SOFTWARE ENGINEERING

Feasibility Study

- Technical
- Financial
- Operational (security, safety, usefulness, performance...)

Heterogeneity ⇒ different platform techniques

~~Software process Phases~~

Proben definition

Feasibility Study (Olabilirlik) ⤻ Technical, economic, operationally

Requirement analysis → SRS (Software requirements specifications) ((Not) Functional specifications) Result document

System modeling

Component → Component structure, modules to implement, database and file format

Implementation ⤻ Unit Test

~~Testing~~ → ~~Verification, validation, documentation is finalized~~

Deployment*

maintenance

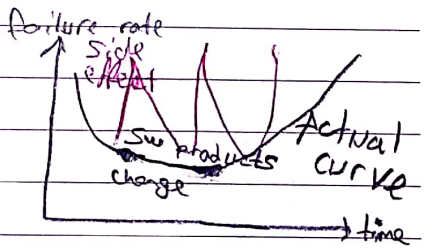
Return on investment = $\frac{\text{Total Benefits} - \text{T. costs}}{\text{Total Costs}}$

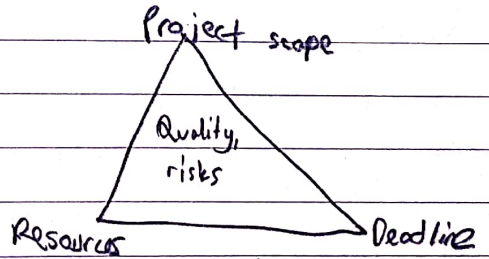
Example of economic ... den önceki sayfa midterm'de

cost = revenue

Break even Point = $3 + \frac{\text{Sum} \dots}{9}$

	2012	2013	2014	2015	2016
Benefits					
Increased Sales					
Reduction in Customer Complaint Calls					
Reduced Inventory Costs					
Total Benefits					
Development Costs					
2 Servers $125k	250 k				
Printer	100 k				
~~Sw licenses~~	34,825				
Server Sw					





Each software process step ends with verification validation (V & V)

Requirement ⇒ System analysist performs ↗ funct. ↘ non-funct.

System
- partition (vertical)
- layers (horizontal)

⇒ Heterogeneity
⇒ Development speed
⇒ Trust

Good SW Product Characteristic ⇒ Functionality, Maintainability, Reliability, Efficiency, Acceptability

Midterm

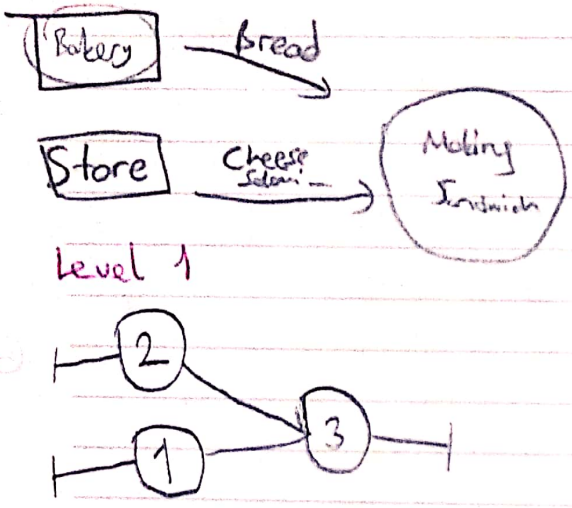
Use case descriptions

1. part ⇒ Require diagrams
additional diagrams
requirements spec.
Functional, not functional (practical part)

2. part ⇒ (theoretical) comparison
explanation
Not so details
Advantages, disadvantages

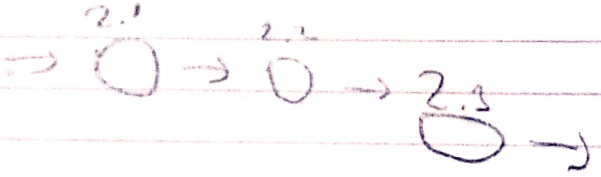
Data Flow Diagram (DFD)

Level 0



Level 1

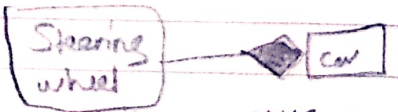
Level 2



UML (Unified Modeling Language)

Relations Class Diagram

Aggregation (has a)



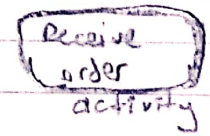
composition owns a

○ → Interface port a component needs for its operation.

● → start

◉ → end

Activity



SRS

- Testing (validation)
- design
- maintenance
- finding agreement with customers/users

SRS ⇒ Software Requirements Specifications

- Introduction
  - problem description
  - description of the idea
- Use-cases
  - Use case diagram
  - Use case description + Additional diagrams → (performance/functional requirements)
- Interfaces
  - GUI
  - API

Recursion (Downtime)

Shall be made by;

- customers
- management
- developers
- sys. analyst
- testers

control from an individual (using control lists)
Group review
Supervision

Models

Linear (waterfall) ⇒ The working SW available at the end.
Revision of previous step.

V Model ⇒ In each phase, there is test.

Spiral Model ⇒ Each cycle, includes phases of linear model. / Longer time
" " , prototype develops.
↳ Definition of goals, alternatives
Evaluation, risk identification
Development of current cycle product.
Planning of next cycle.

Phased development > Throwaway prototyping
↳ The initial system with limited functionality deploys sooner.

* Unified Process Model Framework ⇒ UP, RUP
↳ Inception (Başlangıç)
Elaboration (Ayrıntılandırma)
Construction (Yapım)
Transition (Geçiş)
* Use-case model
↙ Use case diagram ↘ document that describes it.

Diagramlar
* Flowchart (Türkçesi nasıl?)
Data flow diagram (DFD) veri akış şeması
Entity-Relationship diagram (ER) → database tablo diagramı
UML (Unified Modeling Language)

Diagram (and model) Types
Context
Behavioral
Structural
Domaining (model) Domains
Application Domain
Solution Domain

↳ Start, and Process

Entity Relationship Diagram
Student — entity/set
— optional
relation
name — Attribute
Student id — key

Object Diagram
myStuff
id = 156
name: negmed

Use Case Diagram
Actor
system
<<extends>>
use case
<<includes>>
Senaryolar

Sequence Diagram
Synchronous message
Asynchronous Message
Return message (response)

Goal of Analysis

Problem Domain (The real world)

Implicit model (people) → Analysis → Explicit model (Specification)

Requirement Management
- Identification
- Tracing
- Prioritizing
- Validating
- Communicating

SRS ⇒ Software Requirements Specification (SRS)
↳ To define basis for system design, design, validation.

Requirements Collection (important)
↳ How to define system scope.
↳ Understanding
↳ What if changed?

Properties of system environment should be used.
If requirements cannot define clearly, use prototypes.

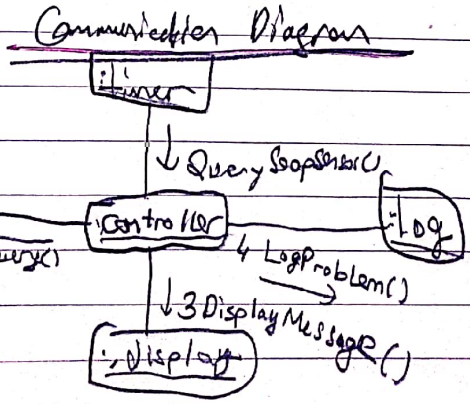
Requirement presentation Models
- Natural languages
- Mid-formal notations
- Formal notations

Requirement Specifications Models
- Data
- Process
- Execution (control) model

Requirements Modeling

Structural Analysis
- ERD
- DFD
- Activity diagram or Flowchart
- ELH
- List of events
- Contextual Diagram

Using UML (OOA)
Use case + other diagrams
(+description)



SRS Reversion
- Validity
- Consistency
- Completeness
- Reality

(System requirements)

- Testability
- Understandability
- Traceability
- Adaptability

(Individual Requirements)

Control flow

SRS Supervision
Goal is finding errors.

class diagram
italik = abstract
design pattern

Use Case Description Contents

Use case name
Actors
Requirements ⇒ ~~Customers, operations [illegible]~~
Trigger
Basic (Main) Flow ⇒
Post Conditions
Alternative Flows → 1a
2a
2b

Peer to peer is hard because of deadlock threat. (Alternation client/server)

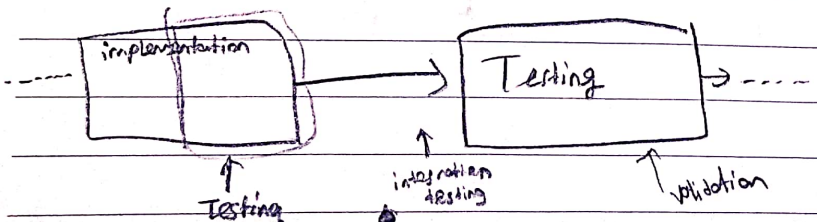
Pipes and Filters

ps (process status)
grep (search for pattern)
sort (sorting)
more (display at the terminal, one screen at a time)

Architectural Plan Structure

- Purpose
- Priorities considered
- Summary of the architectural plan
- Description of the proposed architecture
- Other details of the architectural plan.

(structured)
Observer ⇒ notify() function observe changes



verification / Unit testing

code Review

Testing

verification ⇒ follow input document
validation ⇒

→ Unit test

whitebox (structural)
black box (behavioral)

Deviation (sapma)

Path testing is in

M2

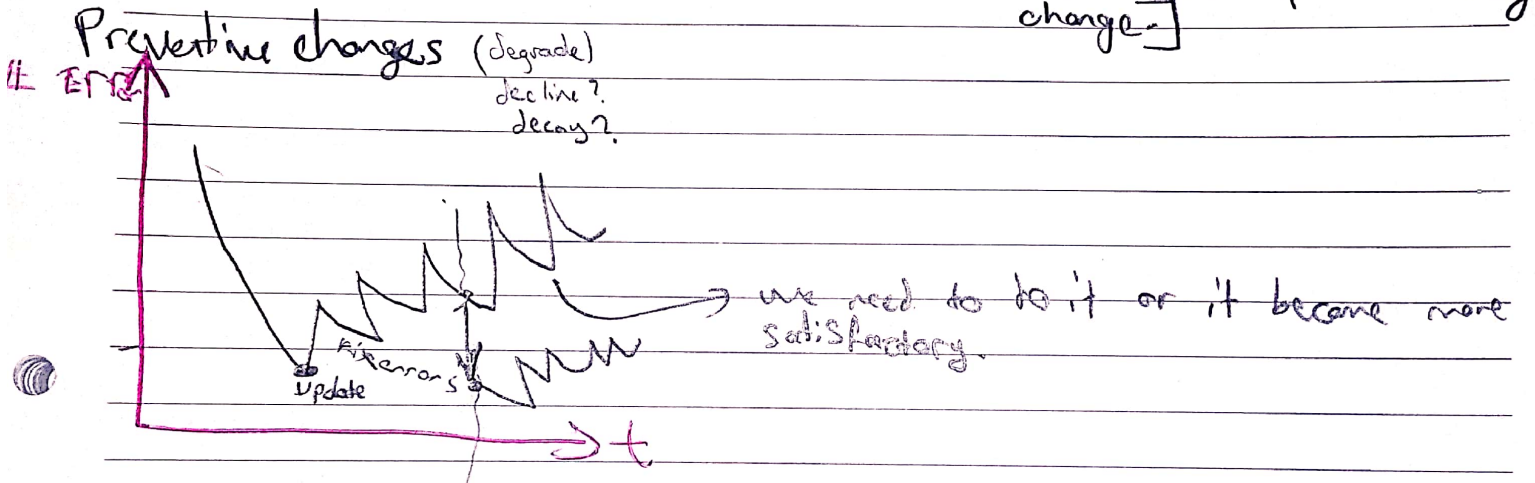
development environment
testing
operation and maintenance

Later, ~~not~~ difficult to modify so maintenance costs.

Maintenance is last resort

Definition of problem
feasibility study
↓
Not exactly part of project
Project starts with requirement adysis.

After operation and maintenance, then we see evaluation and completion.
Then project finishes.
[Maybe later, requirements may change.]



~~System design~~ model divides the system into subsystems (components), describe their behavior, their relationship and interfaces.
↳ includes all decisions influence on structure of whole system (Software architecture)

Stable architecture assures reliability of system and easy maintenance.
↳ so add functionality with min arch. changes.

- Definition of design goals → performance, dependability, cost, maintenance, user
- Decomposition of system into subsystems
- selection of reusable components.
- Relation between SW - HW
- Selection of infrastructure for persistent data management.
- Definition of the access control policy.
- Selection of global control flow mechanism.
- Handling borderline cases.

Allows to find requirements

Layers separate subsystems hierarchically, vertically.
Partitions separate subsystems of the one layer, horizontally.
Layer provides services to upper layers and depends only on the lower layers.

Closed Architecture
Layer uses only the services of the first lower layer.

Open Architecture
The layer can use the services of all the lower layers.

Layers can be replaced without affecting others.

- UI should have separate layer. This layer provides functionality defined by use-cases.
- The lowest layers provides network communication, database access, general services.

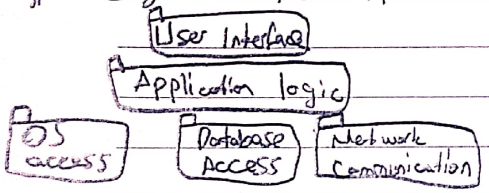
AGILE METHODOLOGIES

Criteria for decision of methodology selection
Time available (more time - spiral)
Resources (more prototyping, spiral)
Small number of developers
Quality of developer team
Quality requirements

Formal Methods

Like	Dislike
	A lot of controll ATs
Do some programing	
Don't like to write documents (reduce documenting / meeting in person)	
Inadvert of multiple phases (do multiple type of tasks)	

Typical Layers Example in APP



It's desirable that subsystems are weakly connected.
This reduces performance impact to other subsystem.

Subsystem Coupling Types

Content, Common, Control, Stamp, Data, Routine Call, Type Use, Inclusion/Import, External

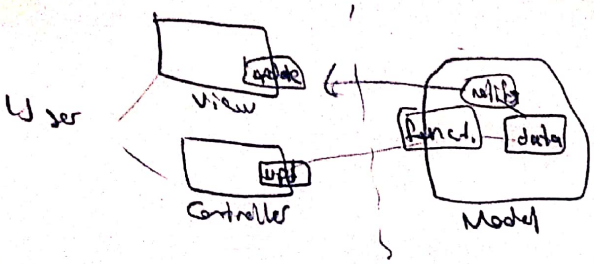
Cohesion of Subsystem (Internal relations)

Functional, Layer, Communicational, Sequential, Procedural, Temporal, Utility (worker than sequential (better))

Global access table

Specifies the rights for the relation
Global Control Flow Mechanism
- Procedure-driven control → waiting input
- Event-driven control → waiting for an external event
- Threads → most intuitive (seagisel), difficult to debug

Pattern
Each pattern describes a problem which occurs over and over again in our environment.
Documented solution of a reoccuring problem
Each architectural pattern enables designing flexible system architectures. (to consist independent components)



Architectural Patterns

- Layers – Communication is through API. Independently tested layers.
- Model/View/Controller – Separate UI Layer from others.
  Model: Combines all domain knowledge and operation
  View: Provide info to user.
  Controller: controls user interaction and transmit data into
- Shared Repository – Subsystems communicate through central data structure. Multiple subsystems access data from a shared da
  words
- Microkernel – Different versions can offer different functionalities. All of them use common functional kernel (microkernel) (DBs
  Typical
- Client/Server – Distributed system that servers provide service to client ss. Client request is sent with remote procedure call mech
  Control flow is indepen
- Peer-to-Peer – SS can act like server or client. Control flow is independent except accesses synchronization. (deadlock, thread, make difficult)
- Pipes and filters – SSs process the data; obtain from other systems with inputs, pass with inputs.
  SSs → filters, associations between them are pipes. Each filter process data in its own way.
  Architecture is flexible; filters can be omit, replaced, added. Order can be change. Suitable arch. for data flow transformation with user interact
  (Not suitable for more complex)

Client-server architecture is generalization of the "Shared repository".

Documenting forces important decisions to be made before implementation begins. + Enables communication

Architectural Plan Structure
Purpose / Priorities considered / Summary of Architectural Plan / Description of proposed arc / other details
→ with who will build system
→ with who will change architec
→ with who will build other systems will connect to planned one.

Recension
Architectural Plan Recension doesn't include external reviewers (clients, users)

Component Design (Objects)

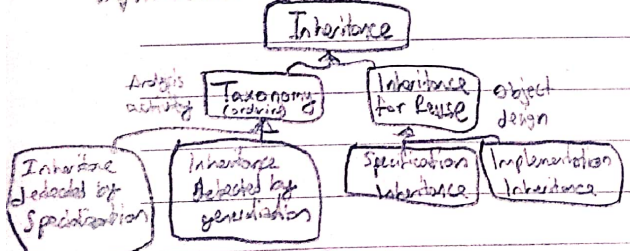
When to?
Planning of reuse the existing components (libraries, design patterns.)
Filling the gaps between app objects and selected existing components.
→ Improvement on object
→ Interface specification
→ Optimization of sys design (responsiveness, memory)

OO Component(object) Design
- identifying existing components
- Specifying interfaces
- System restructuring
- Optimization

Objects communicate by messages like getter, setter.

Delegation

Make sure that delegate is properly used preventing inappropriate usage.



If dependencies are less; easy for maintenance.
Connascence: measure of software dependencies.
Pattern doesn't require knowledge of both

Liskov Substitution Principle
If class S is a subclass of class T; then objects of class T may replace objects of class S without altering properties.

Encapsulation: Separate contract interfaces from implementation.

Modularity:
Modular program consists units communicate through interfaces.
Module is any unit of program consists of several parts. (program, subroutine, package, class, operation, line of code)

Principles: Small modules, info hiding, min privilige, coupling (connascence), cohesion

Depending on the scope
Class : (static)
Object : (dynamic)

Design Patterns

- Pattern name
- Problem
- Solution
- Implication

Purpose Depending
- Creational: Process of creating objects
- Structural: Determine the composition of classes and objects
- Behavioral: Determine way classes interact / distribution of responsibility

Creational DP
Abstract factory: Groups object factories that have common theme.
Builder: construct complex objects.
Factory Method: Create object without specifying exact classes to create.
Prototype: Cloning existing object and create.
Singleton: Restrict object creation for a class to 1 instance.

Structural DP
Adapter: classes with incompatible interfaces can work together. (If interface doesn't match)
Bridge: decouple an abstraction so 2 can vary independently (2 ye ayirma)
Composite: Composes similar objects so manipulated as 1 object.
Decorator: Dynamically override behavior in existing method or object.
Facade: provide simplified interface to a big code.
Flyweight: Reduce the cost of creating similar objects.
Proxy: provides a placeholder for another object to control access, reduce cost.

Behavioral Design Patterns
Chain of Responsibility: delegate commands to a chain of processing objects.
Command: creates objects which encapsulate actions and parameters.
Interpreter: implements a specialized language
Iterator: Access elements sequentially.
Mediator: loose coupling between classes by being only class that knows about them.
Memento: Provides undo.
Observer; publish/subscribe pattern; so when an object changes, the others notified.
State: Allow object to alter its behavior when internal
Strategy: Allow one family algorithms selected on the fly (during runtime)
Template Method: Defines skeleton of algorithm as abstract class.
Visitor: Seperates algorithms from object which they operate.

Documenting: Component Design
A component design must define:
The intend structure of all components
Their dependence (interfaces)
Their structure (operations, attributes, interfaces, compositions)
Processing methods
Algorithms
Data structures
Properties (Func./Nonfunc, throughput, reliability)

Interface specification
Syntax → defines the elements of communicating medium.
Semantics → Specifies the meaning of messages.
Pragmatics → How messages are used to perform tasks.

Defect Distribution
Requirements 15%
Design 12%
Code 7%
Others 10%

The individual components are tested for compliance with component specifications (component design)
The individual components are integrated into software system and tested accord SS (system design)

Verification: Checks consistency of outputs with input parameters.
Validation: Check consistency with user needs.
Executed by following a prepreparated plan that follows SRS.
Tests the compliance of the system with the requirements. (functionalfunc)

Reconsider / Review
Systematically scan documentation to find possible errors.
Involves a meeting with author although reviewers can do separately individually.

Reviews should be constructive and positive.

Documenting Reviews
Start with a resume and be positive.
Use electronic mechanism to record comments.
Make an agreement that author doesn't need to answer all questions.

Reviews vs Testing
They are based on different ways of verifying a product.
When using both, chances of error are smaller.
Reviews should take place before any testing.
Reviews allows you to find bugs that affect maintenance possibilities and performance.
Testing makes it possible to look for errors in complex solutions and have clear consequences.

Testers perform series of test cases.
A test is successful if unknown error is found.
A good test case is one that likely reveal an error not discovered.

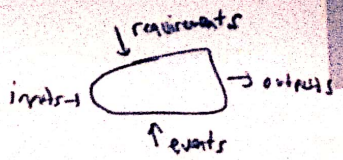
Testing Principles
All test cases should be related to the requirements.
Testing should be scheduled before it begins.
When testing Law 80-20 (Pareto principle) applies.
(i.e. 80 of all errors detected during testing originate from 20 of the components.)
Comprehensive testing is not possible.
Independent testers perform the most effective testing.

Good SW Testability
Operability, observability, controlability, decomposability, simplicity, stability, understandability (including documentation)

Testing methods

White Box Testing (Structural Testing)
Based on knowledge of the internal structure of the program.
Purpose is to ensure each condition satisfied and every part of code executed at least once.

Basis Path Testing: Basis path contains at least one new piece code / condition.
Number of independent paths depends on the cyclomatic complexity. (Measure of logical complexity. Related to errors)
Testing nested loops: Start with innermost loop. Execute with least iteration.



Behavioral Testing (Blackbox)

Blackbox testing is used in the late stages of the testing process.
It doesn't focus on the control flow but the processed information.
Based on the declared external properties of the system.

Test case planning questions

How to verify functional correctness?
How to check system behavior and throughput?
What input form good test cases?
Is system sensitive to certain input values?
What are the boundaries of data classes?
What range of data can the system tolerate?
What impact do specific combinations of data have on system?

Equivalence Class ↓
Group of valid/invalid input conditions.

Boundary Value Analysis (BVA)

Input conditions for test cases between $a \leq < b$
Testing with BVA is also about output.

Other Behavior Testing Techniques: Error guessing method, decision table techniques, cause effect graphing

Testing is from the bottom up (component level - class, object - to larger modules)

Unit Testing

Testing individual components or units. Prepare a test driver.

Integration Testing

Test component group that are integrated into a system.
Based on system specifications (requirements, system design) — behavioral testing (blackbox)

Incremental Integration

Top down integration
Bottom up integration
→ Often a combination of both approaches.

Validation Testing

Testing if meets reasonable expectation of users.
If deviations found, should documented in deviation list.
Configuration review: Ensures all elements of SW product are properly developed, cataloged and fit maintenance phase.
Alpha Testing is performed on developers side in a controlled environment in the presence of users.
Beta Testing: Performed by clients without presence of developer.

System Testing ⇒ recovery testing, security testing, stress testing (test excessive loads), performance testing.

↳ Shows undetected errors. Important on distributed systems.

Debugging: Manifestation of error and cause don't necessarily have an obvious connection.

Report: Test results may depend on the environment, so test reports should include an identification of environment (OS, database version, related systems)
Result may depend on other SW running concurrently with the system tested.

MAINTENANCE

After acceptance of SW product, customer may perform acceptance test. Can be done by

Deployment: At developer (factory acceptance test - FAT)
at customer (site acceptance test - SAT)

- Documentation is intended for 2 target groups: System maintainers and system users who use system
- Customer support

Installation

Direct Installation, Parallel Installation, Installation at Specific Location, Phase Installation
↳ There are 2 locations ↳ Kademeli geçiş (gradual)
first replace one func. component until new system is fully installed.

Consider about data conversion of existing system, to be installed and try.

Maintenance in general; functional checks, servicing, repairing/replacing components and all other activities that keep product in operation.

Maintenance cost is 1-100 times higher than development cost.

↳ Application type, system uniqueness, personal fluctuation, experience, system lifetime, variable environment dependency, hardware characteristics, design quality, source code quality, documentation quality, testing quality, use of tools, number of users.

Maintenance Includes 4 types of Software Changes

1. Corrective Changes - Error correction
2. Adaptive Changes - Adaptation to changed conditions
3. Perfective Changes - Improvements of the SW product
4. Preventive Changes - Elimination of SW degradation. (degrade)

Software Evolution

What's Evolving?

Requirement evolution ↳ requirements specification → must be synchronous (Co-evolution problem)
Architecture evolution ↳ Design
Data Evolution
Source code Evolution
Documentation Evolution → Implementation
Technology used Evolution
Testing case Evolution → Testing

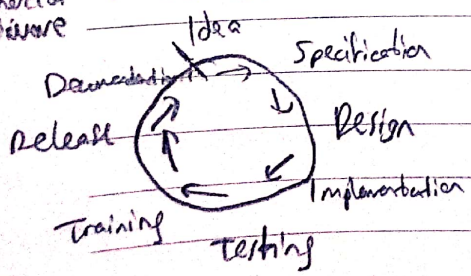
S-System: don't require change
P-System: Requirements based on applied solution. They need incremental adaption.
E-System: Require constant adaptation to real world. (most SWs)

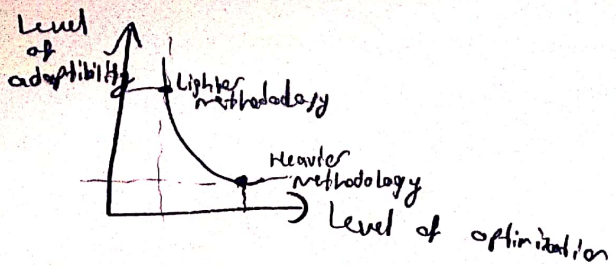
Lehman's Laws of SW Evolution

Law of continuing change, Law of continuous growth, Law of increasing complexity, Law of declining quality

fits with commercial software

Maintenance Process





Engineering

Forward Engineering: Traditional process that runs from high level of abstraction and logical execution-independent design to physical execution of system.

Reverse Engineering: Analyze system to identify and represent it with higher degree of abstraction.

Reengineering: Conversion of system to reconstruct and implement to new design.

Rationale (Gerekçe) → Answers why and how.

↳ Includes issues, alternatives, decisions taken to resolve issue, criteria led decision, discussion needed in team.

AGILE APPROACHES

Some phases are performed informally to increase process flexibility, shorten delivery time.

Individual relationship is more important than processes and tools.

Running software is more important than extensive documentation.

Responding to change is more important than strictly following plan.

Simplicity, persad communication.

XP (Extreme Programming)

Bring team together with enough feedback to see where they are and make situation = practice.

If something is useful do it together / do it planning daily.

Customer representative, no specialization from developers.

XP Principles

Planning game, Small releases, metaphor (hikaye anlatı), simple design, testing, refactoring, pair programming, collective ownership, continuous integration, 40-hour week, onsite customer, coding standards

SCRUM

Pig Role

- Scrum Master: Ensure smooth execution of process (role of project manager, not team leader)
- Product owner: Represent interest of customer.
- Team: 7 (±2) members whose tasks is to analyze, design, implement and test.

Chicken Role

- Stakeholder: Customers, users
- Management: Establish condition for project implementation.

Owner of product arranges backlog (binlerce if wish list features) → 4-12 parts. Each part 4 weeks to complete.

↳ Sprints (koşturma)

related part → sprint backlog (smaller than product backlog)

Sprint Planning Meeting: At start, choosing work scope, prepare sprint backlog for time

Daily Scrum: Everyday during sprint period about status of sprint - only pigs have word.

Scrum of Scrums: If other daily scrum; each team represented by 1 team.

Sprint Review Meeting: Review of completed (even unfinished) work, presentation to stakeholders.

Sprint Retrospective: All team member present their POV to sprint finished.

Burn-down chart: Shows remaining work required updated daily, easy insight to progress.

Disadvantages

Extensive documentations are missing.

Not everything is testable.

Difficult to track fast changing requirements.

Continious code refactoring create errors.

Agile approaches cannot completely replace formal approaches,

Agile-formal approaches can be complementary.

Modularity: easier do; understand, interpret, modification, testing, debugging, better reusability, easier to set up

Modularity Principles: Small modules, info hiding, min privileges, coupling, connaccenct, cohesion ↑

Contract

Assumptions, Obligations, Work, Compensation, Conditions for Modification of Contract

Maturity Levels

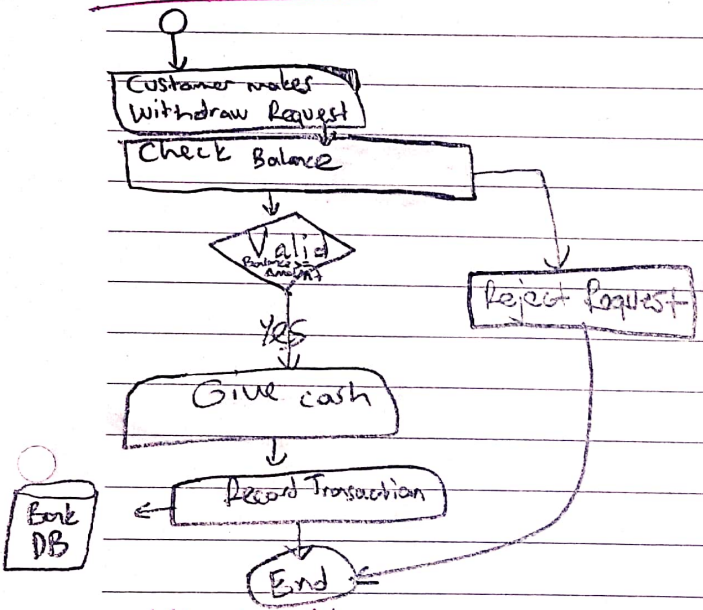
Initial, Repeatable, Defined, Managed, Optimizing

- Initial: Process depends on individuals
- Repeatable: using experience for similar project
- Defined: following documentation
- Managed: setting quantitative quality goals
- Optimizing: continuous process improvement.

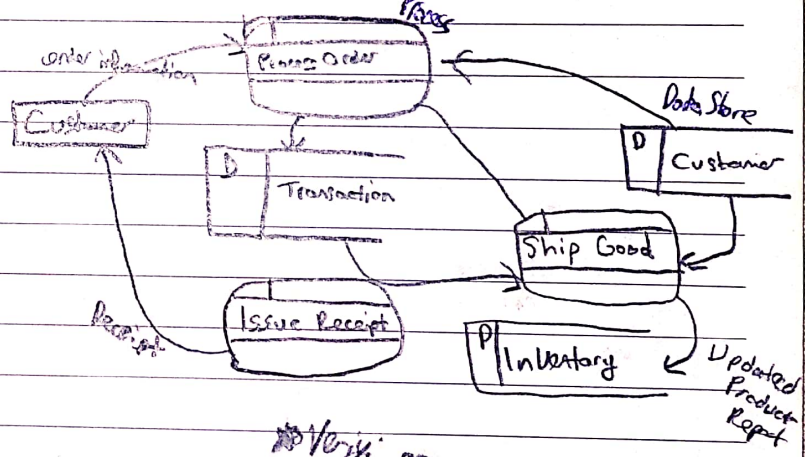
SRS

Introduction / General Description / Requirements for Each Component / Requirements for External Interfaces / Performance Requirements / Design limitations

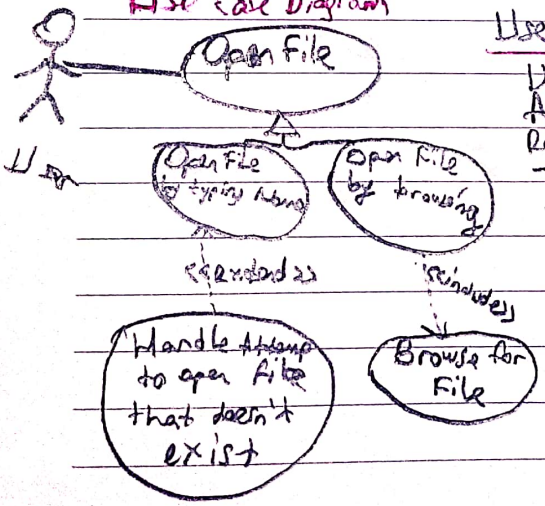
Flowchart Diagram



Data Flow Diagram (DFD)



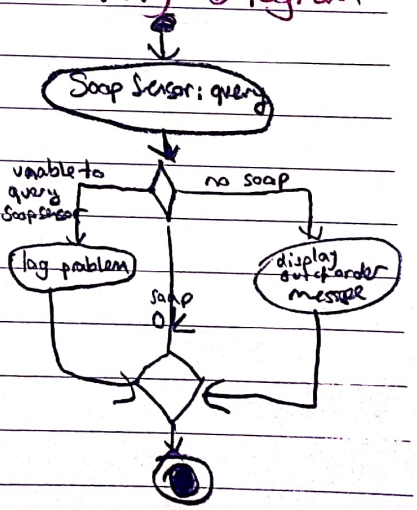
Use Case Diagram



Use Case Description

Use Case Name
Actors (that access that use-case)
Requirements: (Need/want)
Trigger
Basic (Main) Flow: (use case ends!)
Post Conditions: (The appointment has taken)
Alternative Flows: (use case ends!)
↳ You can write errors.

Activity Diagram



Parallel things happen

Fork: not join right for swimlanes of different actors

Sequence Diagram

asynchronous message

Synchronous message

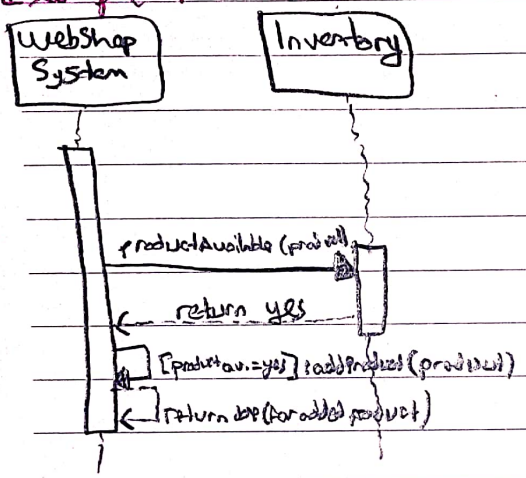
<-------- Return message

alt for return

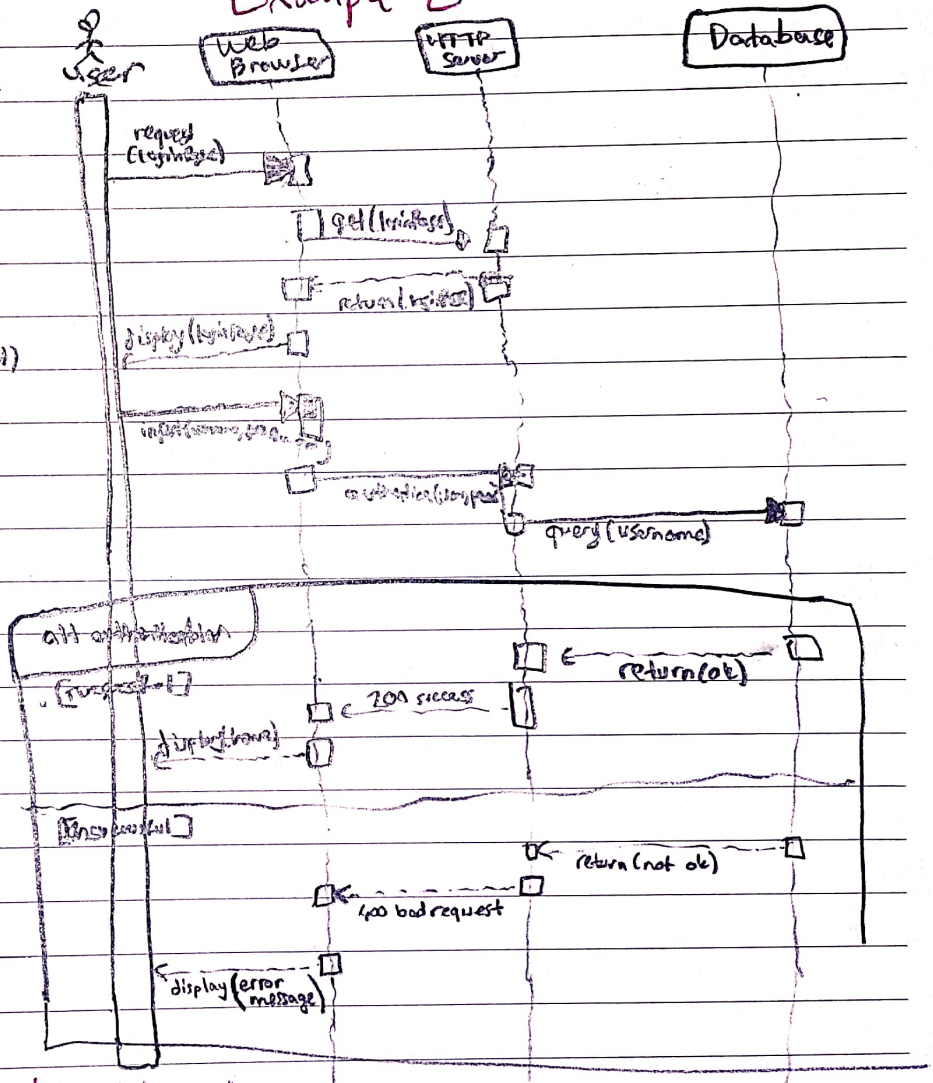
loop

Loop

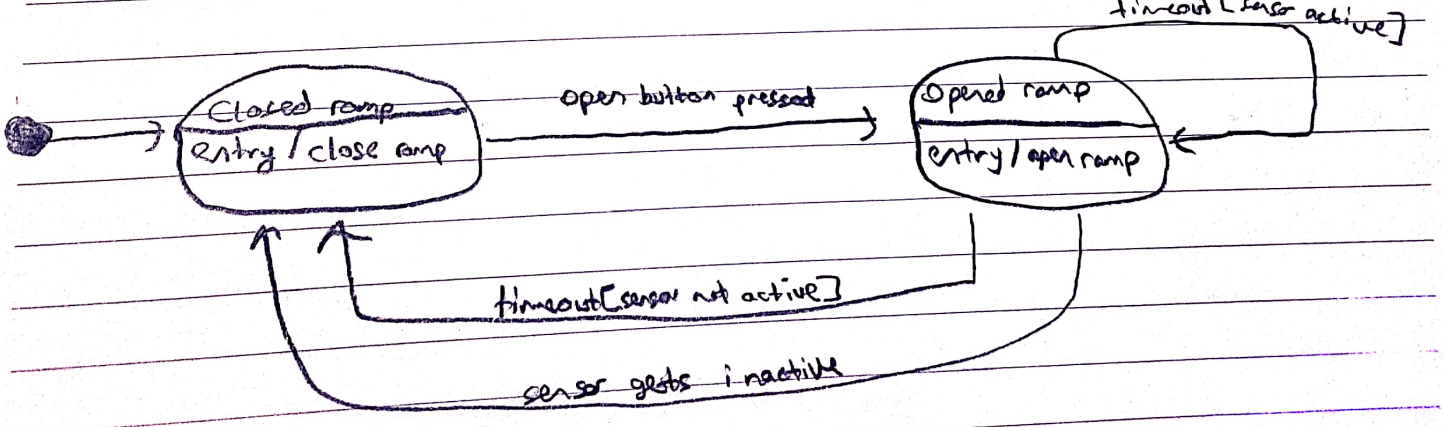
Example 1



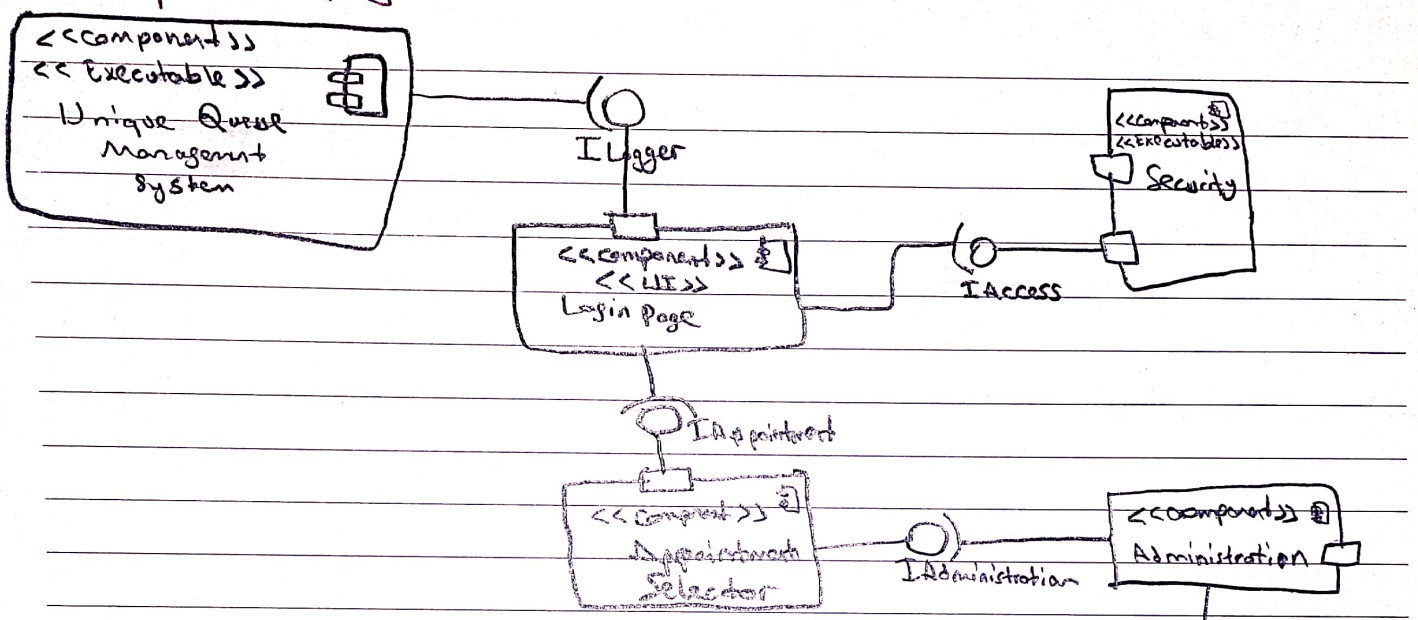
Example 2



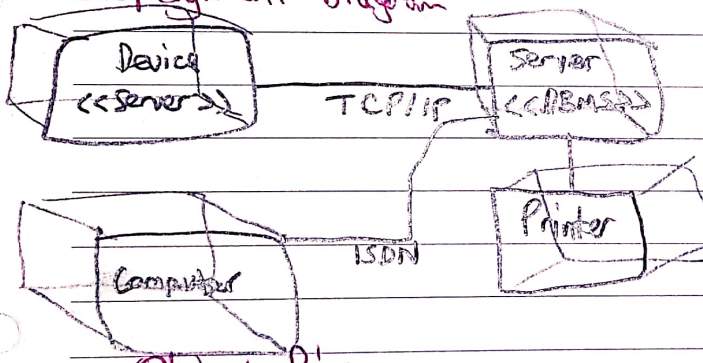
State Machine Diagram



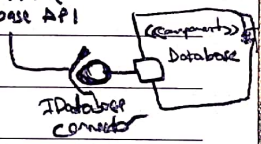
Component Diagram



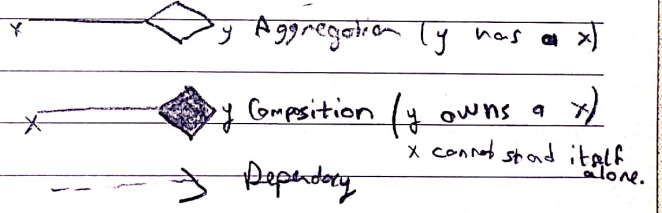
Deployment Diagram



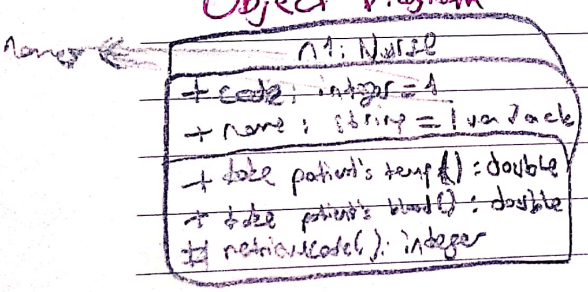
Firebase Realtime Database API

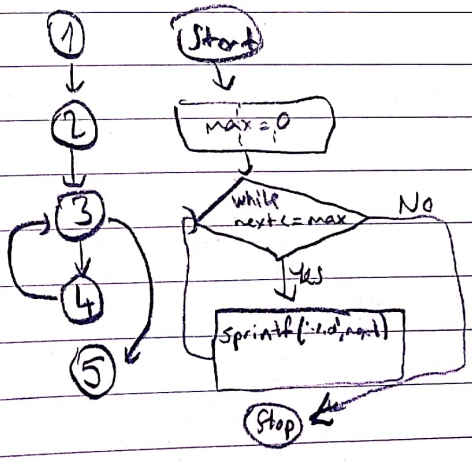


Class Diagram



Object Diagram





Boundry value

Start / End Values

Test case'de only valid values.

Equivalence

Defining Valid and Invalid Classes