Software Engineering

Process

Three phases

1) **Development**
2) **Operation**
3) **Maintenance**

CREATIVITY (Development) + VERIFICATION (v&v, maintenance)

Development >

The "shorter" phase of the software engineering process – *mostly effects the developer.*

- Requirements >> defining the functionalities of the product – what it aims to do, who will use it; everything that is related to the product needs to be specified > **requirement engineering**
- Design >> how the product will be implemented – any product is made by single parts – design aims at specifying what these single parts are (physical, logical…) > **which units, how organized…**
- Implementation >> once we define what the product aims to do and how it aims to do it, we start building it >> **creating the single units and integrating them with each other.**
    - **Prototypes > high / low fidelity (***sketches, mock ups***)**

Operation & maintenance >

The longer phase of the software engineering process – the phase through which the users are using the product – *mostly effects the user.*

Maintenance is a series of smaller development phases, each of which aims at fixing errors and implementing new functionalities – ***the difference is that maintenance is constrainted by what has been done during the development phase.***

- Verification (*verification and validation – testing*) **>>** are the solutions we proposed aligned with the requirements? Are they implemented correctly? > **error and inconsistency checking**
    - **Testing > unit & integration & acceptance tests**
    - **User feedback >** *interviews, ethnographics (observing user behaviour), focus groups*
- Management > *configuration – quality- project management; project planning, tracking, budgeting,*

These operations are not sequential – they can be repeated: the process of software engineering (AGILE method) is based on the repetition of different phases, to ensure that our product matches as much as possible with the requirements.

Goal > produce software (documents, data, code) with defined properties (cost, duration) and certain product properties (functionalities, reliability) >

- *Creating a product that provides specific functionalities in a reliable way, with a given cost and duration.*
    - **Top-down process**
        - Start from the requirements > proceed with design and implementation, then verify if the created product matches with the requirements (verification) and repeat the process while fixing problems.
        - *Bottom-up because we first create a high-level definition of the product, and then we proceed to implement the single parts of it.*
- **Software engineering is based purely on the variable maturity of customers and users >>** *it does not rely on any mathematical principle.*

REQUIREMENTS

Complete and consistent

Remember: each phase of the software engineering process is subject to iteration – this means that every phase will be reviewed, so that any defects, errors and inconsistencies will be corrected.

Without requirements, the product properties are unclear – these let us stay consistent with what the end user needs, making sure we deliver the right product with the right functionalities.

Requirements is the process of identifying the user's problem and the solutions to such problem.

- *A requirement is the description of a product property.*
- It can easily happen that the final product doesn't resonate with the user actual request.

Requirements are described inside the **requirements document**, which contains:

- Informal description
- Business model
- Stakeholders – *who takes part in the product development?*
- Context diagram – *how the different stakeholders and entities are related to the product*
    - Interfaces – *logical, physical…*
- Functional & Non-functional requirements – *what to do and how to do it*

        - Table of rights
- Glossary & Class diagram – *classes, entities*
- System design – how is the system structured
- Deployment diagram – *where the different product parts are delivered*
- Use case diagram – *how will the product be used?*
        - scenarios

1) Informal description

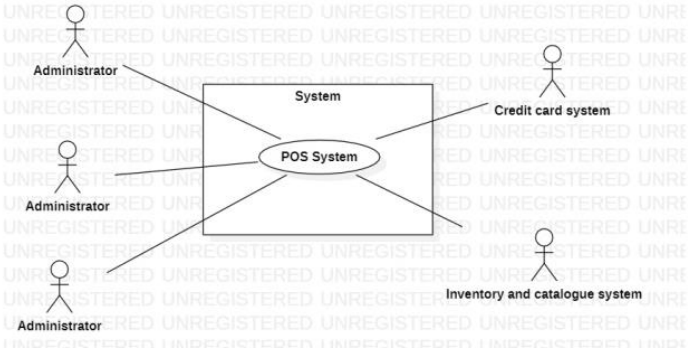
Description given by the user / stakeholder telling us what he wants and needs.
These should be as precise as possible, and in case they are not we should ask for clarification.
Defects > omissions of certain properties (something that is not written or specified), ambiguous information or repeated information.

2) Stakeholders

All the people who have an interest and have a stake in the product we are producing.
(End user, stakeholders, admins, developers, 3rd parties that provide services…)

3) Context Diagram

"In" and "Out" of our product – related to the general context of our product.



- **Interfaces >** *define how each actor interacts with the product*
    - Physical
    - Logical

Actor	Physical interface	Logical interface
Cashier	Screen, keyboard	Graphical User Interface (to be described, ex slide 46)
Product	Laser beam	ReadBarCode (to be specified, ex slide 44)
Credit card system	Internet connection	API description (ex https://developer.visa.com/docs for ViSA APIs
Administrator	Screen, keyboard	Graphical User Interface + command line interface

4) Functional and nonfunctional requirements

Description of what the product should do > Functional

- Description of services
- Main functionalities of the product

Description of how the product should do what it does > Non-Functional

- Constraints on services
- Product characteristics:
  - **Usability >** effort needed to learn the product and use it
    - **Simplicity + Readability**
    - The product and the UI should be easy to understand and require no explanation to use.
  - **Efficiency >** response time, memory usage (less components, the better)
  - **Reliability >** defects over period of time
    - **Stress >** reliability of the system under limit conditions
  - **Maintainability >** effort needed to fix / maintain
  - Portability
  - **Security >** protection of users + their data (different layers of protection)
  - **Safety >** absence of harm to the user
- **They must be measurable**
  - *Doesn't make sense to have a requirement simply say "the system should be easy to use" – we need a concrete way to do it.*
- **Prioritize NF requirements that are important TO USERS, not developers.**

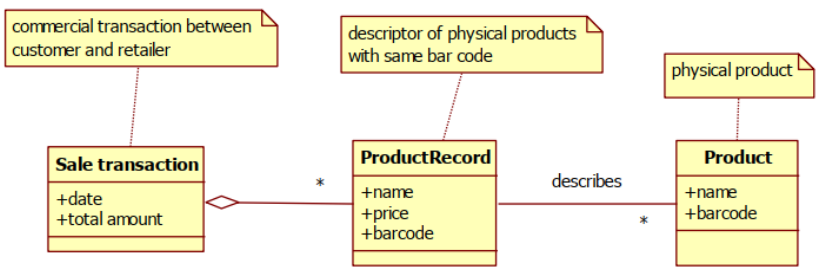
Non functional requirements are important as they describe HOW the product delivers its functionalities – they regard the User Experience factor overall, but they also include constraints of real-life such as state policies, ecc.

5) Glossary & class diagram

The glossary is used to describe the object of our functions and overall the entities that will be used in our product, along with their relationships between each other.
The class diagram shows on a diagram the relationships between these items (classes).
Consider SINGLE items :

- Physical entities > *person, car…*
- Roles > *employee, director…*
- Events > *sale, order, request…*

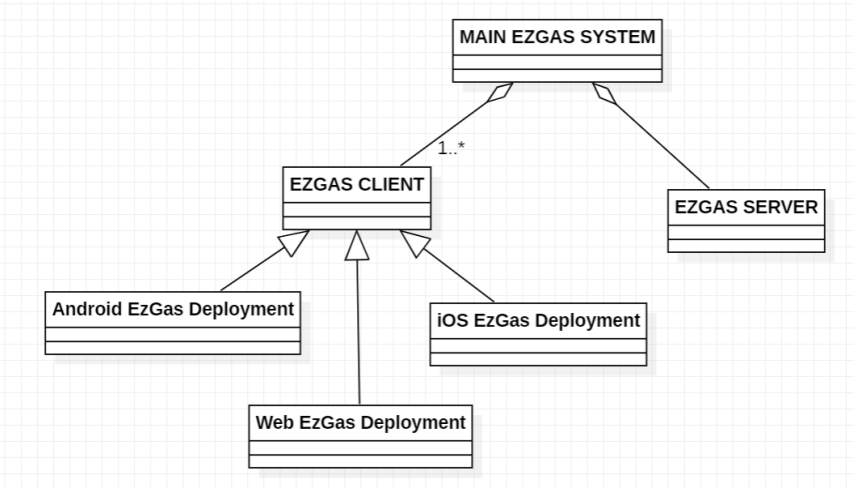
Avoid loops where two links lead to the same object. Keep them if two different items lead to a common one.



6) System design

Describes how the system and the product itself will operate -describing at a higher level the relationships between the components of our product.

Describes the subsystems that compose the system (software and not)



7) Deployment diagram

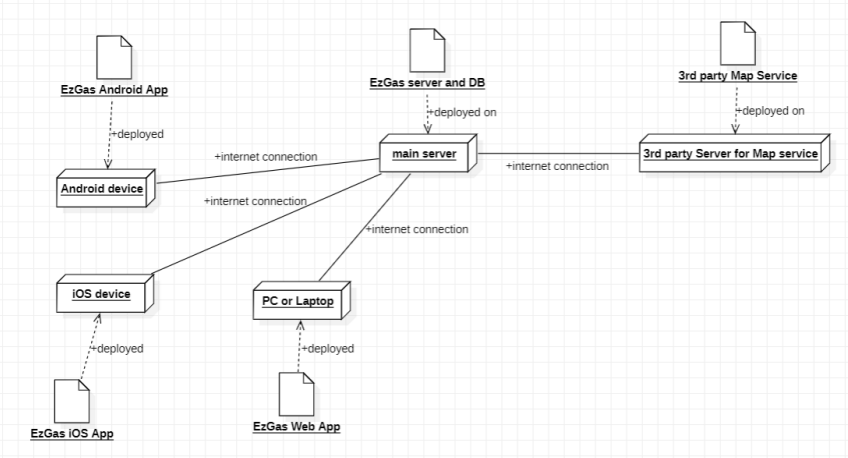
Describes how the components of our product and versions of it will be deployed, in relation to the external components and 3rd party entities that contribute to its functioning.

It's a more-descriptive version of the system diagram, as it doesn't only involve the system itself but also what is interacting with it from the outside.

Nodes > boxes, physical entity or software capable of processing.

Association > physical link

Artifact > file, library, database table..



8) Scenarios & use cases

Use case > functional requirement

Scenario > different versions of executing a use case

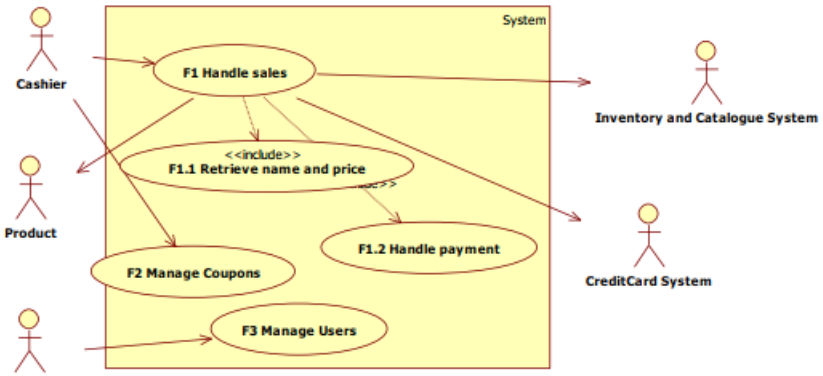
- **Nominal :** *correct execution*
- **Exceptional :** *wrong execution*

Scenarios are used to describe the possible usages of our product, by specifying the actions executed by the users.

They are based on the functional requirements provided in the requirements section.

They describe *all the possible outcomes of a functional requirement (action)*

- Description of the various steps taken in each scenario / use case
- **Precondition >** condition that needs to be valid before starting such scenario.
- **Postcondition >** condition satisfied at the end of the scenario.
- **Both are used to describe every possible use case.**



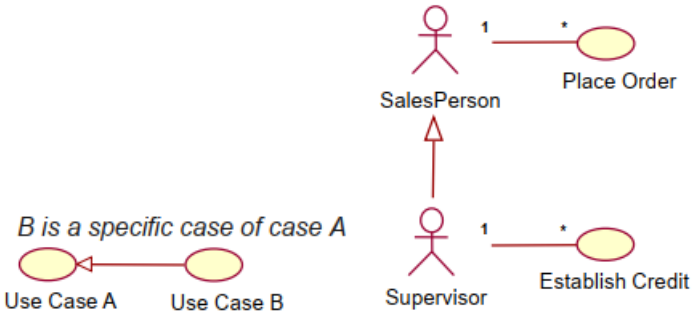
Relationships

They define how two elements or more are associated to each other (if they are).

Generalization - Special case & general case

A is a generalization of B >> **B Is a special case** of the general case A

A supervisor is a specific case of a sales person.



Multiplicity

Specify the maximum amount of links an item can have with another

A car can have up to 4 wheels. A wheel can only belong to a car.

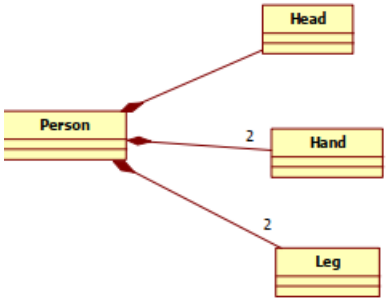
Aggregation

B is part of A.

B can exist without A.

Composition

Same as aggregation – but *B cannot exist without A.* *(hand / head / leg can't exist without a Person)*



Teams and teamwork

When considering a team, we need to also consider its single members.

Single people >

A person, aka a member, has 5 main traits (Myers "Big five")

- Neuroticism
- Openness
- Coscientiousness
- Extraversion
- Agreableness

Each personality trait is fit for a specific job.

Teams >

Perfect size is 4/7 – more than 7 can cause communication problems, creation of sub-groups, need for formalization, possible less participation

- **Team norms >** productivity + protest
    - Productivity defines what is too much and what is too little work.
    - Protest defines how and when to express conflict and contrary ideas.
- **Specialization >>** defines wether all team members can do all tasks or not
    - **Yes >** *specific team members do specific tasks*
    - **No >** *all team members perform all kinds of tasks*
- **Leadership >** useful to define who does what or who represents the team – should not be defined at priori but it's better to have it emerge through democratic decisions
    - *Organizations where members can express ideas and critiques succeed*
- ***Summary >*** *working in teams is a must – conflict is necessary, especially to have different ideas between different people and recognize defects in our product more easily (if everybody has the same ideas, it's harder to recognize wether we're failing or not) – a good team performs more efficiently than single people.*

PROJECT MANAGEMENT

First step of software engineering process – *even before development*

- *Continues throughout the entire process*
- **Initial estimation + tracking during process**
    - How much > *estimation + tracking*
    - When > *estimation + scheduling + tracking*

1) Initial estimation

- initial proposal > start with planning and estimation of how long and how much (often not realistic)
    - **no initial estimation can ever be accurate – anything can happen during the development process, just like in life.**
        - An approximation of size estimation can be done depending on the units we will be using.
        - Estimation becomes more accurate as the process goes on.
        - It is important to have a limit to the project (certain goals / milestones, *Parkinson's Law*)
- **ESTIMATION BY DECOMPOSITION >>** most efficient and convenient
    - **BY ACTIVITY >** the work is divided in sub activities, then an amount of person hours is estimated for each sub activity.
    - **BY PRODUCT >** identify products, estimate effort per product.

- **vendors usually overpromise so that the buyer is then stuck to his product during maintenance.**
- **inception >** development of the idea and defining the requirements.
- **dimension >** is the product a service or does the user have full ownership of the product?
- **Constraints >** safety, domain…they effect deeply the SW is developed.
- Also used to define contractual aspects (contract, price (fixed / usage fee)

2) Development + tracking

- Different decisions
  - Project structure
  - Resource allocation
  - Time planning

3) Deployment + post-mortem

- Post-deployment meeting, collecting project data and understanding total effort (estimated and actual), achievements, problems, causes and errors.
- The goal is that to make the next development processes better.

Planning and estimations are based on the concepts of resources and activities to complete – milestones.

- ***Milestone*** >> key event in the project
- ***Deliverable*** >> prototypes / deliverable to the user to test functionalities and receive feedback.

The initial planning, especially regarding time, is often based around the benchmark value of other companies and products. (*a company will base the amount of time it takes to finish a product based on other companies' time)*

PLANNING TECHNIQUES – Agile / Waterfall; Cost, size and quality – GANTT chart

2 ways >

1) Waterfall

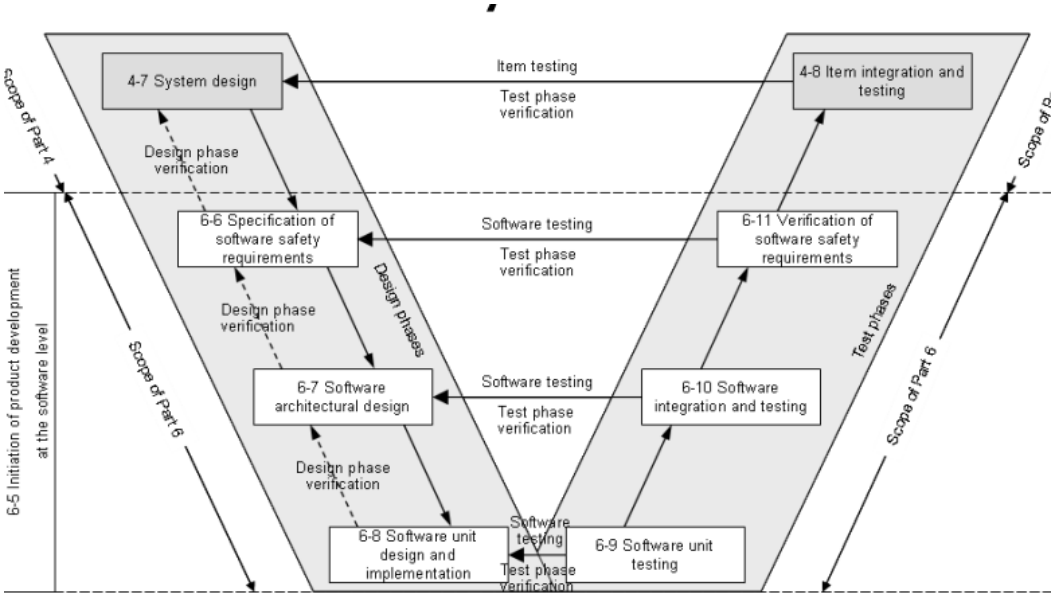
Sequential activities – focus on every part of the process

Fixed price models

Static approach, based on sequential phases – once one is done, proceed to next

- Requirements > planning > define GANTT > development.
- More suited for bigger projects, especially with hardware / safety dependencies.
- even if approach is sequential, prototypes can be built directly during requirements phase > requires team skills to develop and maintain throughout the process.
- **Estimations are based off effort (cost*ph)**

- **PROS >** req. document usually very compliant with user needs, better for large products and large teams
- **CONS >** changes have heavy impact – tension to avoid changes (*worst case are design and requirement changes*)
    - **Contracts usually defined at start of development – changes in the product development may require changes in contract (price,duration…)**
    - **V model >** similar to waterfall, focus on V&V activities (Validation & verification)
        - Tests are written right after design phase



2) Incremental – RUP

Parallel activities – partial emphasis on documents

Very similar to waterfall approach, but the integration phase is incremental and repeated – every loop of integration produces a part of the system that is reviewable by the end user and can be used to fix defects in the next iteration. (*Early user feedback*)

- also good for bigger projects with bigger teams, mostly used for new developments.

3) Agile

Parallel activities – no emphasis on documents

Time and material model – dynamic price

Dynamic approach, multiple repetitions of same phases – provides a more accurate product and less defects:

- **Requirements and design can be very short at beginning > they are adjusted every iteration and every time a defect is found. (***Either by the user or by the developers***)**
- **Agile methods rely on being in touch with the user and adjusting the product to his/her needs.**
- When an error, inconsistency or ambiguity is detected, go back and repeat other steps.
- In case of errors, solving one error could cause other errors **(REGRESSION)**> once fixed, check if no other errors are present > otherwise, repeat.
- More suited for smaller projects, different repetitions may take longer than expected.
- **Estimations are based off effort(cost*ph)*iterations**

AGILE PRINCIPLES:

- COMMUNICATION > problems can arise from individuals not talking about something important. (*Project management*)
- SIMPLICITY > the simplest solution is often the best one. (*Simple design, runs all tests*)
- FEEDBACK (*customer satisfaction + On-site customer + small releases*)
- COURAGE > doing the right thing, without fearing failure. (*Software quality, refactoring*)
- CODING & DEVELOPMENT STANTARDS

AGILE ADDITIONAL METHODS:

- **Scrum:**
  - Team leader + dev. Team + user.
  - Team has a meeting and defines how to achieve a certain goal (sprint) and when to have it ready to show to the user – then proceed to show the demo to the user.
- **Extreme Programming (XP):**
  - Write tests first, write code accordingly.
  - Less iterations with fast delivery
  - No upfront design, development of infrastructure
- Pair programming:
  - Two people code together, one develops code, one thinks how to make it better (less efficiency? Using double the effort over the time unit)

4) City model

Always-on model (maximum reliability), horizontal functions

Example: Twitter

The product is always on, what teams work on is different functions that do not interfere with the main functionality of the product itself:

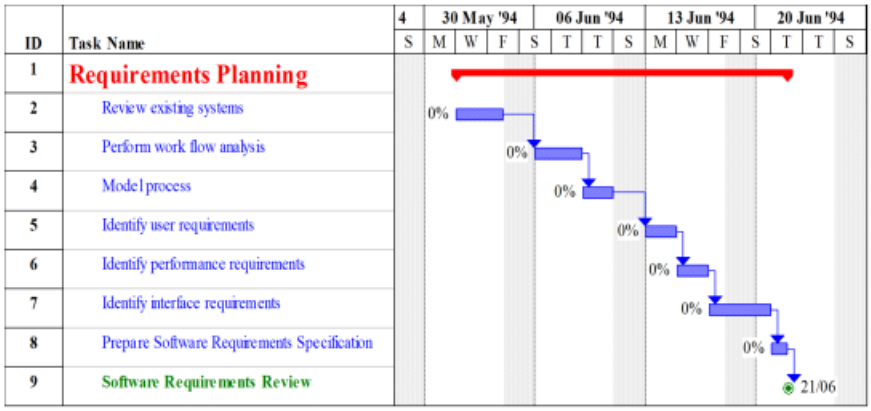
- Functions are said to be horizontal.
- Changes are frequent.

WBS – Work breakdown structure

Planning by decomposition – by activity

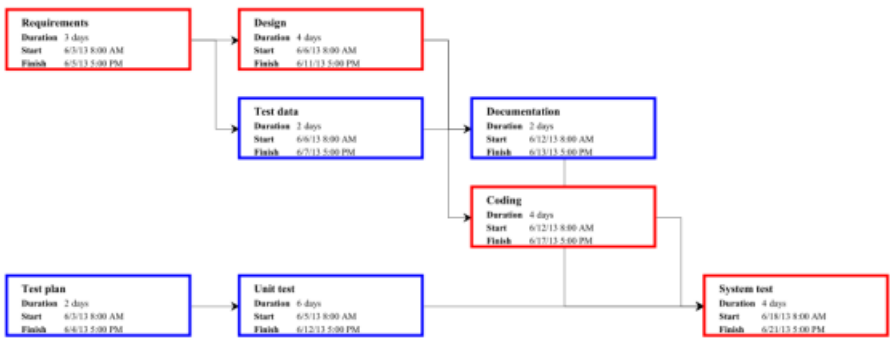
The work is divided into sub activities, then a total of person hours is estimated for each activity – part of requirements planning.

GANTT CHART >> defines how the work is divided + temporal planning + *tracking* of what is actually done



GANTT chart used along with **STAFFING >** estimating how many people are going to work on certain activities.

PERT CHART – directed acyclic graph, shows different steps of development process in order (arrows indicate what needs to be executed before certain tasks).



Cost, Time and Quality are linked together.

1) **COST**

- Relative – *month 1, month 2…*
- Absolute – *precise date*
    - *Total cost depends on hardware, software, and staff (person hours spent on tasks)*
    - *The longer the time spent on the process, the less important is the cost of acquisition for a resource.*

2) **EFFORT**

DURATION * RESOURCES = time taken by a resource to complete a task, measured in Person Hours

- 1 person day = 7 ph

3) **SIZE**

Size of our project >

- *Source code > LOC (lines of code)* (specify what to count and what not)
- *Documents > pages*
- *Tests*

- **Productivity = TOTAL SIZE / TOTAL EFFORT *(person hours)***

TRACKING TECHNIQUES – *timesheet, list of closed activities (milestones)*

The goal of tracking is that to collect project data, effort spent by resources (person hours), size of the project > *the goal is that to make the initial estimation as accurate as possible throughout the project.*

Timesheet > used to keep track of how many person hours have been spent on a certain activity.

Activities closed > activities that have been completed.

V & V = Validation & Verification

Preventing, finding and minimizing defects

Static and dynamic analysis > inspection, code analysis and tests

Validation = is it the right solution? Is it really what the user asked for? It is the most effective solution to the user's problem? →*external correctness, user based*.

- Depends on the stakeholders / user needs.
- A product can be fully functional but still not be valid if it's not what the user wanted.
- This means that even if a product functionality is simple, as long as it solves a user need, it is valid and does not need to be more complicated.

Verification = is our solution implemented correctly? → *internal correctness.*

- Done by the engineers.

Coupling > The degree to which two modules are connected (ex share of a variable, message passing, function call)

FAILURE & FAULT

FAILURE = unexpected execution of the product

FAULT = malfunction by the system, causes failures (unexpected behaviors by product)

- Especially after release, 10% of modules cause 80% of all faults.
- These could easily lead to continue changes in our product, as a fix to one fault could cause some others.
- The earlier they are detected; the less resources are needed to fix them.
- **Fixing a fault later in the process could cause more faults to rise up > REGRESSION.**
    - **Regression can be avoided by fixing the faults while using the same test suite > keeps product coherent**
    - **Regression is more likely to happen in case of multiple dependencies – when the dependencies are based off a unit that has a defect.**

Regression testing

After fixing a defect, tests previously defined are repeated to ensure that the new change hasn't introduced any new defects.

VALIDATION AND VERIFICATION TECHNIQUES

The goal is that to find as many defects as possible.

Tests need to be performed on a variety of data, not always on the same ones.

Test cases should be documented (*formal testing*) and *automated* – testing, just like any other activity, requires resources – but it can require more than expected.

1) Static

Inspections

- Reading documents / code – can find many defects at once, tests find one at a time
- Early inspections reduce avoidable rework
- Usually done by groups of people – the goal is that to make the product as valid as possible, finding defects is the tool with which we reach that goal.
- **Checklists** can be used to set standards.

Source code analysis

- Compilation analysis (syntax, types and semantic analysis – done by compilers nowadays)
- **MISRA-C rules**: *common guidelines for software products – originally created to make the C language safer*
- **Bad smells >** bad programming patterns and usages
  - *duplicate code, long methods, switch statements, wasteful code (useless and unused variables…)…*
- **Data flow analysis >** analyzing the value of certain variables during the execution of our program to look for anomalies (*debugger for example*)

2) Dynamic

Testing – requires a product prototype to perform tests on.
The goal is that to detect the differences between actual and **required** behavior.

- Testing is based on finding *defects.*
- Debugging is based on finding faults (errors in implementation)
- Testing and debugging can lead to spotting design failures.
- Tests can be written by either the developer or the testing team.

System test : *testing entire system*
Unit test : *testing singular functions / units / components*

Test suite – set of test cases

- **Test case –** test performed under certain conditions.

- **ORACLE :** gives us the expected behavior of a program / function for a given test case.
  - **Easier said than done >** can use the requirement document to know what to expect
- **CORRECTNESS >** correct output for all possible inputs → *exhaustive testing*
  - **Not possible >** there are infinite test cases, testing takes resources too.
  - **No system is perfect – our goal is that to find defects, not demonstrate that they are not present.**
- **COVERAGE >** how much of the product our tests are covering
  - **Entities considered by at least one test case, over all the possible entities**

- **Testing and Object-Oriented classes > Observability (**how to read the internal values?**), Controllability (**how to manipulate them?**)**

TEST CLASSIFICATION

- **How to analyze each item?**
  - Unit > individual modules
  - Integration > different units together, 1 test per dependency
  - System > all modules together, 1 test per func. Req.
- **What approach to use?**
  - Black box
    - *Testing input-output without knowing internal code.*
    - Unit, Integration & system
  - White Box
    - *Testing input-output starting from source code – takes less time, more precise.*
    - Unit tests
  - Reliability & Risk based (System)

UNIT TESTING – black box & white box

Black box testing

Equivalence classes

Criterion: defines an attribute.

Predicate: specifies the value of a criterion.

Boundaries: specifies boundary values of a criterion.

Coverage requirement > *atleast one test case per partition / boundary*

Specifying all possible criterions and all possible values for them (predicates) will give us the best possible coverage.

Tests are written by taking as input a combination of all possible predicates and boundaries specified.

How to write test cases with equivalence classes + boundaries

1) Define criterions and predicates properly – define valid ranges of values and invalid ones
2) Write boundaries -> boundaries are the ranges of values, valid and invalid, which can be passed to the function we are testing
3) Write tests picking values from each boundary range ->

sex	height	weight	BMI	Valid / invalid	Test case
[minint, 0]	-	-	-	I	T(-10, 20,20) → 0
[3 maxint]	-	-	-	I	T(11, 20,20) → 0
-	[minint, 0]	-	-	I	T(1, -10, 20) → 0
-	[300, maxint]	-	-	I	T(1, 400, 20) → 0
-	-	[minint 0]	-	I	T(1, 100, -20) → 0
-	-	[500, maxint]	-	I	T(1, 100, 2000) → 0
1	[1, 299]	[1,499]	Normal	V	T(1, 180, 75) → 1
			Below	V	T(1, 180, 50) → 2
			above	V	T(1, 180, 300) → 3
2	[1, 299]	[1,499]	Normal	V	T(2, 180, 75) → 1
			Below	V	T(2, 180, 50) → 2
			above	V	T(2, 180, 300) → 3

Here, sex can have valid values of only 0 and 1.

This means that the boundaries for "sex" are:

- 0
- 1
- [2,inf]
- [-inf, -1]

Which means we will write test cases where we consider 0,1, then a random value between [2,inf] and one between [-inf,-1]

Criteria

Criterion id	description
C1	Sign of carbs
C2	Sign of protein
C3	Sign of fat
C4	Formula 1
C5	Formula 2

Predicates

Predicate id	Predicate
P1, interval	C1 >= 0
P2, interval	C1 < 0
P3, interval	C2 >= 0
P4, interval	C2 < 0
P5, interval	C3 >= 0
P6, interval	C3 < 0
P7, single value	C4 true, false
P8, single value	C5 true, false

Boundaries

Criterion	Boundary
C1	-1,0, 250

Equivalence classes and tests

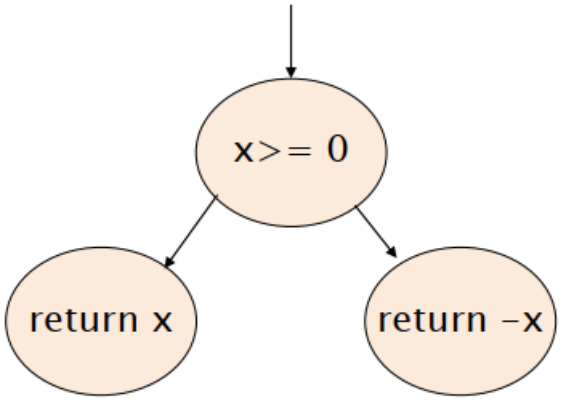
Test cases with B are test cases considering boundary elements.

C1	C2	C3	*Valid invalid*	Test case
>= 0	>= 0	>= 0	valid	T1 = true / false
= 0	>= 0	>= 0	valid	T1.0B = true / false
>= 0	= 0	>= 0	valid	T1.1B = true / false
>= 0	>= 0	= 0	valid	T1.2B = true / false
= 250	> 0	> 0	invalid	T1.3B = false
= 250	= 0	= 0	valid	T1.4B = true
> 250	>= 0	>= 0	invalid	T1.5B = false
> 0	= 250	> 0	invalid	T1.6B = false
= 0	= 250	= 0	valid	T1.7B = true
>= 0	> 250	>= 0	invalid	T1.8B = false
> 0	> 0	= 111	invalid	T1.9B = false
= 0	= 0	= 111	valid	T1.10B = true
>= 0	>= 0	> 111	invalid	T1.11B = false
< 0	>= 0	>= 0	Invalid	T2 = false
>= 0	**< 0**	>= 0	invalid	T3 = false
>= 0	>= 0	**< 0**	Invalid	T4 = false

White box testing

Coverage requirement > *all statements*

Based on coverage – *we basically transform our program in a control flow chart, where each node is a statement. (**Function becomes a graph**)* → our goal is that to cover 100% of the nodes (*statements*) with our tests.



We also consider edge coverage – this should also be 100%.

Node coverage = Statement coverage

LOOP COVERAGE > Loops need to consider 3 cases:

- **The loop is not entered.**
- **The loop is entered once.**
- **The loop is entered more than once.**

PATH COVERAGE > you need to consider all the possible paths that your function will take.

- This applies mostly to loops > if I have a loop that iterates n times, I will have n different possible paths – where n is atmost maxint, so maxint paths.
- This makes it basically impossible in case of said loops to compute all possible paths

- Apart from loops, path coverage depends on all the other possible statements in your function (such as if / else, Boolean conditions…)

MULTIPLE CONDITION COVERAGE > you need to consider, given a Boolean condition that is based off multiple boolean conditions, all possible combinations of these:

- This translates into producing as many tests as necessary to cover all TRUE / FALSE cases
    - Es. I < 0 || E < 0
    - I need to consider all the possible cases: FF, FT, TF, TT
    - If one of the said ones are not possible, specify it.

So…

1) Define control flow chart of function.
2) Define test cases given our function.
3) Write test table and select best combination of test cases to obtain 100% coverage.

Compute coverage for the given test cases

Test cases	Nodes covered	Node coverage per test case	Node coverage per test suite	Edge covered	Edge coverage per test case Per test suite	Loop in line 5
all_equals = {0,0,0,0,0};	N1,N3,N4,N5,N7,N8,N9,N10	8/10 nodes covered	80%	All but N5-N6, N6-N8, N1-N2	8/11	Loop many
all_positive = {1,2,3,4,5}; note: this test case doesn't provide anything additional from last test case - useless	N1,N3,N4,N5,N7,N8,N9,N10	8/10 nodes covered	80%	All but N5-N6, N6-N8, N1-N2	8/11	Loop many
all_negative = {-1,-2,-3,-4,-5}	N1,N3,N4,N5,N6,N8,N9,N10	8/10	**90% (incremental from 80% of preceding cases, we added one node (N2))**	All but N5-N7, N7-N8, N1-N2	8/11	Loop many
[] out_of_size = {1,2,3,4,5,6};	N1,N2	2/10	100%	N1-N2	11/11	--not relevant, does not try to execute the loop
mixed = {-10,10,3,5,-6}; **note:** all test cases after 100% is reached, are not needed.	All but N2	9/10	**90% (Already 100%)**	All but N1-N2	**10/11 (Already 11/11)**	Loop many

Integration testing – incremental testing

Based on each unit's relationship with each other – we need to test dependencies and units.

Dependencies are tested by using stubs (fake unit, simpler**)– simulating single functions values.**
Functions and units should first be tested separately, and then tested by their integration – this will make it easier to detect errors and solve eventual defects. (*avoid big-bang integration*).

- **Incremental integration** is an example of this technique > *finds defects really fast, but lots of tests to write (effort needed)*
    - **Top-down >** *very efficient for early-versions of a product – cannot test single units directly, allows for detecting design problems.*
    - **Bottom-up >** *starts from single units, doesn't deliver high-level functionalities at first – easier to detect errors and to know where they are coming from.*
    - **Both are good, just avoid BIG-BANG testing.**
    - **They are usually mixed.**

System testing – requirements and scenarios coverage

Coverage requirement > *atleast 1 test case per requirement.*

Test of all units composing the product – consider all aspects:

- Functional / non-functional properties
- Platform: development, production
- Player: developer? Tester? End User?

The platform defines where the product is to be tested, produced, and used. *(environment)*

- Development
- Production > *where it's produced.*
    - *Tests usually performed here.*
- Target > *where it's going to be used.*
    - *tests can be performed here too – just do not use the same target as the users.*

System testing can be done by both the user (beta testing**) and developer.**

- **Acceptance testing >** system test performed by users, written by acquirer.

- **Beta testing >** system test before the product is released.

System tests are done by:

- Covering functional properties / requirements
- Covering use cases / scenarios.
- **How?**
- **USAGE PROFILES >** defines the most common usages of the product (*usage patterns*), testing the functions that are used "more."

Risk-based testing

1) Identify risks.
2) Rank them based on **probability / effect** relation.

Mutation testing

Used to verify test validity:

- Keep same tests, insert errors in code.
- If the tests catch the errors, tests are valid.

What is the cost of testing?

Economics for test automation

Cost:

- Effort to >
    - *Invent test case.*
    - *Document test case.*
    - *Run test case.*

- Automated test cases only require the invent and document test once – on the other hand, non-automated tests will require way more resources.
    - **Automating a test is worth it if it is executed many times.**
    - **<u>All types of tests will need to be changed if a defect is found >> this is why testing requires many resources.</u>**

SOFTWARE <u>DESIGN</u>

Architecture & Design

<u>Most defects come from this phase.</u>

Architecture: high level description of how the product is going to be built, interaction within components

Design: low level description of each component

- Given a set of requirements, many design choices are possible.
- The choice of which design to adopt is based off the team's skills, creativity and mostly experience.

The design process is different based off the kind of product we are developing → developing a simple software only requires a software design phase:

- Developing an entire system requires system design + software design (*also defining how the system interacts with each software…*)
- **Architecture and design can be described *<u>informally (</u>*** *<u>simple tables, better for showing the user how the product behaves</u>* ***<u>)</u>*** **or *<u>formally (</u>*** *<u>UML diagrams, better for developers</u>* ***<u>)</u>***
- **Pre-existing Patterns** can be used as architecture and design patterns → *re-usable solutions to recurring problems*

Process design

First, we define the architecture (high level description with relations between each component, if existent), then we define the design of each component.

- **Coupling**: refers to the degree of dependency between two components.

- **Cohesion:** refers to the degree of *consistency* between the functions of a component.

Process design and NF requirements – what to do?

- Performance → *localize critical operations and minimize communications – so that whenever an error occurs, it's contained.*
- Security & Safety→ *use layered architecture to protect critical assets and operations.*
- Availability → *use redundant components to make sure the data is always available.*
- Maintainability → *use replaceable components.*

Obviously, not all properties can be defined – decide necessary tradeoffs (*for example deciding whether to have less layers of security but better performance*)

1) Architectural patterns

A real system is often influenced by more than one type of pre-existing pattern:

- ***Repository*** > one central data holder or multiple ones that exchange between each other.
  - *Centralized management; subsystems MUST have common data model*
- ***Client-Server*** > stand-alone servers provide data to specific services.
  - *Straightforward distribution; new integrations are easy*
  - *No shared data model between consumers – need to adapt to server's data model*
- ***Abstract-machine*** > system organized in different sub-layers, each with its set of services.
- ***Pipes & Filters >*** data flows in, filter is applied, data flows out (*ex. API*)
- ***MVC – Model View controller >*** model contains data, view shows data, controller handles interaction between the two to keep shown data consistent with model.
  - ***Separation of responsibilities, but more complex (less performance)***
- ***Microkernel >*** different APIs, one main hardware component – need to maintain portability and consistency
- ***Microservices >*** different independent services based on their own http APIs; each one has its own data model
  - *Less efficient*

2) Design patterns

Design is the description of how a single component is structured…

A design pattern identifies the key aspects of a common design structure:

The goal is that to create a reusable object-oriented design

A component can be for example a class >

- **Choices can be made starting from glossary and context diagram.**
- **Class patterns >** common class usages, such as defining public / private, getters / setters, defining types.
- **Relationships** are the ones discussed previously -> association, generalization, specialization…

Possible design patterns are as follow:

- ***Adapter >*** required class doesn't implement required interface – class has the goal of being the intermediate between required function class and required interface.
- ***Composite >*** used to represent a series of objects that belong to a hierarchy.
- ***Façade >*** used to represent a single object that contains multiple objects functionalities – without accessing the objects themselves.
- ***Observer >*** monitors a particular object to apply changes to other objects.
- ***Strategy >*** used to perform a specific action between two or more objects – applies an algorithm.

SOFTWARE PROCESSES

Definition, who executes them, what are their results are.

Processes are divided into

- **Primary**
  - Acquisition / supply(acquiring resource suppliers and customers)

        - Development
        - Operation (deployment)
        - Maintenance
- **Supporting**
        - Documentation, configuration management, quality assurance (V&V)…
        - *Not directly related to the product*
- **Organisational**
        - Project management
        - Monitoring
        - Training
        - *Activities that regard the developers of the product and the product utilization*

Process models = models used to decide how the activity is organized in terms of temporal and materialistic constraints

- *What, who, when, constraints*
- A process model is *only a part of the software process (among with activities and roles)*
        - Activities are the ones we described before (requirement doc, design, implementation…)
- **Process models can be created from scratch, or be based off standards >**
        - **Waterfall >** one iteration, longer duration for each activity
        - **Agile >** multiple iterations, shorter activity duration
        - *An example of a process model is one that is based off building a starting version of the product – then showing it to the user and fixing it accordingly to its requirements > this could work for a small project, but for bigger projects it would result in an excessive waste of resources.*
        - This means that **the choice of what process model to adopt is base off different factors**, such as:
            - Product constraints (national, safety, mission-critical (certain requirements MUST be implemented at all costs))
            - Size of end product (end product must have a certain size)

Processes – reuse

Using multiple times open/closed source components.

Why?

- easily available, higher quality with low cost
- …but not owned by developers, also no control over their evolution (*an internal change in the component can't be controlled and may change our product*)
- **Reusing components that already exist needs to be taken into consideration during both the requirements and design phase.**
    - Also consider constraints that follow these components' usage.

Maintenance

Maintenance is based off the concept of change:

- Defect (fault / failure)
- Modification to existing function
- New function

Every change requires the product to maintain its original architecture over the time – and maintain its market sustainability >

- Reason why only a part of changes is actually implemented, even if corrective.
- Changes can be required or created by either users (who interact with the product by signaling faults / failures), by developers or automatic failure detection systems.
- **Three types of changes during maintenance >**
    - Corrective
    - Enhancement
    - Evolutive

DevOps

Dev Ops is about the relation between development team and operation-phase team.

- Development team knows the product layout and information, its requirements etc.
- **The goal is that to remove communication and knowledge barriers.**

Synch and stabilize

Context where there are many defects that need to be fixed, but cannot be fixed at the moment: the procedure is split in small groups that each focus on a certain amount of tasks – *every team synchronizes with each other frequently*

- ***This creates an iterative process where the product is fixed incrementally and in a stable way*** → *all teams converge in a single version of the application every iteration.*

How?

3 phases

1) ***Planning:*** product managers define goals, priorities, and team formations → manager + developers + testers
2) ***Development***: development is split in groups, 1 subproject is analyzed by more teams – each team produces code, debugs it and tests it
3) ***Stabilization***: when a subproject is finished, the code / product developed by each team is joined in a single component, that is stable.

Sync and stabilize solves the problem of DevOps by having multiple teams work on the same subproject (product component) and interacting with each other consistently during the stabilization part.

Product qualities – how to pick process model.

- Criticality + Domain – if product has certain constraints, sequential is better.
- Size – large project size is better faced with a waterfall / synch and stabilize process, to avoid inconsistent product versions and too many resource-wasting repetitions.
- Technical debt – the effort we don't spend today, we will have to spend later + interest. Developing a starting version of the product fast might require more time to fix later on – sequential is better (waterfall / sequential)
- Relationship with developers – if developers are external, might need a process model that bases itself more on documenting the product requirements (sequential)

A process model can also be picked based on what we want to guarantee to the end user >

- **Reliability:** a reliable product is easier to develop in sequential process. (waterfall)
- **Market driven:** a product that is able to adapt to the market is easier implemented by Agile methods and overall methods based off multiple iterations.