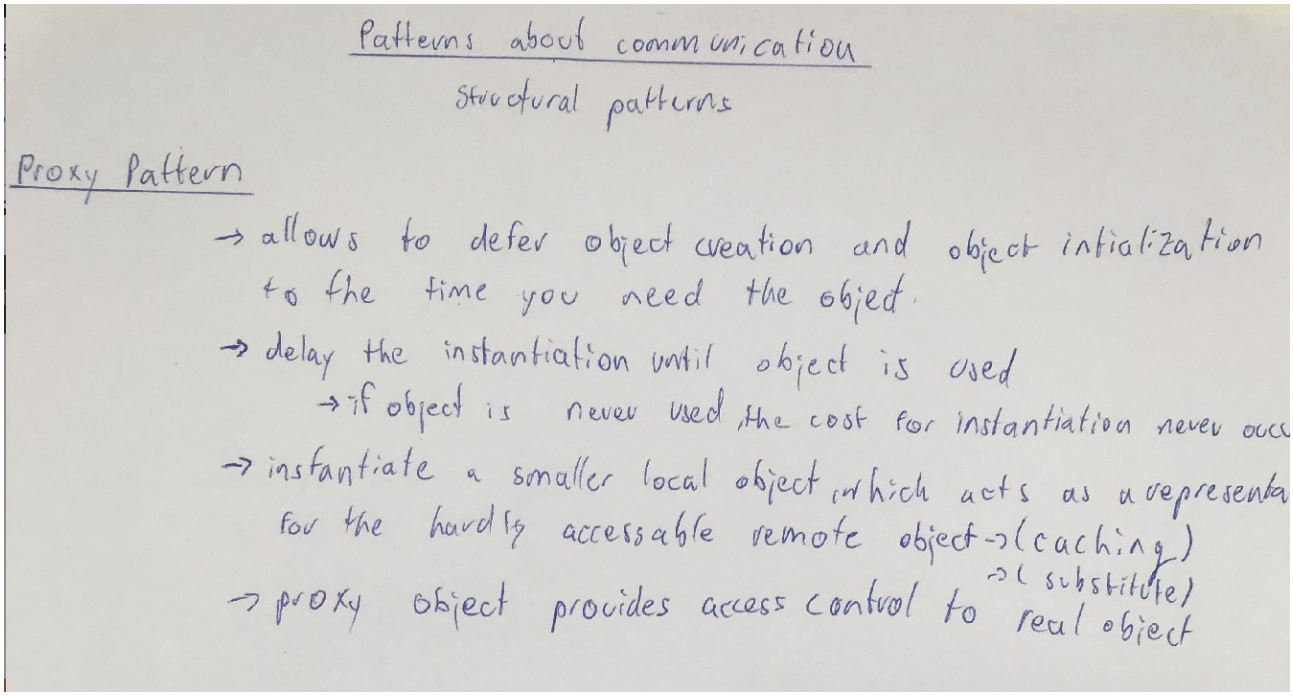
Composite Pattern



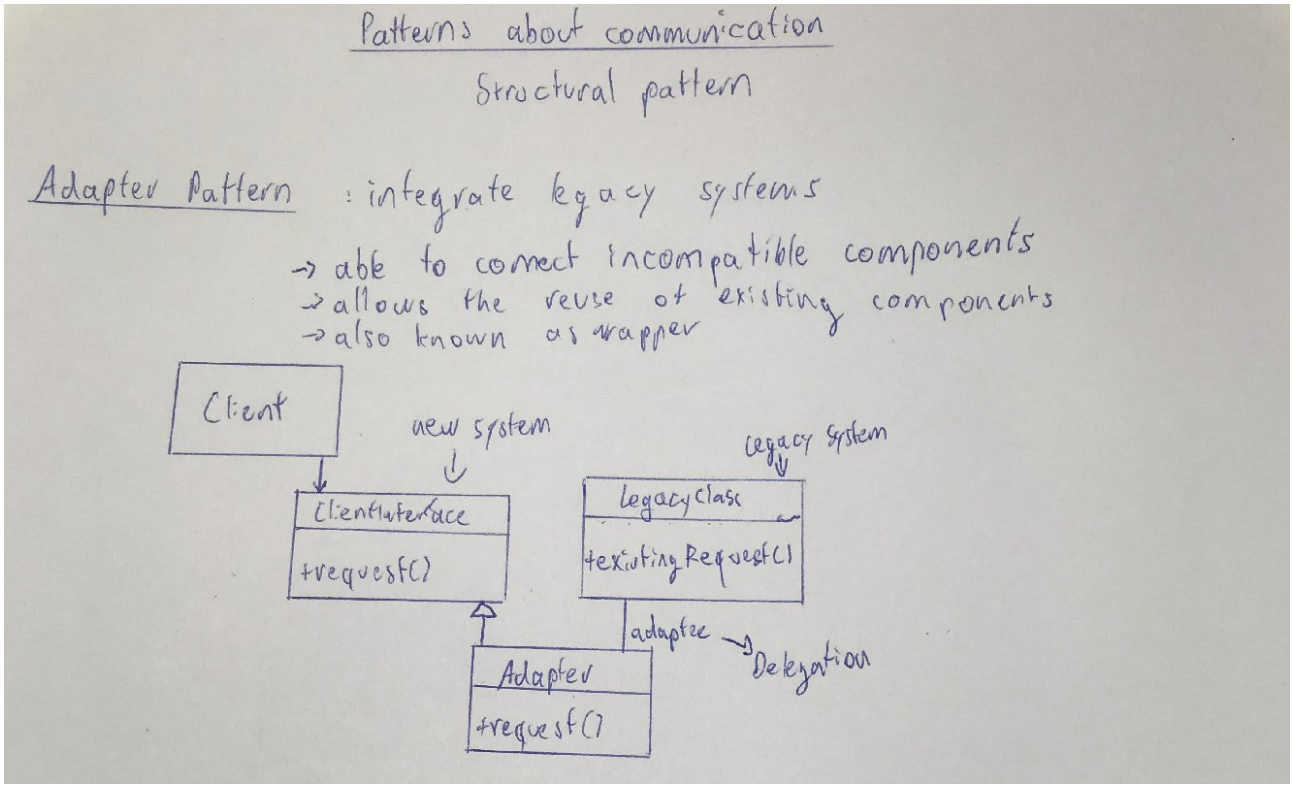
Bridge Pattern:



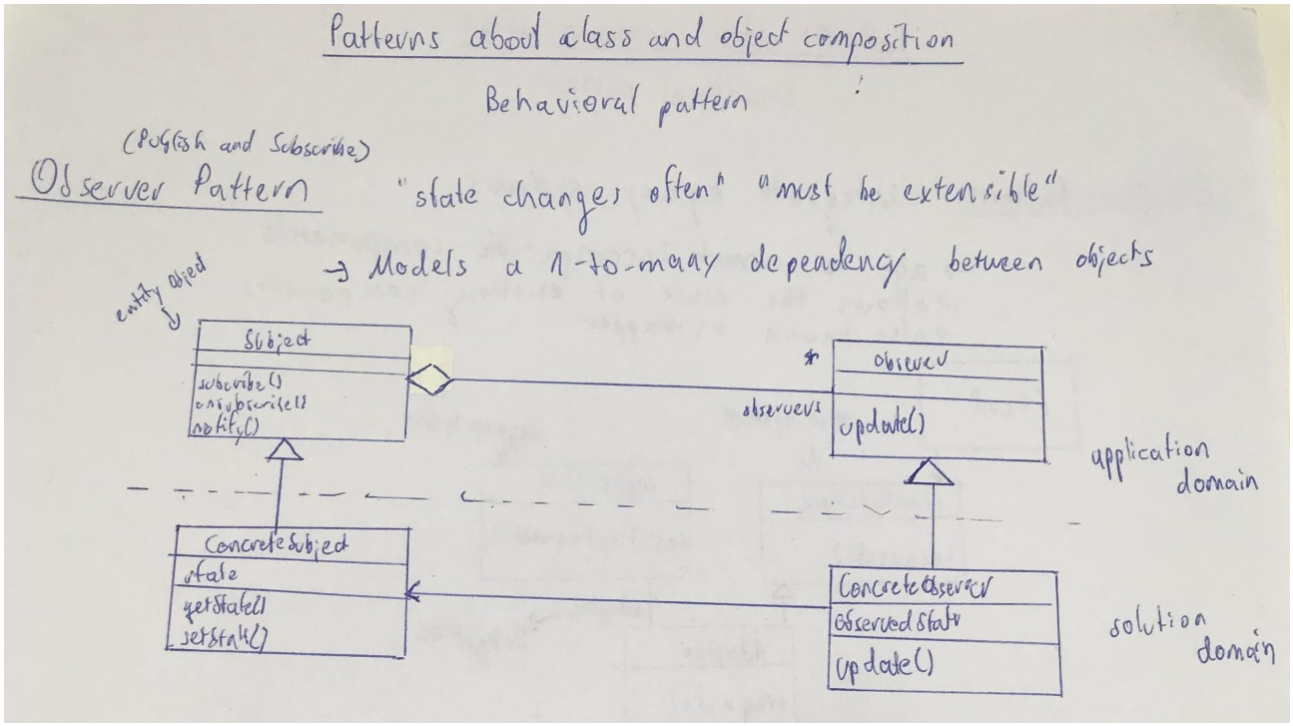
Proxy Pattern:



Adapter Pattern:



Observer Pattern:



Strategy Pattern: