Objective of the subject

This course covers the basis of the most used Software Engineering process.

We will focus on 2 things:
- Methodologies
- Tools that the methodology provides to create the software and complete the different tasks

Includes the needs for methodologies of software development:
- Analysis of software requirements → What it needs to do.
- Design → Create the architecture of the software
- Implementation (20% of the work) → Make the code.
- Validation

These are the 4 tasks that represent the overall creation of the software.

There are different methodologies to create software. With an overview of the basic elements of Object-Oriented Design:
- UML: Unified Modeling Language
- OO abstraction: Inheritance, polymorphism and interfaces
- Testing tools and methodologies
- Model views

What are software engineering, methodology and OO design?

Software engineering is a discipline to develop software with a good organization and following a specific methodology. It's a technical task that requires a good organization of our documentation, diagramas, source code…
- Since we are working in a group, we need to follow a methodology that everyone will follow, to create a software.

A **methodology** is a set of rules, methods and techniques to achieve an objective.
- To define the architecture, we could use different diagrams. So, perhaps some methodology describes different tools to create these diagrams.

Object Oriented Design is a technique to model a software system using the principles of OO programming.
- An object is an instance of a class

Therefore, you must modify your workflow to make it perform differently.

The general objective of the course is to provide an overall and ordered view of the software development process which does not only cover programming.
We will introduce the main activities that constitute the development process.

Our aim is that the student acquires a general vision of what processes are involved in software development, specifically…

- How to manage a given software project → Repository, who needs to complete each task…
- How to model the software → Definition of the architecture
- How to implement (code) and validate the results of the project (check that the result of the implementation is following the model that we have specified).

Working methodology

Scrum: relevant topics

- Product backlog: list of requirements of the software to develop
- Sprint backlog: a group of requirements (divided in micro-tasks) used during the sprint. Requirements of the team to complete some part of the product backlog.
- Scrum-Master: person that manages the tasks, does daily monitoring, etc. (ideally one person of the group each sprint)
- Product-Owner: it is usually the client (Supervisor)
- Daily-Meeting: it is explained briefly the progress
- Sprint-Review: each session is discussed more in detail the progress (practicum session)

Software tools: register and read a guide

- Online repository: http://www.github.com/
- Task Manager: https://trello.com/
- Simple UML modeler: http://plantuml.com/
- Documentation: MD (markdown) + PDF (print from any editor) + PNG/UML (for diagrams)

Block 1. Basic Topics

It is relevant to comment on different paradigms that we can use in the different methodologies to complete the different phases.

First, we will introduce how the software has evolved over time.
Why is it necessary to use Software Engineering to achieve good results?

In the field of software engineering, these are basic concepts:

- **Problem**: something that we need to solve.
- **Specification**: problem description and analysis about what you need to implement.
- **Algorithm**: problem solution (written for humans).
- **Program**: problem solution (written for computers).

So, the objective is to implement a program from a problem in a correct way. To do this, we must follow the pipeline shown before.

The main objective of SE is to solve a problem using computing programming in the best possible and efficient way!

Therefore, the key factor is the efficiency of:

- The implementation (program)
- Task development (software engineering) → Spend to much time and money

Software engineering has different areas covering different objectives:

Operative Objectives

- ***Correctness***: to what extent the program satisfies the initial specifications.
- ***Accuracy***: it is the property that defines with what degree the software complies with the established requirements.
- ***Efficiency***: it is a factor that refers in every way to the execution of the software, and includes factors such as response time, memory requirements and processing capacity.
- ***Integrity***: software accessibility control for unauthorized persons.
- ***Usability:*** the necessary effort to lean and use the software in a correct way

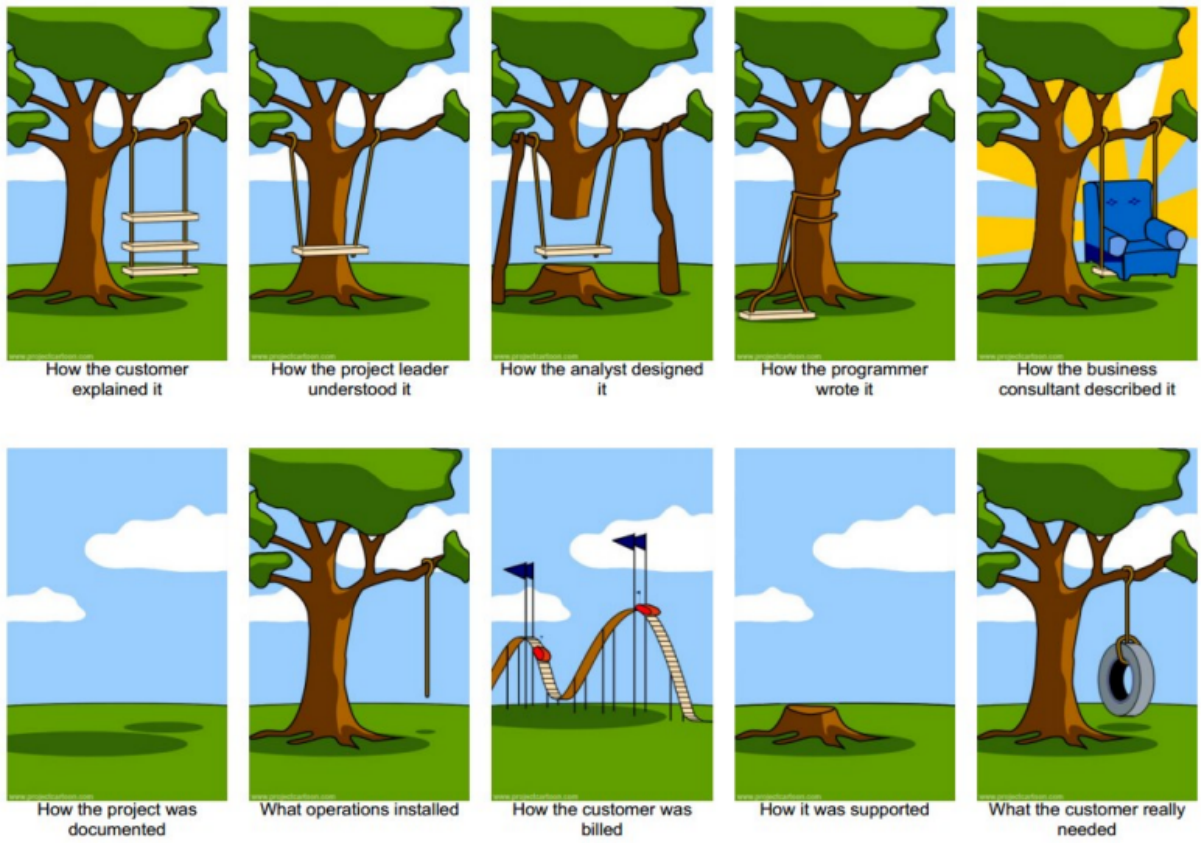
Revision Objectives

- ***Maintenance***: this is the effort required to detect and correct errors in a program that is installed.
- ***Testability***: it is the effort required to prove, to ensure that the system or module performs its purpose.
- ***Flexibility***: it is the necessary effort to modify a program once it is installed.

Transition Objectives

- ***Transportability***: it is the effort required to transfer the software from one hardware configuration to another.
- ***Reusability***: to what extent parts of the software can be reused in other applications.
- ***Interoperability***: it is the necessary effort to connect the system with other systems.

The main problem in SE is **communication**, so it is important to provide a clear idea, good documentation, diagrams…



The cost or complexity of development increases exponentially over time.

- For this reason, the complexity nowadays is very high.

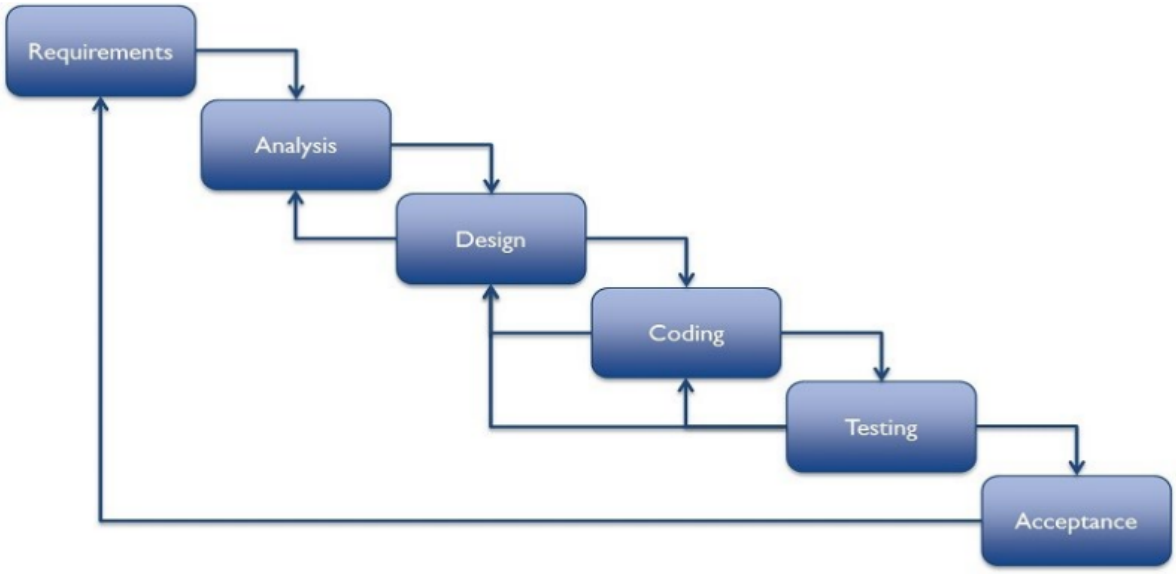
Software development Life Stages

The relevant stages in software development are:

- **Requirements**: The problem defined by the customer, written in human language.
- **Analysis**: Analysis of the requirements, ordered by priority, …
- **Design**: Decide which architecture, what parts, etc. fits better with the analysis.
- **Coding**: Implement what has been designed.
- **Testing**: Quality assurance to ensure the coding has no errors (issues).
- **Acceptance**: The customer must check that the implementation covers its requirements.

There are different ways (paradigms) of completing the software lifecycle:

- **Sequential linear model:** The classical life cycle of waterfall or cascading. You complete one step, then you go to the next one…<u>You can also return</u>



Advantages:

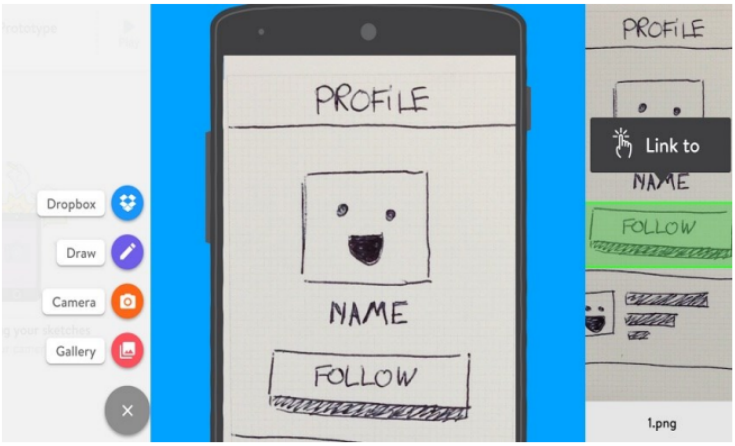
- Start-to-End of each defined stage → Easily measurable project progress
- Promote detailed documentation
- Signed agreement of system requirements. Because you are defining every task or phases of the project

Disadvantages:

- It is not flexible. Thus, it is impossible to make full sequential all the tasks of a project.
- A real project is never so markedly sequential.
- It's necessary to consider the possibility of returning to previous stages of development.
- Difficulty in explicitly establishing all the requirements at the beginning of the project.
- The client may not have requirements clear or may change during the process.
- The client must wait to see the result at the end.
- All planning is geared to a single day of delivery.
- It does not allow a stage development.
- Sometimes it would be better to do a part of the system and then improve it and complete it, but this can not be done if all the requirements must be known a priori.

- **Prototype construction model:** Centered on the UI to identify the functionalities. It is not reflecting the real User Interface, it is just an extraction.

  Why does it focus on the UI? Because the interactions between the user and the system reflects the functionalities that will be implemented in the software. Thus, it is a good starting point to model the functional requirements.

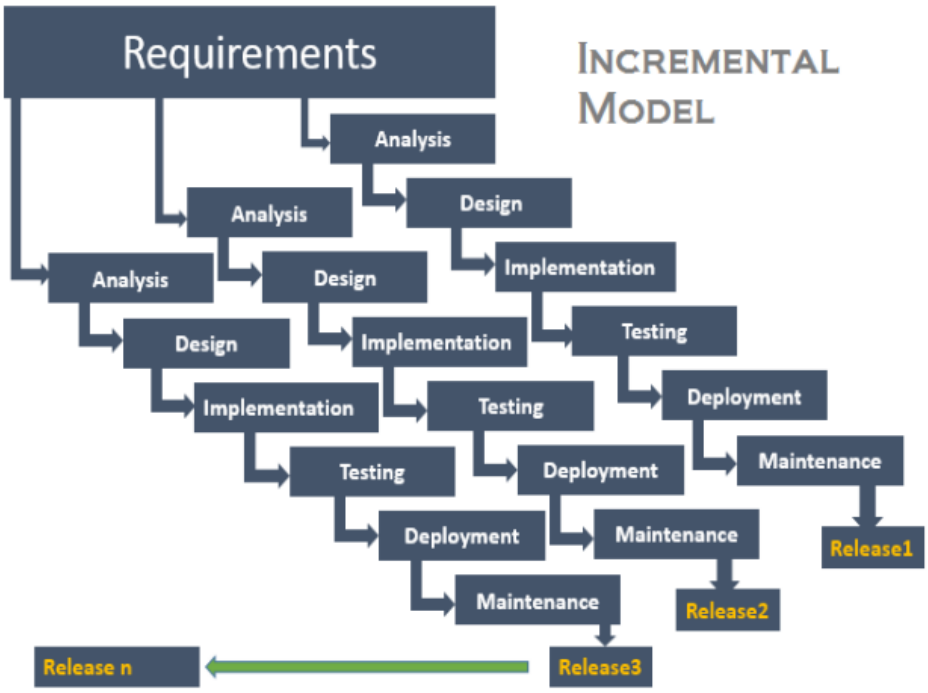


  Advantages:
  - More flexible than the waterfall model.
  - Improvement of the first three drawbacks of the classical life cycle.
  - It is developed from a set of known requirements (general objectives): collection of global objectives, rapid design, construction of the prototype, prototype test.
  - Enable two forms:
    - **In width:** all functions for limited or inefficient algorithms. Present a small part of every area covered by the software → Good to see all functional requirements
    - **In depth**: a subset of the final product, problematic functions. For a specific functionality, you present all the interactions → Good to see a complex functional requirement.

  Disadvantages:
  - The client may see it as a definitive software:
    - If it is very simple the user will say that we are really bad
    - If it is very good, the user will think that it is already implemented, which is false.
  - Prototypes can be made with inadequate resources: so the developer should not "get carried away" for comfort:
    - They will implement the software using the same tools used for the prototype → The software will be inefficient
    - They will use regular tools to implement the prototype → Doing too much work for a prototype, which is not good.

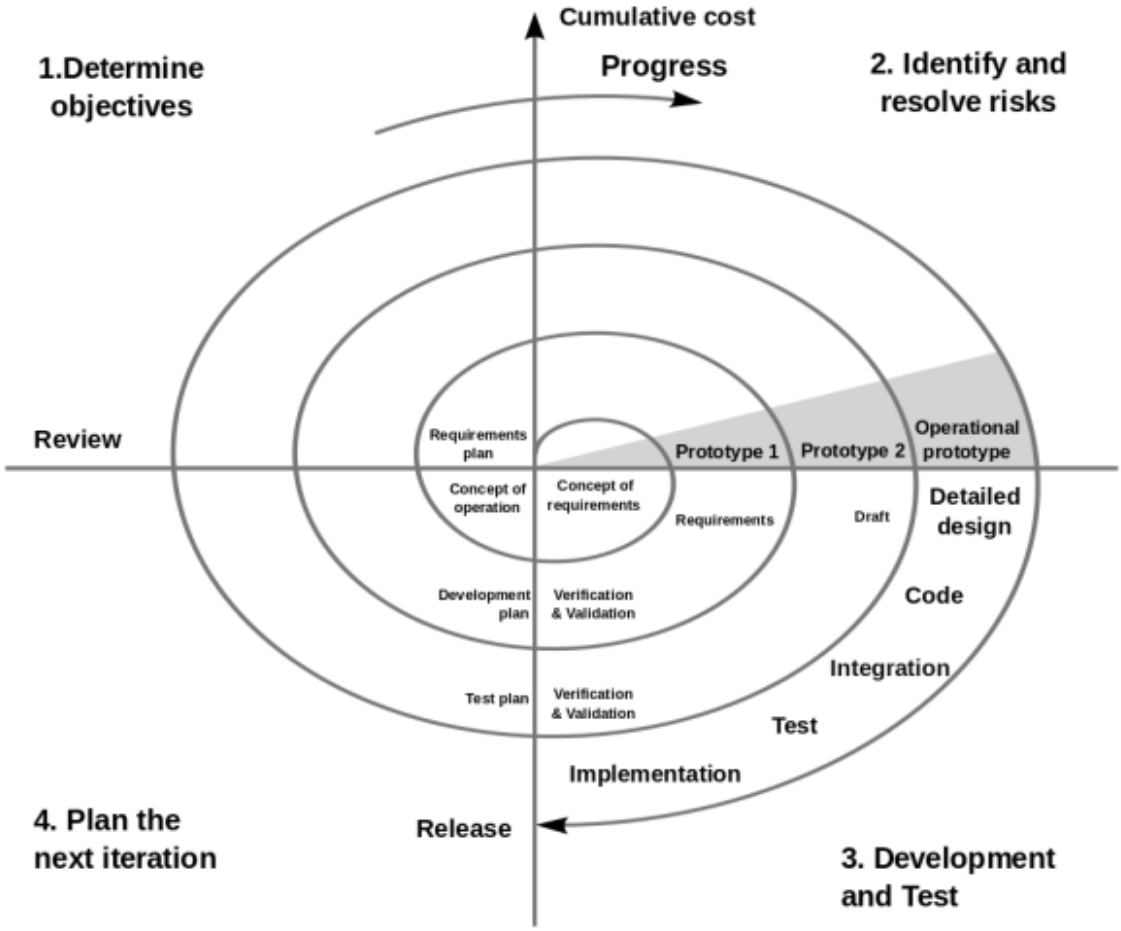
- **Rapid Application Development Model (RAD):** Construction based on short features additions into final project. It's a sequential linear model but with an extremely short time.

- **Evolutionary-Incremental model:** Takes into consideration the evolutionary nature of the software. It's an iterative model that allows the development of more and more complete versions:
    - **Assembling of Components:** Starting from an initial class library, combine those that are useful and add new ones that are added to the bookstore.
    - **Incremental/Concurrent development model:** Modifies the stages of the ES process as a set of states and transitions between them.
    - **Spiral:** Multiple cascading iterations choosing to implement some part in each iteration.
    - **Model of formal methods:** Allows a mathematical specification of the software.



As we can see, we have different releases of the same software. We cover every task multiple times.

- Start with a subset of the requirements
- Design how to implement them
- Release and review
- Add more functional requirements
- …



Advantages:

- The iterative model allows the development of progressively more complete versions, combining the advantages of waterfall and prototyping. So, we advance in the development.
- In each iteration a new feature is added, but always keeping the overall vision in mind.

Disadvantages:

- The end of the phases is not defined (never ends).
- The list of requirements is not closed until the end of the project.
- Risk control: a new change to be implemented may not be compatible with previous work, and this could cause a redesign.

Software development paradigms: Agile

Manifest (principles) of the Agile Software Development:

- People and iterations over processes and tools.
- Intuitive and functional software over extensive documentation.
- Collaboration with the client over contract negotiation.
- Response to the change over following the planification.

Different methodologies that follow Agile Software Development principles:

- SCRUM
- KANBAN
- LEAN
- XP

These methodologies come as a result of the experience from an evolutionary spiral model. They said that it is good to use that model but that we need to make clear some important things to achieve all the phases without risk.

As we have seen, they cover the communication between product owners, developers, how to complete the meetings, how long each iteration should take, how many functionalities we need to implement in every release…

The three main phases of the software lifecycle:

- **Analysis**
- **Design**
- **Implementation**

Analysis

Phase to describe the functionalities that need to be implemented in the system (functional requirements). So, we want to describe all the system requirements.

Understand the problem and represent its basic elements (requirements), describing each one of them and creating the relationship links between these elements.

To do this, we need to define the different use cases → Description of the interaction between actors (anything that provides inputs to the system. Human, computer, server…) and the system.

During the analysis, the characteristics of the problem are described in terms of requirements and use cases. In this phase, the what is described.

All this information (functional requirements) will be documented in the Software System Specification.

ACTORS → USE CASES → IDENTIFY FUNCTIONAL REQUIREMENTS

Special concepts:

- **Requirement**: Set of ideas that the client has about what the software to develop should be. They represent the behavior of the system.
- **Use Cases:** A use case is a list of actions or event steps typically defining the interactions between a role (an actor) and a system to achieve a goal.
- **Actor**: Entity that will use the software interactively or causing inputs to them.
- **Software Specification**: Official technical document listing and categorizing the Requirements and Use cases

Design

How to implement the system requirements to achieve this functionalities identified in the analysis. So, to complete the design, we only need to describe the physical implementation. In other words, describe the architecture of the system.

We are going from the problem domain (described in the analysis) to the solution domain (your particular implementation of the requirements).

Produce a model or representation of the system that can be used, at a later stage, in order to implement it. **It is not a full implementation, it is just a model.**

During the design, we use our experience and intellectual skills to generate a model of the system. In this phase we describe the how.

Definition:

- The process of applying different techniques and principles in order to define a device, process or system with sufficient levels of detail to allow its physical realization.
- It is at the technical core of the S.E. process. And it applies regardless of the development paradigm used.
- Problem domain → Solution domain

Objective:

- Produce a model or representation of the system that can be used, at a later stage, in order to implement it.

Design Levels

The results of the design phase can be depicted at different levels.

- **Preliminary (done at the end of the analysis phase)**: Focus on requirements transformation and software architecture (classes, sequences).
  We should never provide a result of the analysis without a preliminary design.

- **Detailed (done at the end of the design phase)**: Data structures refinement and algorithmic representation.

A good design requires achieving a sufficient degree of these objectives:

- **Verifiability**: To be able to evaluate the correctness of the design.
- **Completeness**: All components (data structures, modules, external interfaces, etc.) must be specified.
- **Consistency**: That there are no inherent inconsistencies.
- **Efficiency**: Proper use of system resources. The resources are not infinite.
- **Traceability**: All design elements must be able to "map" towards the analysis of requirements. → verification matrix
- **Simplicity / Comprehensible:** It should be noted that the design document, like the requirements specification, will be used as the basis for the subsequent stages of the design process.

What design does and does not involve:

- The design process should not be unique, alternatives should be evaluated.
- Each element of the design model must be able to go back to the analysis to find out what requirements it satisfies.
- Design does not have to invent anything new. Reuse as much as possible.
- Minimize the distance between the problem domain and the solution domain.
- It must be uniform (a constant style).
- It must be structured to admit changes.
- It must be robust. Accept unusual circumstances.
- The design is not writing code and writing code is not designed.
- The quality of the design is evaluated while it is being made, not afterwards.
- It must be reviewed to avoid conceptual (semantic) errors: omissions, ambiguities, inconsistencies, etc.

Special concepts:

- **Abstraction**: Something that you describe before a particular level.
  If we describe the communication between 2 different models, we need to make an abstraction to exchange the communication. Generic level vs. Concrete level.
- **Modularity**: Subdivision of the system into elements. Necessary because the components of the software is very complex and you need to divide them in small tasks.
- **Refinement**: Formal verification of the specification implementation.
- **Software Architecture:** Global structure of the software and the way in which this structure provides conceptual integrity to a system.
- **Structural partition:** The program structure can be divided horizontally or vertically.
- **Data structure:** Logical relationship between individual data elements.
- **Software procedure:** Processing details of each module individually.
- **Principle of concealment (of information):** The modules must be designed so that the information (data and processes) is inaccessible to other modules that do not need it.

Level of abstraction has a major impact on the design phase.

- **Abstraction**: Allows components to be defined in abstract form at different levels, without worrying about their implementation details.
    - An abstraction describes the external behavior of the component, without having to pay attention to the internal details that produce the behavior.
    - The abstraction is an aid to modularization, since it allows to see the external behavior of the modules so that they can be interconnected.

- Three models:
    - **Functional abstraction:** The system requires to do this, this and this. From the point of view of the function it performs. A sequence of instructions that has a specific and limited function.
    - **Data abstraction:** The functionalities described in the functional abstraction need to be applied to some data that is described in the data abstraction. A collection of data that describes a data object.
    - **Control abstraction:** We also need to control how to apply these functions to the data. Implies a program control mechanism without specifying the internal details

If we cover the 3 areas, we are completing the design.

Process Tools: Software Specification Document

The analysis is provided for in what is known as the Software Specification Document (also called the S.R.S. or Software Requirements Specification):

- This well organized (you have different sections) and written document includes all the system requirements as well as the use cases.
- It acts as a contract to be validated by the client with respect to all the functionalities to be provided by the system.
- It also includes all the necessary constraints (**related to the non-functional requirements**) that are due to causes of the problem domain and not of the design. For this reason you provide an initial architecture (high level description of the design) in the preliminary phase (end of analysis phase)

Process Tools: Design Document

The design is provided for in what is known as the Design Document.

- This document outlines the overall vision of the design: the elements/components, their characteristics and relationships between them. Both at a preliminary level and at a detailed level.
- It defines how the overall structure and its behavior will be implemented → keeping in mind that everything must be derived from the requirements that appear the Software Specification Document!
- To perform this design task and part of the analysis we need to use some methodologies and tools: **UML** (used to make diagrams) is the most widely used at present.

Implementation

Generate the code and validate it.

Conclusions

1. Software engineering principles help to create quality software in a systematic way (efficient software created with efficiency). We can use different methodologies.
2. They can be applied at different stages and for different goals.
3. Each software engineering methodology has its own steps to follow the principles.

https://www.geeksforgeeks.org/software-paradigm-and-software-development-life-cycle-sdlc/

Explanation Exercise 2:

The goal of this exercise is to start working on the creation of the Software Specification Document (SRS) for a new product.

The SRS document describes **what** the software will do and **how** it will be expected to perform. It also describes the functionality the product needs to fulfill the needs of all stakeholders (users).

An SRS helps to:

- Define our product's purpose and scope.
- Communicate our vision and expectations to the development team.
- Organize and prioritize the requirements.
- Track and verify the progress and quality of the product.

Following a standard structure for the SRS, the main sections are as follows:

- **Introduction**: Provides an overview of the document, its purpose, scope, definitions, acronyms, references, and assumptions.
- **Overall Description:** Provides a general description of the product, its perspective, functions, user characteristics, constraints, assumptions, and dependencies.
- **Specific Requirements**: Provides a detailed description of the functional and nonfunctional requirements of the product, as well as the use cases that illustrate user interactions with the system.
- **Appendices**: Provides any additional information that may be useful for understanding or developing the product, such as data models, diagrams, prototypes, etc.

Block 2. Software Management principles

Why do we need management?

Key point because of the risk of high cost.
The problem is that:

- As we have seen, software is getting more and more complex. Therefore, development is consuming a lot of money and time.

So, we need to be faster. To achieve this, we have changed the methodologies (Agile). This typically implies more effort, because we are doing more work in less time.

- We are using more people for development.
- Having a lot of people doing the development is very difficult, because the complexity of the management goes crazy.

So, we are trying to use less people, be faster, cheaper and produce good software.

Software engineering costs

Project development needs to be monitored because the risk of high costs are real:

- If the customer gets angry, then a lot of money is wasted and software satisfaction is low.

Therefore, the cheaper, faster and better quality, the more successful we will be.
The objective is then to reduce costs through good management.

Software engineering management paradigm comparison

In this table, we just need to look at the amount of advantages and disadvantages.

	Advantages	Disadvantages
Waterfall	• Start-End of each defined stage → Progress of the measurable project. • Promote detailed documentation. • Signed agreement of system requirements.	• A real project is never so markedly sequential. It is necessary to contemplate the possibility of returning to previous stages of development. • Difficulty in explicitly establishing all the requirements at the beginning of the project. • The client may not have them clear or may change during the process. • **The client must wait to see the result at the end.**
Protyping	• Improvement of the first three drawbacks of the classical life cycle. • It is developed from a set of known requirements (general objectives)	• The client sees it as definitive software. • Prototypes done with inadequate resources
Incremental	• Iterative model that allows you to develop more and more complete versions, combining the advantages of waterfall and prototyping. • In each iteration, a new feature is added but always take into account the general overview.	• End of phases not defined. • Requirement list not closed until end of project.

Incremental model (Agile model) has less disadvantages.

Every model follows a different paradigm, but all use a management model.

- I need to use an Agile model to increase the quality with less effort.

We can not mix them.

Management versus Paradigms

Our development can follow several strategies (paradigms):

- Models of software development:
  - Lineal-sequential (Classic life cycle)
  - Evolutionary (Iterative or Incremental)
- Classics methods (lineal-sequential) have some inconveniences:
  - Planning phase requires a huge effort
  - In environments with fast changes, the specification of requirements can be hard
- So, evolutionary methods are currently recommended:
  - Leading model: Agile Software Development

But regardless of the strategy, control must be guaranteed.

Configuration Management Control → Repository

Since it is relevant to have a good control, we need to introduce the concept of the configuration Management Control. This is done using a repository.

The repository is the central point where you Store all your project.

All developments need a process to establish and maintain the consistency of a product (in our case a software solution), during the whole development lifecycle and its deployment. This process is known as Configuration Management.

Consistency involves two areas:

- The items (elements): The results, documentation, source code...
- And the control (the procedures): Procedures to create the items.

Both areas are essential, so we target on each.

So, we are focusing in the elements that we are storing and the procedures to store these elements in the repository.

Configuration Management: Items

In the field of software engineering, all items/elements must be under control. For exemple:

- **Documents**: Technical documents, diagrams, meeting notes, guides, …
- **Code**: Source code, test code, releases, …
- **Reports**: Issues, bugs, feedback, …
- **Others**: And any other material involved in the software development process.

The repository is not only a storage or hard drive... it also maintains under control all the items in the repository. This involves:

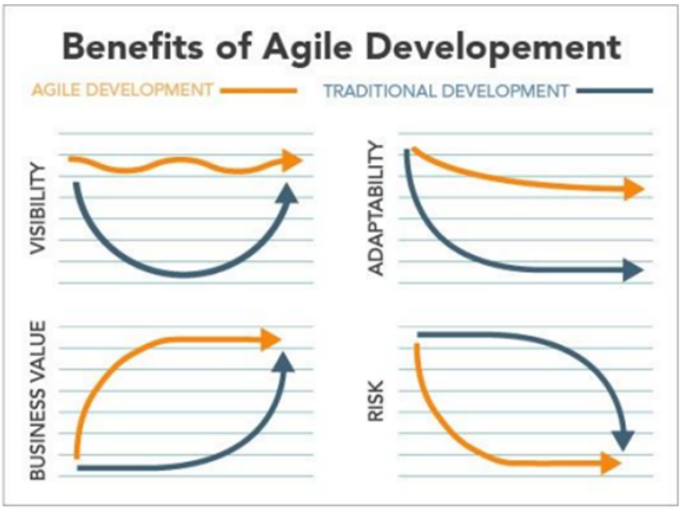
- Identify each element: You can not consume a lot of time to recover something from the repository.
- Versioning the elements.
- Trace changes between versions.
- Enumerate the people involved.
- Add a timestamp to each item to recover previous releases.
- Centralize the information.

Therefore, the solution to centralize the management is to use a Repository.

Do not mix several repositories, use only one.

Why are Agile methodologies successful?

Here we have a diagram that compares Agile vs traditional development methodologies.



In all areas, Agile methodology is better.

SCRUM methodology follows the Agile model.

The Agile model was created by some experts because they realized that they need to change something. It is not possible that to develop a software, we need to spend 1 year, 50 people, a lot of money…

They proposed some changes. The manifest (principles) of the Agile Software Development:

- **People and iterations over processes and tools**
    - It is not relevant the process and tools used in the methodology. The methodologies do not improve the quality of the software.
    - We need to focus on the people that are doing the software. 3 good programmers (**expertise**) are sufficient to obtain good results.
    - We need to focus on the iterations. Make smaller iterations with smaller increments in functionality and this will produce better software.
- **Intuitive and functional software over extensive documentation.** By doing a good documentation, you are not improving the software (this does not mean that the documentation is not important). Thus, we prefer a good software and a bad documentation over a good documentation and a bad software
- **Collaboration with the client over contract negotiation.** Make a lot of meetings and having a good feedback from the client is better than having a good contract.
- **Response to the changeover following the planification**. A lot of the initial ideas are reconsidered and changed.

Note that Agile Management is a development model, not a methodology!

Many people have created new methodologies based on this Agile model.

Implications for the 4 principles of the Agile Software Development Manifest:

1. Our highest priority is customer satisfaction through early and continuous delivery of valuable software.
2. We embrace changes in requirements, even late in development. Agile processes exploit customer changes for a competitive advantage.
3. Deliver functional software frequently, from a couple of weeks to a couple of months, with a preference for the shorter timeframe.
4. Employers and developers should work together on a daily basis during the entire project.
5. Build projects around motivated people. Give them the environment and support they need, and trust them to get the job done.

6. The most efficient and effective method of transmitting information within and to a development team is the face-to-face conversation.

7. Working software is the primary measure of progress.

8. Agile processes promote sustainable development. Customers, developers, and users must be able to maintain a constant rate indefinitely.

9. Continued attention to technical excellence and good design increases agility.

10. Simplicity —“the art of maximizing the amount of work not done”, in the sense of eliminating the unnecessary— is essential.

11. The best architectures, requirements and designs emerge from self-organizing teams.

12. At regular intervals, the team thinks about how to be more effective, and then refines and adjusts its behavior accordingly.

The most important is that:

- You need to make changes, you need to make meetings, provide a constant rate in the development, simple (less people) and efficient.

SCRUM methodology

The SCRUM methodology follows the Agile model with the regular benefits of:

- Minimization of risk → short iterations.

- Real time communication (face-to-face) → a few written documentation. Have Meetings with one person that represents the user or client.

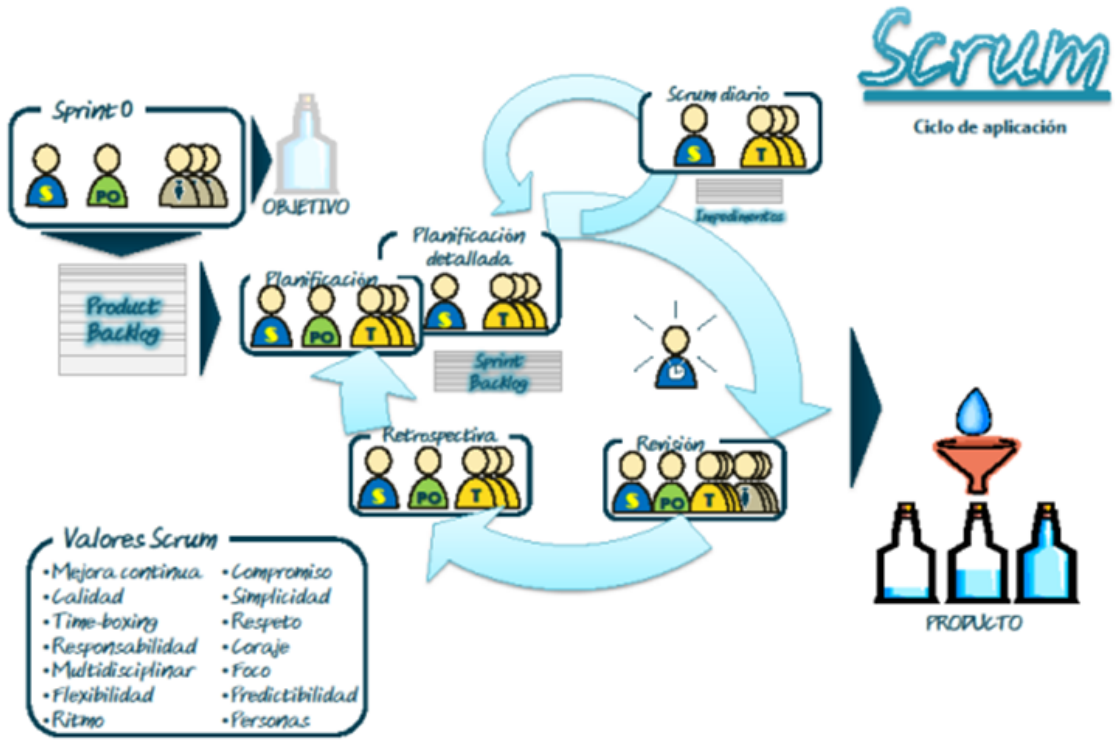
- Make all the work together with all the people at the same level in the development chain.

- Indicated for unpredictable requirements.

General characteristics of the SCRUM methodology:

- **Agile environment** for the development and maintenance of complex products:
    - It allows to tackle complex problems, release products with the maximum value and the shortest period of time.
- **Incremental** and **iterative** process:
    - It allows to develop products where requirements change rapidly.
- Based on teamwork, so a **self organized** team:
    - Improves communication and maximizes cooperation.
    - Maximizes productivity.
    - Protects the team from interruptions due to impediments/difficulties.
- Controls the chaos due to conflicts of interest and needs

SCRUM Architecture



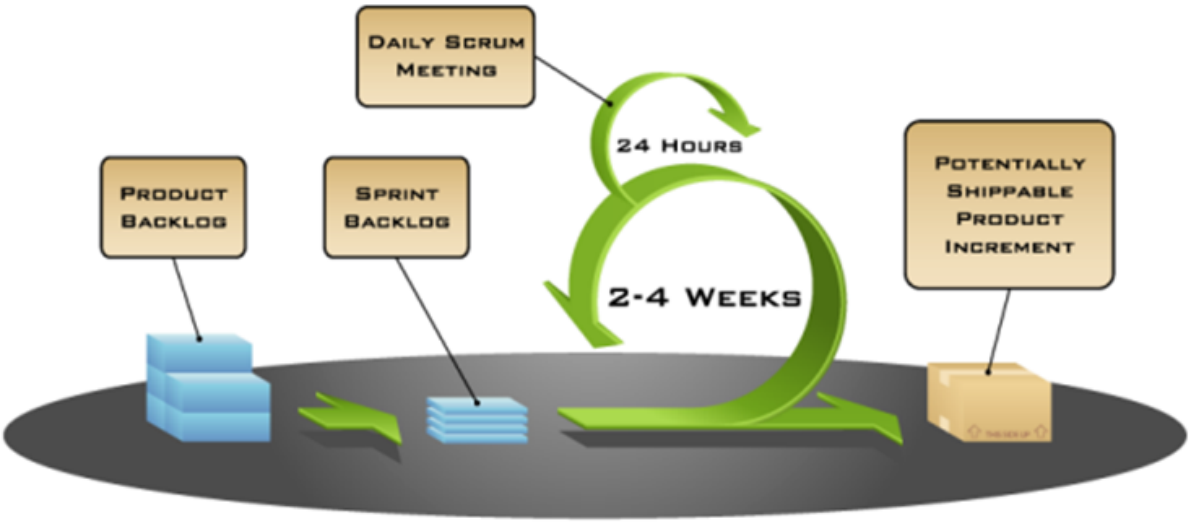
You start with the people involved in the project, you create the product backlog and you enter the development cycle (sprint):

- Selecting some part of the product backlog, that we will call sprint backlog.
- Use the working team to produce or achieve this sprint backlog.
- Every day you do a small meeting
- Before you finish, we do a retrospection of the work done in the “sprint”
- So, we are doing the software by doing small increments. Finally you obtain the full version of the software.

SCRUM Foundations

- It is based on an empirical process control.
    - Knowledge comes from experience.
- Scrum uses an iterative and incremental approach to optimize the predictability and risk control.
- General fundamentals:
    - Transparency : The process must be visible to everybody. The same understanding of the product and the “facts” is shared.
    - Review : We must inspect the artifacts and progress towards the goal.
    - Adaptation: If a process is detected that deviates from the acceptable limits, it must be readjusted.
- Elements (components): This is the relevant part
    - **Roles**: each person has one or more functions.
    - **Artifacts**: results of the sprint.
    - **Processes**: meetings in each sprint

SCRUM Functionalities



This is another view of the SCRUM architecture. The sprint backlog needs to be done fast (less than a month)!

Daily SCRUM meeting: Every day, before starting to work, independently we review your advance from the previous day.

Finally, we are producing increments in the software.

The participants in the project have different roles:

- **Product owner:** vision (connection) of the customer / user / client (only one person is the product owner)
- **Scrum master:** One person from the development team that has the role of coordinating other members of the scrum team. It is not the boss or the person with more experience. It is a person with sufficient skills to make the management (it could be changed if necessary). It makes the communication with the product owner.
- **Scrum team:** working team (executes tasks). Involved people in the project, like the developers (they are all at the same level).
- **Stakeholders:** customers, users. Not the developers

The requirements are saved as elements (items) in a list called “product backlog”:

- The list is prioritized and the effort is estimated (duration)
- These elements are called “user stories”.

The product progresses in small increments of fixed time (<1 month), called “sprints”:

- The functionalities to be implemented are selected → Sprint backlog
- A short daily meeting is made (15min) → Daily scrum meeting to solve the problems
- The product is designed, programmed and tested during the sprint

No planning changes are made during the sprint!

So, we have the product backlog and in each iteration we select something from the product backlog (done within the SCRUM team, product owner and SCRUM master) and then the SCRUM master and team prepare how to achieve this sprint backlog.

- The SCRUM master assigns the different tasks to different persons

Therefore, starting from the theory behind the SCRUM methodology:

- Scrum is not a process or technique; it is an environment where different processes and techniques can be used.
- The scrum rules establish relationships and iterations among the Scrum components.
- Scrum is light, easy to understand… but very difficult to master!

Try to understand the concept and apply it in a simple way.

SCRUM Roles

Product owner. Represents the client:
- He acts as a single voice
- He has the vision of the product

Responsible for requirements:
- It's the owner of the product backlog and therefore, he is the only one than can change the Product Backlog
- Change and re-prioritize the product backlog before of each sprint

Accepts the software
Development team implements what the client wants.

Scrum Master.
Manager, responsible of the process
Makes easier to make meetings, and monitories the progress

Helps the team:
- Remove drawbacks to the team
- Ensures the productivity and isolates the team from distractions
- Can make changes in the Sprint Backlog

He is the connection between the product owner and the scrum team
Interacts with the rest of the organization

Scrum Team (development team)
Scrum team: 3-9 members that develops the product
It is an autonomous team

Multi-functional:
- Each member has expertise

Self-organized:
- There is no default role, tasks are distributed taking into account the pending tasks and experience of each person.

Stakeholders (parts with interest), users, customers

SCRUM Artifacts (elements)

The Product Backlog is a requirements list:

- It's an approximation → So, it's not accurate and can change
- The requirements are prioritized

Belongs to the Product Owner → s/he is responsible for managing it

- S/He can change (add/remove, change priorities) before each sprint

Estimation of the elements of the product backlog:

- The speed of the team development is estimated.
- Effort is estimated: hours/days

The Sprint Backlog is a subset of the Product Backlog:

- Defines the work that will be done in a sprint
- If a task is very long, it must split

It is created only by the Scrum Team

- The team can add/remove elements to the list
- Product Owner is not allowed to do anything with this list

The elements of the Sprint Backlog have 3 dimensions:

- Priority (more or less importance)
- Detail (breakdown of tasks)
- State (pending, doing, paused, done). It should be updated every day

Progress charts:

- Burn-up: Product backlog with things completed
- Burn-down: This represents the effort that remains to complete the project.

Product backlog with things remaining to be completed

SCRUM Processes and Phases

Project kick-off meeting:

- It is a meeting before starting the project
- The requirements list is created, some of the requirements are selected and prioritized → the result is the product backlog

Sprint planning meeting:

- The requirements (from the backlog list) to be developed in this sprint are selected
- The tasks are also determined in a list: sprint backlog

Sprint:

- Daily sprint meeting

At the end of the sprint → sprint review:

- The following sprint is defined
- The unmade tasks pass to the product backlog
- It is decided whether to do a product increment or a release
- At the end of the sprint → sprint retrospective

SCRUM: Sprint Planning meeting

Collaborative Meeting at the start of each sprint:

- Participants: Product Owner, the Scrum Master and Scrum Team
- It can last up to 8 hours and consists of two parts
- The requirements and priorities are defined
- The tasks to be developed in the sprint are defined

1st Part:

- Create the Product Backlog (requirements list)
- Determine the Sprint Goal (objectives)
- Participants: Product Owner, Scrum Master, Scrum Team

2nd Part:

- Create Sprint Backlog (subset of requirements to do in the sprint)
- Participants: Scrum Master, Scrum Team

SCRUM: Sprint

An iteration or increment, of less than a month, where the functionality of the product is increased

No external influence can interfere with the team during the sprint

Each sprint starts with the Daily Scrum Meeting

Here (in the sprint) the software development work is done

SCRUM: Daily SCRUM meeting

It is done at the beginning of the day (maximum 15 minutes)

Each team member answers 3 questions: − What he has been done? − What drawbacks he had? − What is he going to do today?

This allows the scrum master to monitor progress, and in case of problems:

- Plan work not identified in previous days
- Re-assign tasks
- Eliminate problems

Everyone is invited → avoid other unnecessary meetings:

- But only the product owner, scrum master, scrum team can talk

It is a meeting where team members make commitments:

- It is not a session to solve doubts
- It is not a way of knowing who does the tasks on time

SCRUM: Sprint Review meeting

It is done at the end of the sprint

Sprint review → Objectives achieved?

- The team presents what has been achieved during the sprint
- The new functionalities of the product are usually presented to the product owner with a demo

It is informal (less than 2h)

Participants: − Clients − Managers − Product Owner − Other engineers

SCRUM: Sprint Retrospective meeting

Sprint retrospective meeting:

- Sprint summary
- Only participate the Scrum Team

The estimated speed and the actual speed are checked : burn-down

Feedback from the meeting: − What it has been done well? − What it should be improved? − What it has been improved?

SCRUM: The Task board

The Task Board is a tool to organize / manage and monitor the tasks of the backlogs:

- It is a 2-dimensional matrix with rows representing User Stories and columns representing various Status values:
    - Tasks to do
    - Tasks in progress
    - Tasks done
- The objective is to provide immediate visibility of development status.

The Task Board is an abstraction used in several different methodologies.

Level of detail

- Simple
- Regular with user histories
- Complex: Includes the sprint and product backlog. So, we are fragmenting the product backlog into simpler tasks (sprint backlog)

Useful for managing tasks:

- In addition to the status can store other information: priority, etc.

Useful for assigning tasks:

- Users have to take responsibility for specific tasks.
- One person → One task

Useful for controlling tasks:

- The necessary effort must be allocated.
- Progress must be monitored.
- In case of problems, tasks must be reallocated

Conclusions:

1. The repository is a central point for management.
2. SCRUM simplifies management with the Task Board and productive meetings.
3. The Scrum Team coordinates, communicates, solves problems and improves continuously.

Block 3. Software Modeling (Software Analysis)

In this session we will discuss how to generate the content within the ‘Software Requirements Specification’. And we will also focus on the correct way to generate the Functional Requirements.

- Introduction to software requirements
    - Types of requirements
    - Functional and non-Functional requirements

- Understanding of the problem domain
    - Requirements gathering
    - Modeling methods

- Requirements specification
    - The S.R.S. document
    - Review and validation of the requirements

Why do we need an analysis phase?

Because we need to create a solution (software) that resolves certain problem

Analysis Objectives

As the complexity of the problems increases, it is necessary to perform a software requirements analysis phase to achieve the following two objectives:

- Understanding the problem
- Generate the requirements specification

And what is a software requirement?

- Requirements: Set of ideas that the client has about what the software to be developed should be and its behavior. They are the characteristics of the system.

Are description of features and functionalities of the target system. Therefore, we need to understand the problem and identify the requirements

Types of requirements

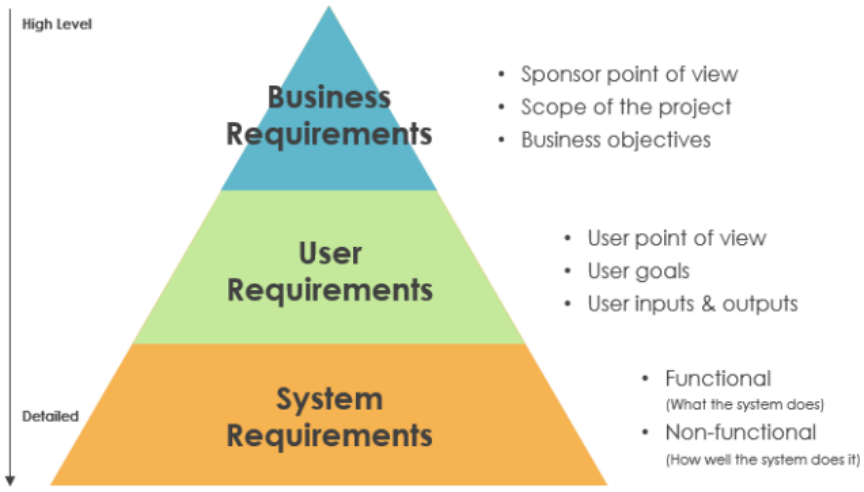
The requirements abstraction pyramid:

- Business Requirements:
  - Generic objectives of the client.

- User Requirements:
  - Natural language description of the services to be provided by the system and its operational limitations. Written by clients.

- System Requirements:
  - Structured document that provides a detailed description of the functions, services and operational constraints of the system. It defines what needs to be implemented so that it can be part of a contract between the customer and the developer. Written by engineers.

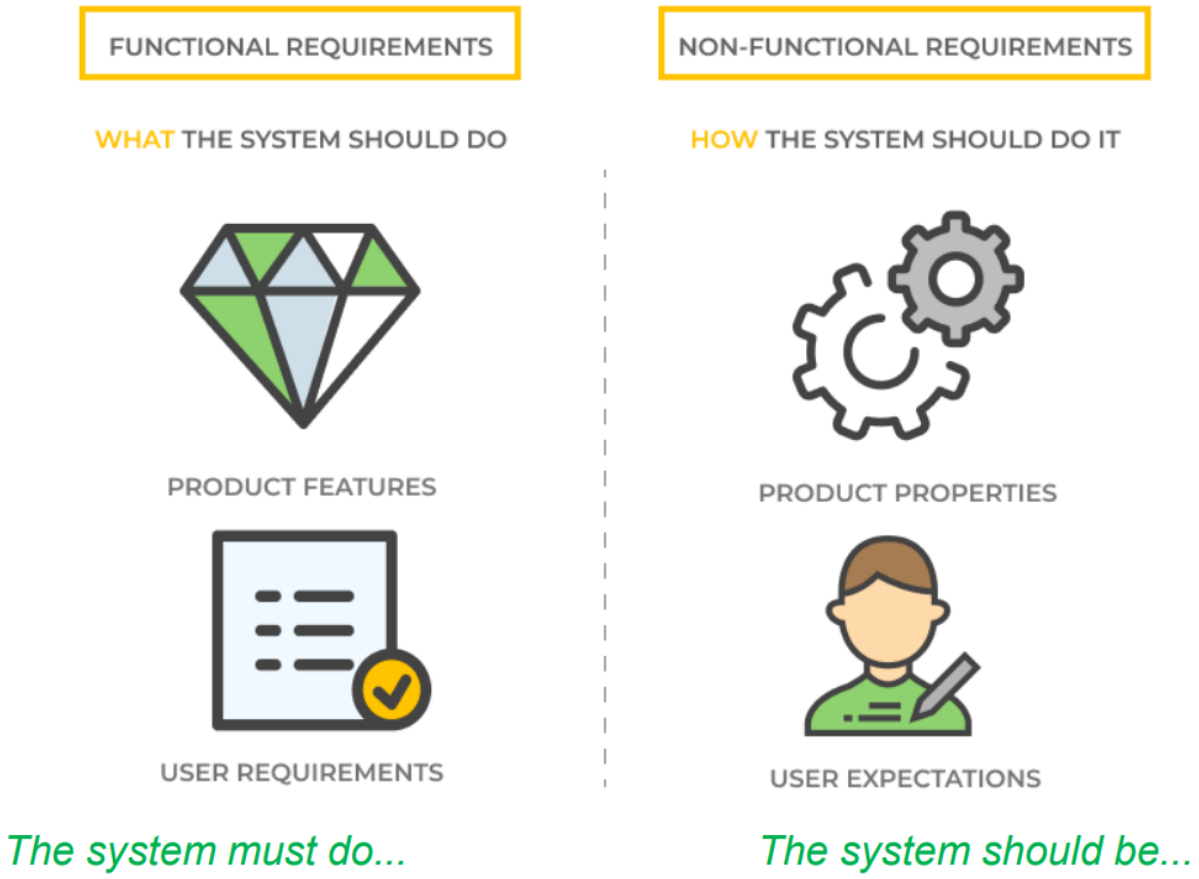
From the user requirements we will generate the system requirements.



System Requirements are classified into 2 main categories:

- **Functional requirements:** They define functions and functionality within and from the software system.
  - Detail product features
  - Describe the work that is done
  - Describe the user actions
  - Characterized by verbs

- **Non Functional requirements:** They are implicit or expected characteristics of software, which users make assumption of.
  - Detail product properties
  - Describe the character of the work
  - Describe the user experience
  - Characterized by adjectives



Functional Requirements (FR)

- Description of the desired behavior for the software.
- Each functional requirement describes, in general, the relationship between inputs and outputs → Given an input or condition, what is the process to be performed to produce a result.
- It is also necessary to describe the expected behavior when unexpected inputs or conditions (errors) occur.

Functional Requirements unambiguously describe the features of the system.
Example: "The system must allow the user to submit feedback through a contact form in the app."

Non-Functional Requirements (NFR)

- Non-functional requirements are constraints proposed by the customer or by the problem itself, related to the design to be implemented later.
- They are generally quantifiable (aka measurable).
- We can find several different categories:
    - Performance related
    - Design related (constraints, objectives, decisions)
    - Related to external interfaces

Non-Functional Requirements limit the freedom of design.
Example: "When the submit button is pressed, the confirmation screen must load within 2 seconds."

NFRs: Performance

There are two main groups of performance requirements:

- Static requirements (capacity):
    - Limits on execution characteristics (number of users, memory used, disk space, etc.)
    - Example: The web server will work for 100 simultaneous users.
- Dynamic requirements:
    - Limits on the behavior while the system is running (number of connected users, response time, performance, etc.)
    - Example: The web server should provide a response in less than 3 seconds.

NFRs: Design

There are three main Non-functional Requirements related to design:

- Design constrains: when design freedom is restricted in these areas:
    - Standards enforcement: formats, procedures, audition, etc.
    - Hardware limitations: computers/servers, O.S., language, licenses, storage, etc.
    - Errors management: tolerance to errors, recovery from errors, high availability, etc.
    - Security: access policies, activity logging, etc.

- Design objectives: general guidelines that apply to the entire design
    - Usually related to the quality of the product: usable, user friendly, maintainable, etc.

- Design decisions: when a specific design decision has already been made
    - For instance, implementation details about some algorithms, other low-level aspects…

Each group is completely different, so don't mix them up!

NFRs: External interfaces

Requirements on external interfaces define the interaction characteristics between the system and people, hardware and other software modules:

- With persons are the restrictions imposed on the U.I.
- Hardware access protocols are described for the equipment
- In the case of external software, specify the restrictions to be complied with

Summary Note:

- Five types of NFRs → Performance, Design Constrains, External Interfaces, Design Objectives & Design Decisions

NFRs: The metrics

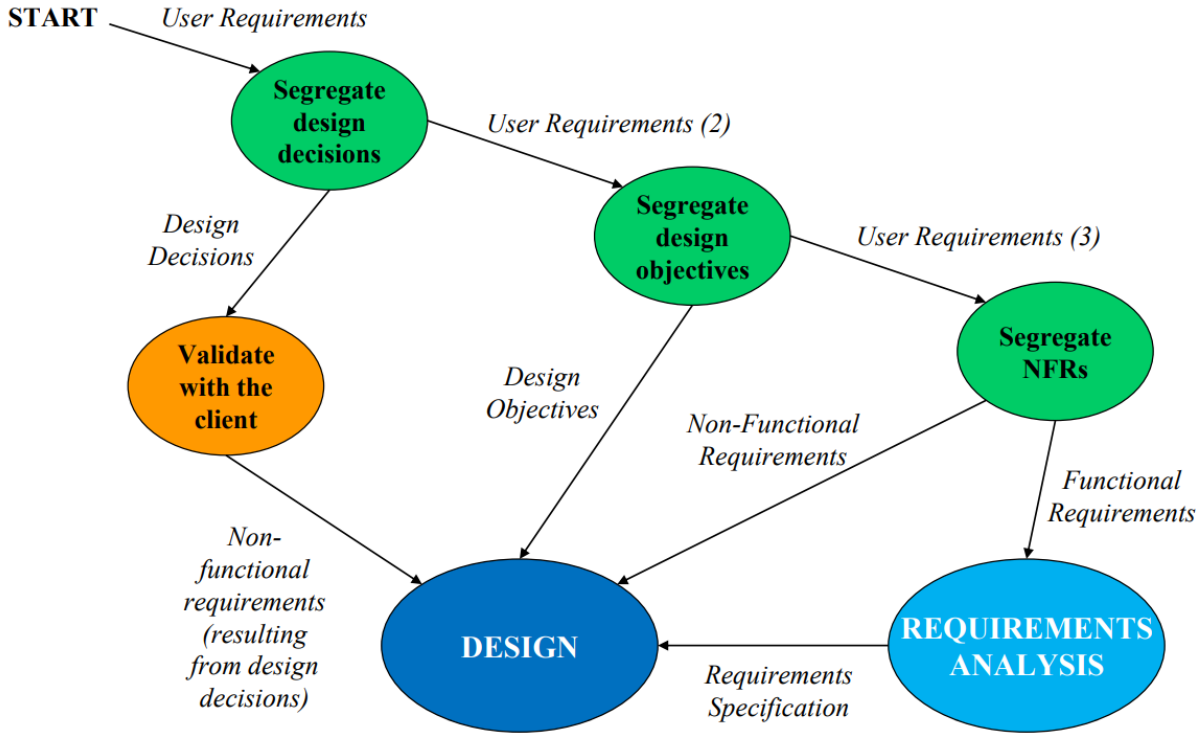
Non-functional requirements are usually quantifiable:

- They are only well defined when we can expose a metric for them.

Examples of common metrics

Property	Metric
Speed	Transactions/sec. ; Response time/requests ; screen refresh time
Length	MBytes ; Number of files
Easy to use	Time to learn ; Number of help pages
Reliability	Mean time between failures ; Error ratio ; Availability factor
Robustness	Recovery time ; Probability of data corruption in failures
Portability	Number of supported platforms ; Percentage of target dependencies

Requirements: Identification process



Requirements: Process

It is a four step process, which includes:

- **Feasibility Study:** The output of this phase should be a feasibility study report that should contain adequate comments and recommendations for management about whether or not the project should be undertaken.
- **Requirement Gathering:** Identification of the requirements
- **Software Requirement Specification**
- **Software Requirement Validation**

Requirement Gathering

Requirements elicitation (Req. Gathering or Capture) is the process of generating a list of all requirements from different sources (clients, users, etc.). This task involves gaining a deep understanding of the problem domain.

Some of the methods to obtain the requirements are:

- Interviews
- Surveys
- Documentation review
- Observation of use
- Prototyping
- Brainstorming

Several different methods can be used at the same time

Interviews

Direct interviews with the client are a primary and direct source of information:

- From the customer's point of view we obtain the user requirements that we will transform into system requirements.

First action to be taken. It includes:

- Talk with the client and write down the problem, the wish list of solutions, doubts, questions, etc.
- After the first iterations, at an advanced stage, we can enter the "Join Application Development" (iterative meetings).
- These are several meetings every 3 or 4 consecutive days with all customers (stakeholders), some end users and some development teams in order to improve the analysis and design.

Surveys

Another direct way to receive customer feedback is through surveys:

- Surveys are well-structured written documents used to obtain specific information.

If this task is carried out, it should always be done after the first interviews. Particularities:

- Useful for retrieving information from many people in a short time.
- Useful in situations where it is not possible to meet with people (geographic dispersion).
- It will require a lot of effort and time.

Documentation review

The documentation review is performed by the development team itself:

- It is the internal task of searching for information about the problem domain.

The research of pertinent data is carried out in the following fields:

- User manuals of previous solutions.
- Standards and regulations.
- Experience gained from other projects
- Market studies.
- ETC

Observation of use

Live observation of the problem domain is another possible source of information:

- The task is to externally investigate how the system currently works, without the influence of the client's thinking.

This task includes:

- Experience the actual operations on the customer side.
- Live the problem from the users' point of view and not from the managers' point of view.
- It is important to consider that sometimes the actual operation does not conform to the protocols established by the company.

Prototyping

Build an abstract model of the application that allows the customer to 'play' with it and better detail their needs:

- The prototype is an example, not a final product.

Two types or phases with prototypes:

- Paper schematic, describing the human-machine interaction.
- Executables, in two subtypes:
  - In with: All functionalities covered with limited or not complete implementation with dummy data and parameters.
  - In depth : A subset of functionalities covered with high implementation, in general the most problematic functionalities.
- Be careful that the customer perceives the "prototype" as a final software version 0. The prototype should not be time-consuming, it should be quick to build, and no time should be wasted on it in order to "reuse it as final software".

Brainstorming

Brainstorming is a method design teams use to generate ideas to solve clearly defined design problems:

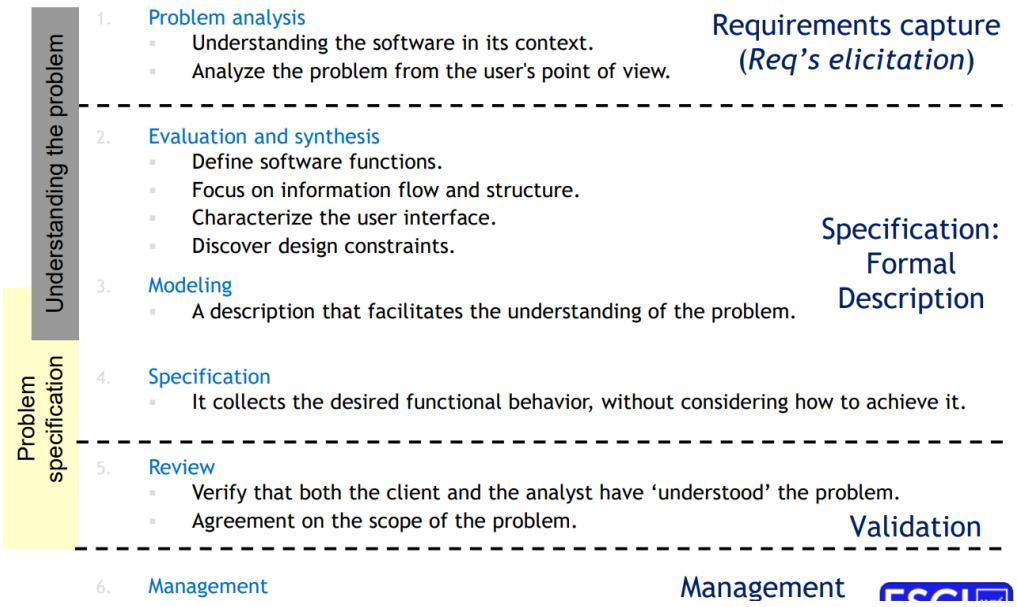
- A wide range of ideas are produced and links are made between them to find feasible solutions.

Aspects to take into account in brainstorming sessions:

- Usable for new products or new functionalities.
- Sessions must be managed and led for them to be profitable.
- At the end of the session, the results should be organized, classified and prioritized so that they can be used.

Tools such as whiteboards or mind-mapping can be used...

Requirements modeling methods



User stories and Scenarios

Scenarios and user stories are real-world examples of how a system can be used.
They are a description of how a system can be used for a particular task.
Since they are based on a practical situation, stakeholders can relate to them and comment on their situation with respect to the story.

SCRUM format of user stories:

The scenario is a formal description structured in the form of a diagram.

Principles of analysis

Each analysis method has its own notation and point of view. However, there are principles that all modeling tools associated with these methods must comply with:

- Partition:
    - A complex system can only be understood if it is structured in inter-related parts.
    - Therefore, the tools we use must be able to describe a problem from its parts.

- Abstraction:
    - We need to be able to define an entity or a problem in general terms. Removing all the details (referencing them separately) to simplify understanding.
    - Consequently, the tools used must allow us to find the details progressively.

- Projection:
    - It is necessary to define the system from different points of view (projections).
    - The combination of points of view ensures a complete understanding of the system, which must be supported by the tools.

The result of the Analysis → The Software Requirement Specification is delivered as a technical document

Requirements specification: Document

Once requirements are clear, they shall be covered by the “Software Requirements Specification” (SRS).
The purpose of the SRS is multiple:

- Contractual: Development will implement this and nothing more.
- Documentation: Working baseline (reference) to be used by the designer and development team.

In some contexts, instead of ‘Software Requirements Specification’ (SRS) one speaks of ‘Software Requirements Document’ (SRD). But both refer to the same content and objective.

Specification properties

The SRS specifies the expected behavior, not the actual implementation.
Therefore, it must satisfy the following properties:

- **Correctness**: The SRS only lists the requirements that shall be implemented.
- **Comprehensible**
- **No ambiguities:** The requirements shall be interpretable in a unique way, two developers shall understand the same:
    - Natural language is ambiguous by default.
    - Formal language is preferred
- **Completeness**: Only if SDR includes:
    - All the requirements related with: functionality, performance, external interfaces, and design restrictions.
    - Defines all the possible answers again all the possible inputs, and in all possible scenarios.
    - Define all the terms and used units.
- **Consistent**: If we cannot find any subset of requirements conflicting with other requirements.
    - Inconsistency in the characteristics of the same element (i.e. format).
    - Logical inconsistency between two processes (i.e. sequence vs. concurrency).
    - Inconsistency in the word and names (i.e. use of synonyms).
- **Ordered**: Group requirements by categories and relevance
- **Verifiable**: Each requirement must be verifiable, there must be a process (finite with an affordable cost) that guarantees the correct implementation of the requirement. Three types of verification:
    - By Test: Software that test software.
    - By Design: The design ensures the implementation.
    - By Code Review: A manual review shall be executed.
- **Modifiable**: The structure and style of the document should allow for uncomplicated changes, so it is important not to be redundant in concepts and to cover all requirements without repetition.
- **Traceable**: In two directions:
    - Backward: The origin of the requirement shall be clear.
    - Forward: Each requirement shall have a unique identification to be referenced in any point in the development

Document format

The format of the "Software Requirements Specification" must be easy to read and well structured. The requirements should be clear, easy to understand, complete, and consistent.

There are several standards that defines how the Specification Document shall be.

- As an example, the ANSI/IEEE organization has established the IEEE 830 standard.
- This standard proposes the following sections:

1. **Introduction**
   1.1 Aim of the product
   1.2 Scope
   1.3 Acronyms
   1.4 References
   1.5 General view / Introduction
2. **General Description**
   2.1 Product perspective
   2.2 Product functions
   2.3 User characteristics
   2.4 General Restrictions
   2.5 Assumptions and dependencies
3. **Requirements specification**
4. **Appendixes**
5. **Index**

- *User Requirements are expressed in natural language.*
- *Technical requirements are expressed in structured language*

And main section of the document should cover:

3. **Requirements specification**
   3.1 Functional requirements
      3.1.1 Functional requirement - 1
         3.1.1.1 Introduction
         3.1.1.2 Inputs
         3.1.1.3 Process
         3.1.1.4 Outputs
      3.1.2 Functional requirement - 1
         ...
   3.2 Non-Functional requirements
      3.2.1 Extern interfaces requirements
         3.2.1.1 Of Users
         3.2.1.2 Hardware
         3.2.1.3 Software
         3.2.1.4 Communications
      3.2.2 Performance requirements
      3.2.3 Design restrictions
         …
      3.2.6 Other NF requirements

Requirement tables
Each requirement should be formatted to cover all basic information.
Typical tables used for this purpose are shown below:

Code : FR-01-001	Users	Verification: D, R
The system shall manage a user access.		
Parent:		

Code : FR-01-002	User information	Verification: D, R
User profile shall contain: •Username : Alphanumeric string from 3 to 8 characters. •Password: Alphanumeric string from 6 to 8 characters. •Email •DNI •Photography : 300x300 pixels in color		
Parent: FR-01-001		

Requirement Validation
Validation is a process of ensuring the specified requirements meet the customer's needs. It's concerned with finding problems with the requirements.

Techniques you can use to validate the requirements:

- Requirements Reviews: interact with the customer to gather requirements, and system developers start reading the requirements in the document and investigate in great detail to check for errors, inconsistency, conflicts, and any ambiguity.
- Prototyping: used when the requirements aren't clear. So, we make a quick design of the system to validate the requirements. If it fails, we then refine it, and check again, until it meets the customer's needs.
- Test-case Generation

The term "tests" here doesn't mean to write and run some code for every function. It means to write a textual description of the "inputs", "expected value", and "steps taken" to perform each function.

Validation: The verification matrix

Another important element related to requirements is the **verification matrix.**

- This matrix links the requirements to the other elements developed during the project.
- The objective is to trace the relationships derived from a requirement to other elements, and it helps to trace requirements that are only defined in the SRS but are not present elsewhere.

The verification matrix can have any needed column:

- **Requirement**: The requirement ID to track
- **Use Case:** The Use case ID (next block in the subject) that defines behavior related with the requirement
- **Test:** ID of the test proposed to verify that requirement
- **Test Report:** Document that shows the results of the test
- **Code:** Module, Source code, that implements things related with the requirement

Example of a verification matrix:

Requirement	Use Case	Test
REQ-F-1-01	UC-01	TP-01

Any requirement with no linked items is a requirement that is not analyzed, not implemented and not tested. So, it is a RED flag!

Conclusions

1. Main objective of the analysis: to understand the scope of the problem and gather System Requirements.
2. User Histories: a useful tool to obtain the Functional Requirements.
3. All requirements must be well classified according to their type.
4. A good SRS document should cover the entire problem domain.

Block 3. Software Modeling (UML I)

In this session we will introduce the ‘UML diagrams’ to perform the analysis and design.

Introduction to diagrams

- The UML as a tool for modeling
- Fundamental characteristics of UML
- UML views
- Types of diagrams in UML

Use Case diagrams

- Syntax of the diagram in UML
- How to use them with SCRUM

Review of the Software Modeling

Software Modeling is the process in which various techniques and principles are applied to define what a device, process or system should look like.
This definition must be done in sufficient detail to allow the effective implementation of it.
Therefore, the Software Modeling is the core of the Software Engineering process and applies regardless of the development paradigm used.

In general, the process consists in:

- Starting point: Problem domain (from user’s description)

- Finish line: Solution domain (from engineers)

We can only reach the goal by making a complete description of the system.

Why do we need to make diagrams?

If the goal of Software Modeling is to produce a representation of the system that can then be used to implement it, then it is necessary to use a common language to model the system.
Many (if not all) aspects can be described graphically.

So currently this common graphical language is UML (Unified Modeling Language).

A proper definition of the system requirements is needed to take advantage of system modeling, because they are the starting point to start building the diagrams.

UML design objectives

Modeling of object-oriented systems, from their initial conceptual stages to executable binaries.

Covering all aspects related to the inherent size of complex and critical systems.

Having a modeling language that can be used by both humans and machines.

Find a balance between expressiveness and simplicity.
The UML language focuses on being useful

UML characteristics

General definition: UML is a language for visualizing, specifying, constructing and documenting, from an object-oriented perspective, the different elements of a system involving a large amount of software.

UML is just notation, not a procedure:

- Different methodologies use UML.
- UML is just a common language for describing the specification.

UML allows specifying all analysis, design and implementation decisions by building accurate, unambiguous and complete models:

- It can be linked to programming languages: integrated IDE, documentation, …
- It allows to easily follow all the stages of a development process (requirements, architecture, testing, versioning, etc.)

As it deals with all areas, it describes many types of diagrams.

UML modeling paradigm

UML emerges from object-oriented modeling, so it is important to properly understand the concepts of Object-Oriented Programing before learning UML.

Basic OOP concepts :

- Objects: Objects represent the basic building blocks of an entity.
- Classes: Classes are blueprints for objects.
- Abstraction: Abstraction represents the behavior of entities in the real world.
- Encapsulation: Encapsulation is the mechanism for binding data together and hiding them from the outside world.
- Inheritance: Inheritance is the creation of new classes from existing mechanisms.
- Polymorphism: A defined mechanism to exist in different forms.

The purpose of object-oriented analysis and design can be described as:

1. Identifying objects in the solution domain.
2. Determining the relationships between objects.
3. Performing a design that can be implemented using an object-oriented language.

UML elements

Basic construction blocks (vocabulary):

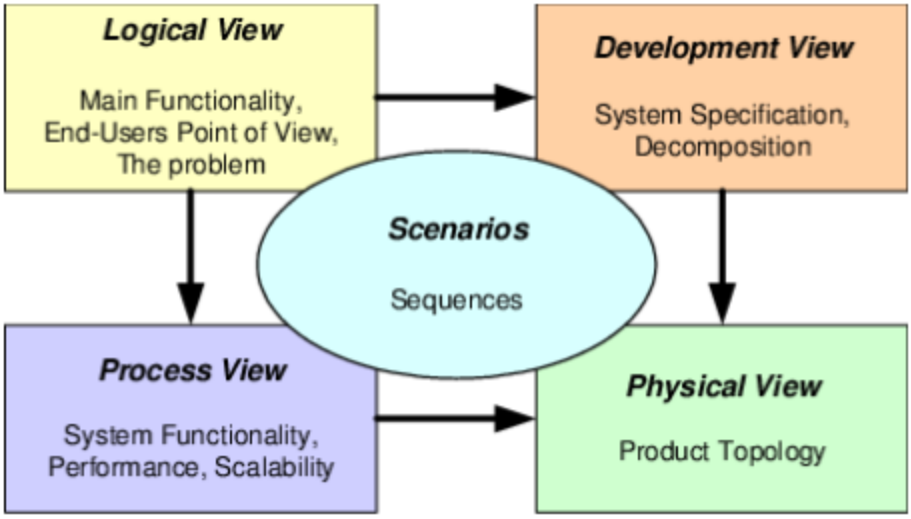
- Views (areas): User, Structural, Behavior, Implementation, Deployment.
- Graphical elements and relationships between them.
- Diagrams: in different types.

Rules (to use the language):

- There are several rules to combine blocs: establish what is a well-formed model.
- Syntactic and semantic rules: names, visibility, integrity, etc.
- Common procedures: notation that allows to fix a style within the model. This notation is valid for the different elements of the different diagrams.

It is necessary to know the different types of diagrams and their syntax. But it is also necessary to know how to interpret diagrams and create them !!

UML views



What are the views for?

- It is very difficult to fully understand a complex system from a single point of view and at the first time.
- Constructing the different diagrams makes us question ourselves about the right questions to ask and make decisions about what and how the solution will be.

Type of view (with respect to the role in the project):

- **User:** system behavior from the user's point of view (answers to what, not how).
    - Stakeholders: users, analysts, designers, testers.

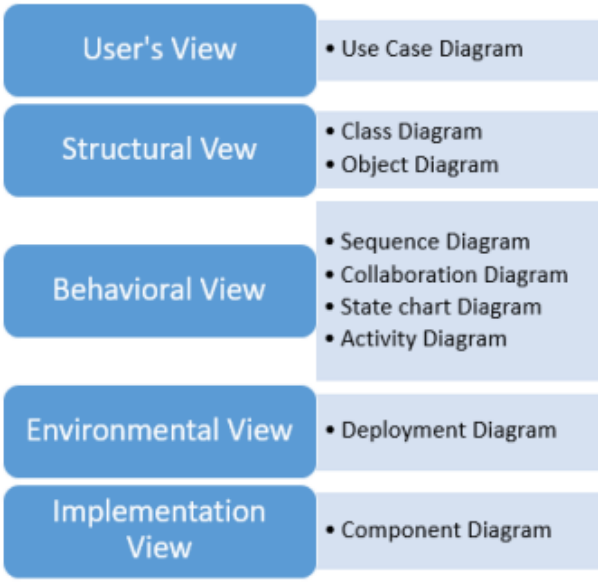
- **Structural:** design of classes to implement functional and non-functional requirements.
    - Stakeholders: designers, programmers, testers.

- **Behavioral:** dynamic aspects of running software: message flow between objects, control flow within a function, state change.
    - Stakeholders: designers, programmers, testers.

- **Implementation:** organization of software with respect to the development environment.
    - Stakeholders: project managers, software quality managers, designers, programmers, testers, others.

- **Deployment:** correspondence between hardware and software.
    - Stakeholders: designers, system engineers.

UML Diagrams



What are the diagrams for?

- Each diagram describes the system from a different point of view.
- Therefore, for each view there are one or more types of diagrams

Diagrams are drawings with various elements and relationships:

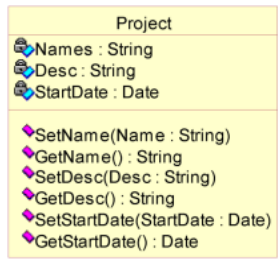
- **Use case:** shows a group of use cases, actors and relationships between them.
- **Classes:** show a group of classes, interfaces and collaborations, as well as the relationships between them.
- **Objects:** show a group of objects and the relationships between them.
- **Sequence:** shows the temporal ordering of messages between objects and actors.
- **Collaboration:** shows the structural ordering of objects sending and receiving messages.
- **States:** shows a state machine.
- **Activities:** subtype of state diagram showing control flows.
- **Components:** show the organization and dependencies between a group of components.
- **Deployment:** shows the configuration of processing nodes.

It's just the beginning, there are more types!

UML elements - Structural elements

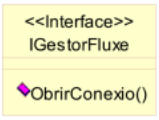
Static parts: Represent concepts or materials

Class:
functional Entity in OOP



Class is a collection of objects with the same properties, methods, relationships and semantics.

Interface:
abstract Entity in OOP



Interface refers to the services (operations) provided by a class or component, and describes the actions of the class or component that are visible to the outside world.

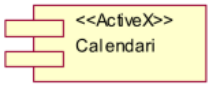
Collaboration:

defines a set of participants and relationships that are meaningful for a given set of purposes

Use Case:
group of action sequences executed by a system

A use case defines the interaction between actors (people external to the system and interacting with the system) and the system under consideration to achieve a business objective; enables a visual overview of system requirements.

Component:
physical and removable part of a system



Node:
physical elements that exist in the system at runtime

UML elements - Behavior elements

Dynamic parts: Represent behavior in time and space

- Interactions: Messages shared between objects to achieve a given objective

  Interaction is defined as a behavior that includes the exchange of messages between a set of elements to accomplish a specific task

- State: Stages an object goes through as an answer to the different events.

  State consists of a series of states of an object, and it is useful that the state of an object during its lifetime is important.

UML elements - Generic elements

Aggregation elements: Package

- Think of aggregation elements as a "box" in which models can be decomposed. Package is purely conceptual and only exist during the development phase, whereas components exist at runtime

Annotation elements: Note

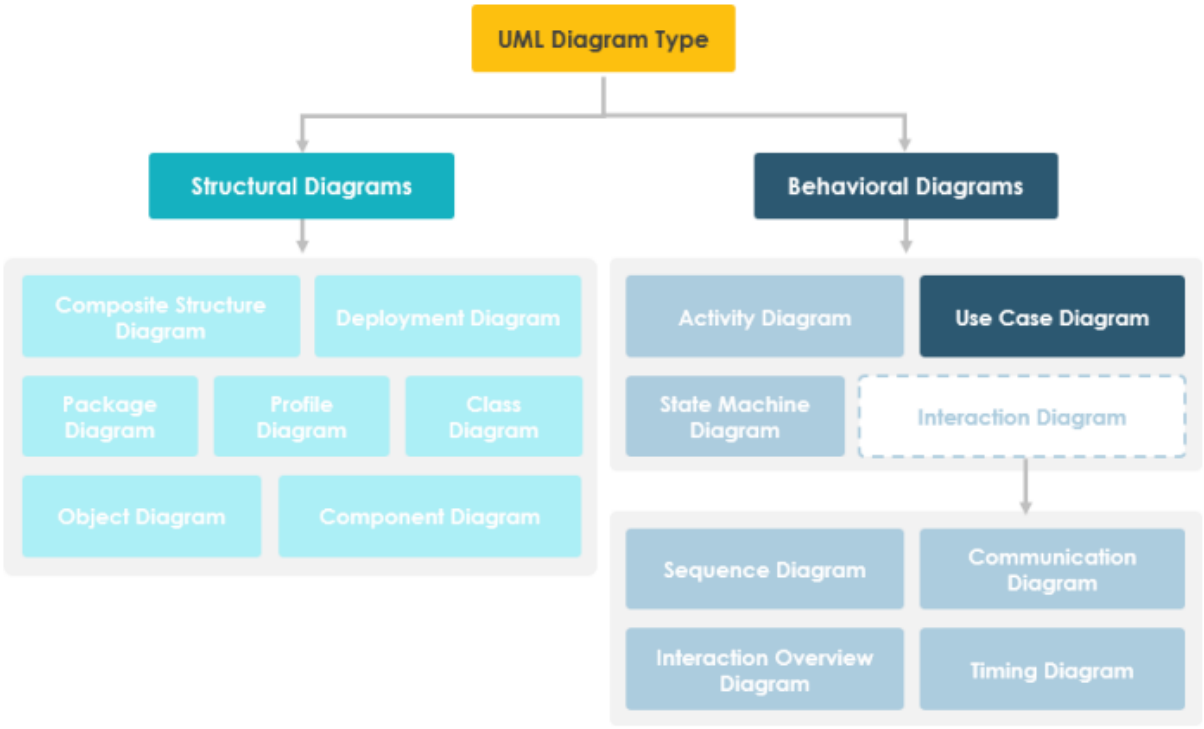
- Annotation elements can be defined as a mechanism to capture the utterances, descriptions and annotations of UML model elements. Annotation is the only annotation element

UML elements - Relationships

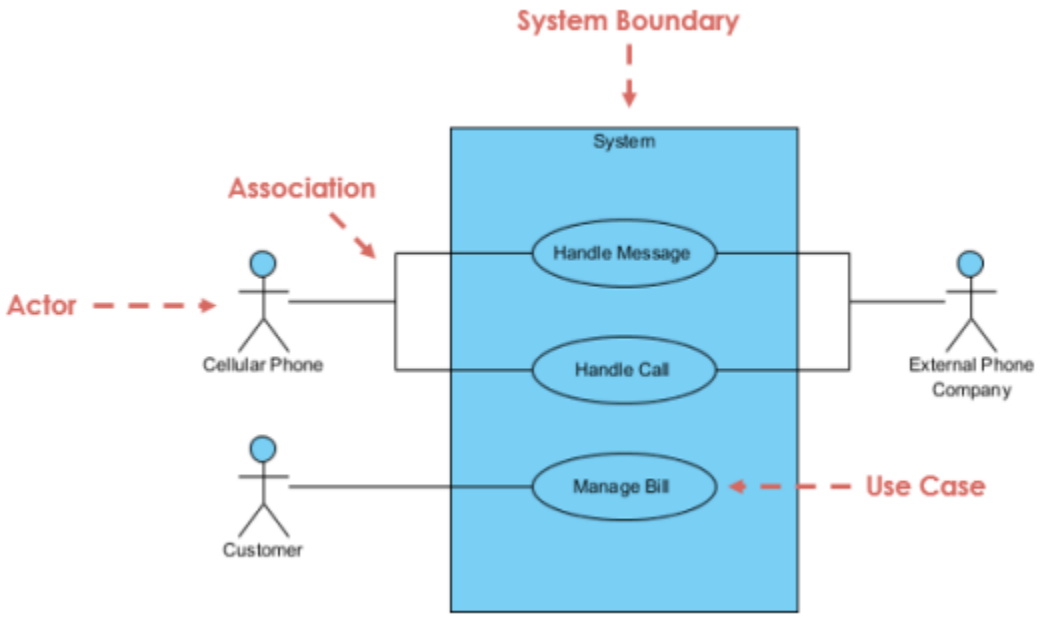
The relationship between elements shows how elements are related to each other. There are four types of relationships defined in UML:

- **Semantic**: a modification of the independent element affects the dependent one -
- **Structural**: connection between elements
- **Generalization/Specialization**: the heritance between classes
- **Realization**: semantic relationship between classes agree on contracts where one class contract is guaranteed to be enforced by the other class.

Graphical view of UML diagram types



Use Case diagrams



A use case specifies the desired behavior of the system.
They represent the functional requirements of the system.
They describe what the system does, not how it does it.
Objectives of the Use Case diagrams:

- Requirements specification: The idea is to specify a system based on its interaction with the environment.
- It promotes the description of the system from the point of view of who will use it, and not from the point of view of who must build it.
- A use case diagram is the description of a series of interactions between the system and one or more actors. Here the system is considered as a black box.

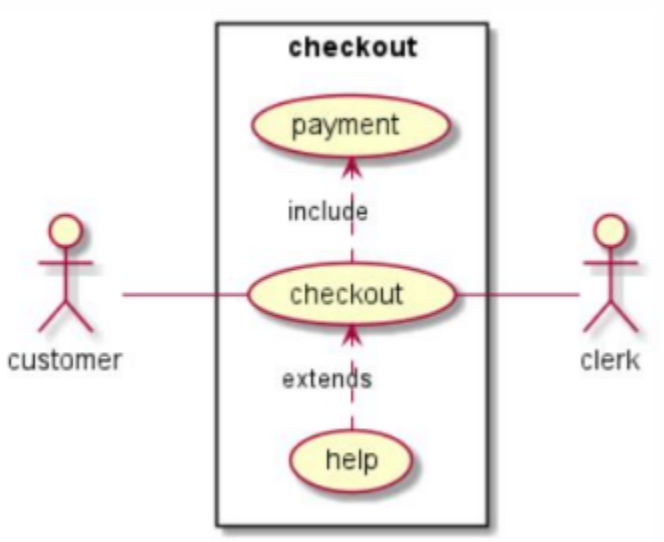
The key is to identify the actors first!

Actors

An actor represents a coherent set of roles played by the users of the use cases when interacting with the system:

- A user can play different roles.
- Different actors can be involved in the realization of a use case.
- An actor can be involved in several use cases.
- Identify use cases through external actors and events.
- An actor needs the use case and/or participates in it.
- The actor could be a human or an external system.

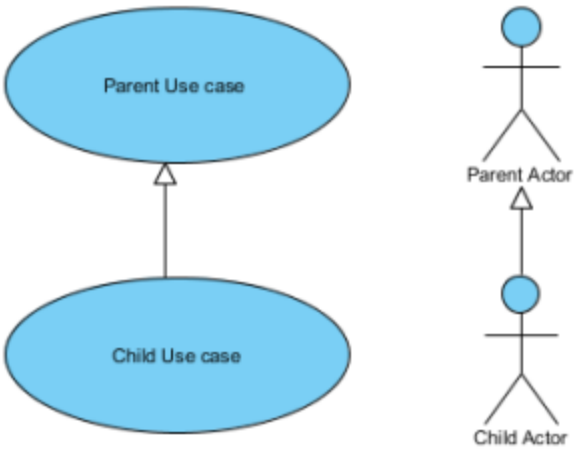
Associations - Relationships



Actors are generally related to one or more use cases.
Use cases are related to one or more actors.
The relationships between use cases can only be of type: extension, inclusion or generalization:

- **Extension**: adds (optionally) new behavior to the related base case.
- **Inclusion**: without it (required) the related base case is incomplete.

Generalization/Specification



Behavioral inheritance.
It can exist both between actors and between use cases.
It only makes sense when the parent and child have a specific differentiated behavior.

Modeling with Use Case diagrams

What are use case diagrams for?

- They offer a systematic and intuitive way to capture functional requirements, focusing on the value added by the user.
- They guide the entire development process since most of the activities (planning, analysis, design, validation, testing, etc.) are performed from the use cases.
- Important mechanism to support "traceability" between models.

Modeling Path: Functional Requirements → User Stories → Use Cases.

- Although it is the methodology that defines the path, we always begin by building the use case diagram from the functional requirements.
- And to achieve this goal, it is necessary to identify the actors involved.

Usage diagrams must be complete to be valid.

Use Case diagrams with SCRUM

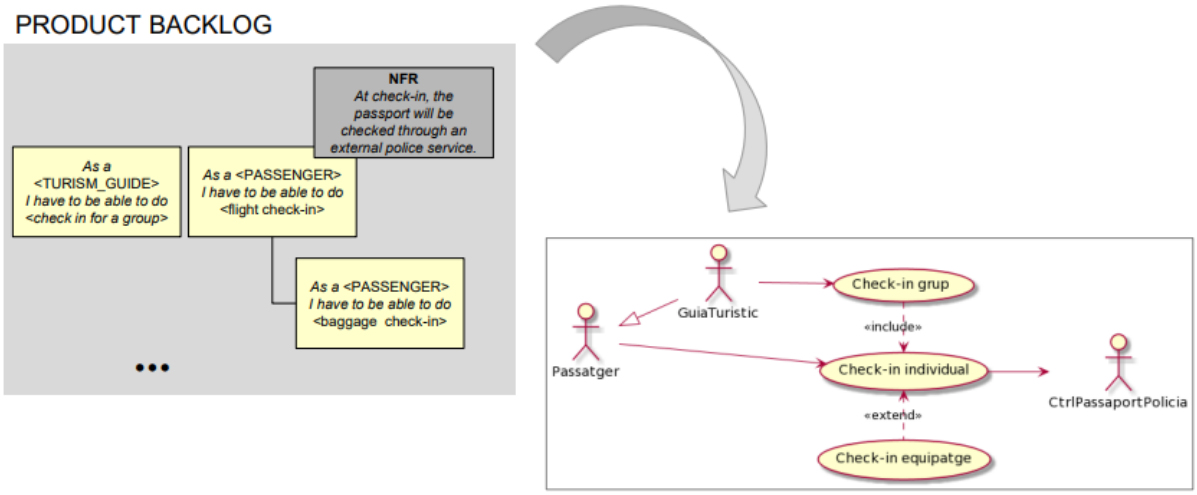
With the SCRUM methodology the Use Cases and User Stories are relevant because:

- A User Story in SCRUM is synonymous with a feature of the software to be built, detailed enough to fit into an iteration (Sprint periods).
- A Use Case provides a contextual view of the software to be built.
- So the Use Case technique helps to identify all alternative scenarios in the use case realizations.
- Use Cases can be used to understand where new stories are likely to be found.
- A Use Case is thus a functional unit of the software, and different scenarios (main and alternative flows) can generate different stories that are resolved in different sprints.

User Stories in SCRUM are related to the Product Backlog

SCRUM: Product Backlog and Use Cases

In SCRUM there is a direct relationship between the Product Backlog and the Use Case Diagram:



It's necessary to be systematic in translating from F.R. to U.C.

Conclusions

1. UML is a standard way to visualize the design of a system.
2. Different views of the solution require different type of diagrams.
3. The Use Case diagrams are part of the Behavioral View.
4. A Use Case diagram consists of actors, use cases, system boundary, and relationships.