Notes Software Engineering

Brendon Mendicino

May 10, 2023

Contents

1 Introduction

Definitions:

Definition 1.1 – Multi Person Multi Version

People coordinating in long period of time.

Definition 1.2 – Software

Is a collection of code and not only digital assets like: rules, documentations, procedures and many more.

Definition 1.3 – Software Types

- **Stand alone**: products used alone, like email, office, calendar;
- **Embedded in software products**: car, smart house;
- **Embedded in business process**: an information system;
- **Embedded in production process**: embedded in factories and classic production pipelines;

1.1 Describe Software

To describe a software we define his properties, they are divided in: **functional** property, which express an action to be performed, and **non-functional** property, that describes how a functional property should behave and describing his correctness. Because it's not possible to define if something is 100% correct in the software field the **reliability** is also defined, the idea is that correctness it's impossible to obtain, thus trying to the number of **defects** a software can have, setting a threshold for the maximum number of them, on the other hand the **availability** is the percentage, over a period of time, of the system without occurring in any defect: $A = \frac{T - T_{\text{down}}}{T}$. Other non-functional properties are **security, safety and deniability**. **Efficiency** is the response time and the amount of resources used.

Every software has a life process divided in: development, operation, maintenance. During development, which will be the main focus of this course, there are 4 main phases:

- **requirements**;
- **design**;
- **coding**;
- **testing**;

Basic rules of software development:

- **keep it simple**;
- **separation of concerns**;
- **abstraction**

1.2 Activity

In software the base is source code, to make it more readable the code is divided in **units**, before starting to code there should be an idea of the **design** of the project, which is how and which are the units are interacting with each other. Only after the requirements and the design the coding starts. The start of the process is deciding what the software should do, those are all the **requirements**. Every part produces a result:

- **Requirements → requirements document**
- **Design → design document**
- **Implementation → unit**

After the requirements there should be some checking with **Validation and Verification** (VV).

1.3 Phases

After the first version of the product is released there is a deployment phase where people will install the product, the product will need **operation** and **maintenance** operations and after a period of time the product will **retire** meaning that it will no longer be supported. During the support span the operations are all the set of the actions done on the product, like security tasting, bug discovery, etc. While maintenance is the action of changing the code to do bug fixes or publish new feature.

2 Git

Git uses distributed CMS (Content Management System) to provide version control of a project, using the concept of snapshots which allows developers to work concurrently on a single project, other than that git also has integrity with a checksum and instead of storing whole files it just stores the **delta** of the changed file (a delta represent the changes in a file based on base file). Basic git concepts:

- **repository**: contains all the files and versions of the project;
- **working copy**: it is a snapshot of the repository, the working copy is on the client side;
- **commit**: it is an atomic operator that modifies the repository, all commits are tracked into a log file, to every commit a message is associated;
- **push**: is an operation that updates the modifications from a local server to the online server;
- **update**: updates the working copy by merging the changes;
- **staging area**: it is local dock that stores the changes that are not committed yet;
- **typical workflow**: checkout project, stage changes, commit;

Git basic commands:

- `$ git init`: initialize a local repo;
- `$ git remote add origin http://server.com/project.git`: add a new remote repository;
- `$ git status`: status;
- `$ git add`: add a file;
- `$ git diff`: changes;
- `$ git commit -m "..."`: commit changes;
- `$ git commit -am "..."`: commit all changes;
- `$ git rm`: removes a file with git tracking, adding the staging area;
- `$ git mv`: moves a file adding modifications to staging area;
- `$ git log`: logs;

- `$ git reflog`
- `$ git pull/push`
- `$ git checkout <branch-name>`: switch between branches;

Every commit points to a tree which contains a list of modified files, it's possible to reach the **blob** of the changes made to that file via a pointer, every commit is linked to the previous one. The last commit of the current working branch is called **HEAD**, every HEAD points to a branch that we last commit to. If we want to switch branch we need to check out the branch, and the HEAD will change which branch it is pointing to. If there are many branches we want to bring the changes of a branch to our current one, we need to use `git checkout other-branch`, when merging two branches some **conflict** can be created, when this happens the changes need to solved by hand, in other case git is able to fix them on his own.

Other than merge there is also **rebase**, which is a bit different that merge, it tries to rewrite the history of the branch we are rebasing to and to commit all our changes to the last commit. The difference between merge and rebase, is the rebase it creates a linear commit history, while merge is keeps track of every commit where done in different branches.

- `$ git rebase -i HEAD~4`:
  - -i $\implies$ interactive mode
  - ~ 4 $\implies$ number of commits we want to target;
- `$ git merge <other-branch>`: this command tries to merge `other-branch` inside the current working branch.

Example 2.1 – Revert cahnges

In case an uncommited file needs to be reset to the original version it is possible to use two commands:

- `$ git checkout <name-of-file>`
- `$ git reset --hard`

Example 2.2 – Resolve a conflict

When merging two branches conflicts can arise, if git is not able to automatically solve it will prompt the user to do so. When a conflict needs to be solved manually a new version of the file is created and inside both versions of the branches are present (highlighted) with conflicting sections, the conflict is solved be hand and then by committing the file.

Fork and pull is the way to go for open source development, the operation of fork copies the full project in your account and thus becoming the new owner, the two projects are separate unless it is synched to the original one.

Definition 2.1 – Conventional commit message

```
<type>[optional scope]: <description>

[optional body]

[optional footer(s)]
```

Type of commit:

- **fix**: a commit the type `fix` patches a bug in your codebase.
- **feat**: a commit of type `feat` introduces a new feature to the codebase.
- **BREAKING CHANGE**: a commit with `BREAKING CHANGE` or appends `!`, introduces a breaking API change

3 CI/CD

We start first by using Docker, Docker is a software that allows running different applications in a virtualized environment, this allows having specific versions of the required softwares without accurring in any conflict between the various applications. Docker shared the current kernel with all the running containers, Docker is composes of the three main components:

- Docker CLI, which manages the images, the containers, the volumes, networking.
- REST API, which links the Docker CLI with the Docker Engine.
- Docker Engine, which manages the interface with the kernel.

Docker has two big arguments: **Docker Images** (is a template of an applicatoin), **Docker Container** (the real application based on the respective image).

The image is composed of many layers that are read-only (called **union file system**), when creating a container and some data are changed docker use a *copy-on-write* policy. In this way when having multiple container the common unchanged parts are shared while the differences are contained only within that container. To keep the data persistent there are three ways that docker exposes:

- **Volumes**: specifing a directory in the pc
- **Bind Mount**: docker will handle the path where the data wil be mounted
- **Tempfs muonts**: temporary directory

To create an image we need to use **docker file**

```
# base image
FROM node:14
# working directory inside the container
WORKDIR /app
# copy from the current path to the container
COPY package*.json
# run commands after install
RUN npm install
# copy the remaining files from host to the container
COPY . .
# cmd to start when running the container
CMD npm install mongoose; npx nodemon server.js
```

and then to create a new container

```
$ docker build -t my-app-image
$ docker run -d -p 3000:3000 -v /path/to/folder:/app my-app-image
```

when having more docker images that need to run concurrently we use **docker compose**, that uses `docker-compost.yml` as configuration file, that allows us to define environmental variables, port forwarding, dependencies, volumes, the containers, ...

Continuos Integration is a software methodology for high quality software, this is done by running automated test and highlighting any error.

Continuos Delivery/Deployment when all the test are passed the software is automatically builded and deployed.

To implement CI with *gitlab* we need to create a `.gitlab-ci.yml`, by default the pipeline is triggered by `git push`.

4 Testing

Testing can be performed in two ways: *static* (by doing code review), *dynamic* (unit testing). The purpose of the test is trying to find defects in the code, while on the other hand debugging should be performed when a fault arises. When a test is failing then we can start debugging.

A **test suit** is a set of unit test related to a specific context. Usally who writes the test should not be same person who writes the code, this is because there could be some bais in the way the code is written, thus not catching some defects, the tema should be dived in **tester team** and **developer team**, or even have a **3rd party tester**. The ideal condition is to have an **Oracle**. When creating tests, we would like **exhaustive test** but this in reality is impossible, due to the high amounts of test that would need to be performed.

Definition 4.1 – Dijkstra thesis

"Testing can only reveal the precence of error, and never their abcence"

4.1 Test classification

- **Per unit**: unit, integration, system, regression
- **Per approach**: black box (functional) ,white box (structural), reliability assessment/prediction, risk based
- **Per formality**: informal, formal

Unit test: individual modules, Integration: some modules work toghether, System test: all modules work togheter.

When using the approaches, the test that can be done are:

- black box: unit, integration, System
- white box: unit
- reliability, risk: system

The **coverage** is how much code we are coveringwith tests, and what kind of requirements we are covering, we are describeing what a test is calling. When doing utit test the converage can be defined for:

- **methods**: methods of a class
- **partitions**: defines a partition
- **boundry**: when we have some thresholds in the partitions we can test them

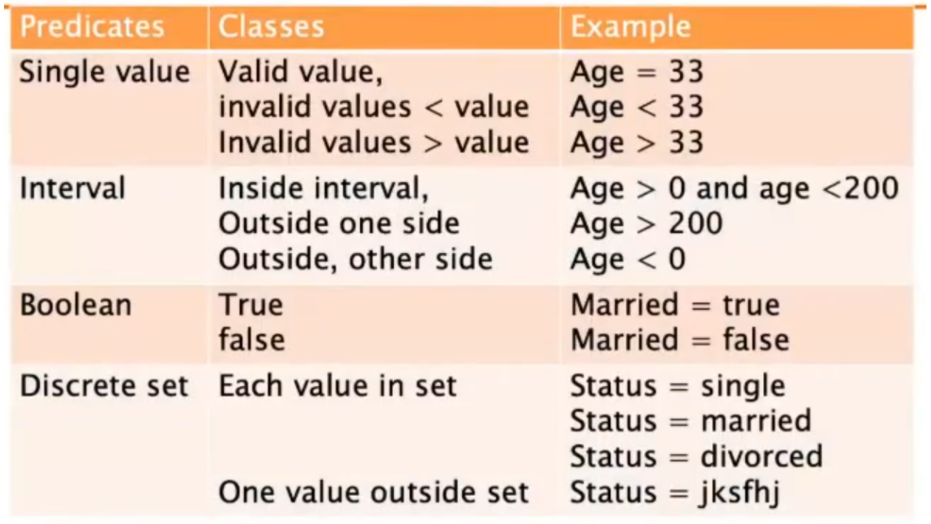
When testing a **system** we can test:

- function requirements
- scenarios
- non-functional requiremnts

Black box means to choose some random input.

Equivalent class partitioning, highlight where the difference are and then test those differences, if a test fails in class then it will fail also in similar classes. An equivalence class is defined by: **creiteria**, **predicates**, **partitions**:

```
Criterion: age
Predicate: age < 100, age > 25
```



Predicates	Classes	Example
Single value	Valid value, invalid values < value Invalid values > value	Age = 33 Age < 33 Age > 33
Interval	Inside interval, Outside one side Outside, other side	Age > 0 and age <200 Age > 200 Age < 0

Boolean	True
	false
		Married = true
		Married = false

Discrete set	Each value in set
	
	One value outside set
		Status = single
		Status = married
		Status = divorced
		Status = jksfhj

Figure 1: Equivalence Class

When building black box testing we need to write a table with all the combinations of the criteria (also boundry need to be included).

In javascript it's possible to create a test by writing

```javascript
import app from 'app';

describe('test suite 1', () => {
  test('T1: convert "123" -> 123', async() => {
    const res = app.convert('123');
    expect(res).toBe(123);
  });

  test('T2: convert "1d3" -> error', async() => {
    expect(() => app.convert('1d3')).toThrow(Error);
  });
});
```

4.2 Lab

4.2.1 Exercise 1

```
boolean acceptableToEat(int carb, int protein, int fat);

The function acceptableToEat receives the weight in grams of, respectively,
    carbohydrates, proteins, fats in a serving of food. It returns true if
- the total amount of calories is < 1000
- (carb + protein) / fat > 1/2
```

Da rivedere...
Criteria:

Criterion Id	description
fat	sign of fat
carb	sign of carb
protein	sign of protein
calories	total calories
proportion	proportion

Predicates:

CId	Predicate
fat	positive, negative
carb	positive, negative
protein	positive, negative
calories	true, false
proportion	true, false

Boundaries:

fat	== 0
	== -1
carb	== 0
	== -1
protein	== 0
	== -1

Equivalence classes and tests:

carb	protein	fat	calories	proportions	result	test
pos	pos	pos	pos	true	true	`T1(1,1,1):true`
pos	pos	pos	pos	true	false	`T2(1,1,10):false`
pos	pos	pos	pos	false	true	`T3(100,100,100):false`
pos	pos	neg	/	/	/	`T4(1,1,-1):error`
...	...	...	/	/	/	`T5(1,1,-1):error`
neg	neg	neg	/	/	/	`T6(-1,-1,-1):error`

4.2.2 Exercise 2

```
double computeFee (int duration, int minRate, int minRate2);

This function computes (in euros) the fee for a bicycle rental, using these
    parameters
    * duration: minutes the bicycle has been used
    * minRate: cost per minute, in cents of euro
    * minRate2: cost per minute, in cents of euro
The fee is computed as follows: free the first 30 minutes. minRate per min for the
     first hour exceeding the first 30 min (30 to 90 minutes), minRate2 after 90
    minutes
```

Criteria:

Criterion Id	description
duration	bicycle rental minutes

Predicates:

CId	Predicate
duration	$0 < \text{duration} < 30$
	$30 < \text{duration} < 90$
	$\text{duration} > 90$
	$\text{duration} < 0$

Boundaries:

Criterion	Boundary
duration	duration == 0
	duration == 90

Equivalence classes and tests:

duration	Valid	Test case

$\in [0, 30)$	y	`T1(15, 30, 30):0.0`
		`T2B(0, 30, 30):0.0`

$\in (-\infty, 0)$	n	`T3(-10, 20, 30):error`
$\in [30, 90)$	y	`T4(40, 20, 30):...`
$\in (90, \infty)$	y	`T4(100, 20, 30):...`

4.2.3 Exercise 3

```
double computeFee(double basePrice, int n_passengers, int n_over18, int n_under15)
    ;

A railway company offers the possibility to people under 15 to travel free. The
    offer is dedicated to groups from 2 to 5 people travelling together.

```

```
For being eligible to the offer, at least a member of the group must be at least
    18 years old. If this condition applies, all the under 15 members of the group
     travel free, and the others pay the Base Price.

The function computeFee receives as parameters basePrice (the price of the ticket)
    , n_passengers (the number of passengers of the group), n_over18 (the number
    of passengers at least 18 old), n_under15 (the number of passengers under 15
    years old). It gives as output the amount that the whole group has to spend.
    It gives an error if groups are composed of more than 5 persons.
```

Criteria:

Criterion Id	description
C1	basePrice
C2	npassengers
C3	over18
C4	under18

Predicates:

CId	Predicate
C1	basePrice > 0
C2	$2 <$ npassengers < 5
	$0 <$ npassengers < 2
	npassengers > 5
C3	over18 > 0
C4	under18 > 0

Boundaries:

Criterion	Boundary
basePrice	basePrice $== 0$
npassengers	npassengers $== 2$
over18	over18 $== 0$
under18	under18 $== 0$

Equivalence classes and tests:

basePrice	npassengers	over18	under18	Valid	Test case
	> 5				T1
> 0	> 0	> 0	> 0	x	T2B
> 0	> 0	$== 0$	> 0	x	T3

4.3 Statement Coverage

It's a test where we try to cover every statemnt branch, every statement terminates with an action.

Sometimes it's better to transition the code to the to a **control flow graph**, with this approach we avoid the specific language dependency.

4.4 Decision Coverage

...

4.5 Edge Coverage

When we have control flow graph we cover every edge of the graph.

4.6 Single Condition Coverage

When we have more than one condition in a statement, decision coverage is not able to provide us with enough information

```
bool is_married;
bool is_retierd;
int age;
if (age > 60 && is_retired || is_married)
  discount_rate = 30;
else
  discount_rate = 10;
```

we need the one combination per each test that allows us to execute it, like putting all condition to true or all to false, by doing that we also *achieve decision coverage*, if we use a different combination with mixed true and false we *don't achieve decision coverage*.

4.7 Multiple Condition Coverage

Multiple Condition Coverage is the same as the single but, all combination of condition is tested.

4.8 Path Coverage

In practice path coverage is very simple, if we insert some loop then the number of paths increases with an exponential curve, in that case we only cover the first iteration, this is called **path-n**. **Loop coverage** is when: we write three test cases where:

- we don't enter the loop
- there is only one loop
- there are n-loop

5 Requirement Engineering

The aim of requirement engineering is to define the *properties of the product* before starting implementation. When defining the functionality there is one or more non-functional property associated. Requirement engineering is divided into:

- **Elicitation**: talk to the end-user and understand what they want
- **Analysis and formalization**: writing down the
- **Inspection**: checking what it has been written down on the documents

Definition 5.1 – Stakeholder

In elicitation the person in charge of the requirements has to extract all information to create the requirements, the stakeholder is the *person or company* that is involved in the building of the project. The stakeholder may be: user, administrator, buyer, analyst, developer.

The starting point is an informal description of the problem, usually they contain defects like: omission, inconsistencies, ambiguity. The best document is the most concise one and doesn't contain omissions, this is called **complete and consistent**.

Example 5.1 – Stackeholder

- POS in a supermarket.
- User:
  - cashier at POS (profile 1)
  - administrator, inspector (profile 2)
  - customer at POS
- Administrator:
  - POS application administrator (profile 3)
  - IT administrator (profile 4)
  - ...
- Buyer:
  - CEO of supermarket

5.1 Context Diagram

The context diagram tells what is the focus of the requirements. The context diagram contains the **actors** of the system which can interact with it, the system is called **use case** and there can be more than one. The context diagram defines the **interface** between the inside and outside.

Example 5.2 – EZGas

"EZGas is an application to help drivers find gas at lowest prices. Gas station owners can register their gas station with prices and eventually discounts. Users look for gas stations closest to them and with best prices and quality of service."

1. **Stakeholder Users**:
   - Person looking for a gas station;
   - Owner of the gas station;
   - Workers of the gas station;

   Administrator:
   - App database(s) administrator;
   - Local gas station administrator;
   - Money transaction administrator;

   Buyers:
   - Local gas station owners;

2. **Context diagram and interface**

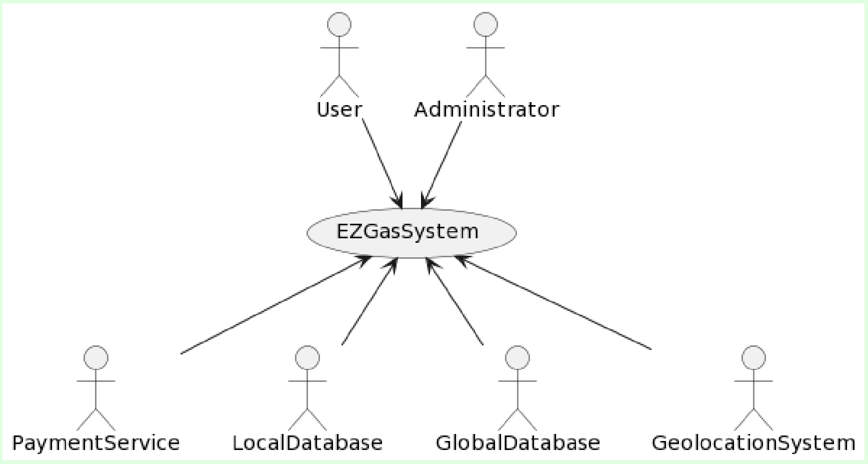


Figure 2: Context Diagram EZGas

Actor	Physical Interface	Logical Interface
User	Smartphone, Internet	GUI
Administrator	Screen, Keyboard	GUI, shell
Payment System	Internet	API
Geolocation System	Internet	API
Local/Global Database	Keyboard, Internet	shell, API

Table 1: Interface Table

3. Functional Requirements

f1	user authentication
f1.1	use authentication token
f1.2	register unregistered users
f1.3	create new account
f2	search gas station
f2.1	choose search options
f2.1.1	closest gas station
f2.1.2	best price gas station
f2.1.3	closest/best price mix gas station
f2.2	select fuel kind
f2.3	select most green company
f3	user becoming gas station owner
f3.1	add owned gas station
f3.1.1	add discount
f3.1.2	remove discount
f3.1.3	modify gas prices
f3.1.4	modify gas kinds
f3.2	pay periodical fee

5. Table of rights

Actor	User	Owner
f1	✓	✓
f2	✓	✓
f3		✓

Example 5.3 – Robotic Vaccum Cleaner

"Since several years robotic vacuum cleaners (RVC) are available. An RVC is capable of cleaning the floors of a house in autonomous mode. An RVC system is composed of the robot itself and a charging station. The charging station is connected to an electric socket in the house, and allows charging the battery on board of the robot. The robot itself is composed of mechanical and electric parts, a computer, and sensors. One infrared sensor in the frontal part recognizes obstacles, another infrared sensor always on the frontal part recognizes gaps (like a downhill staircase). A sensor on the battery reads the charge of the battery. The computer collects data from the sensors and controls the movement of four wheels. Another sensor on one of the wheels computes direction and distance travelled by the robot. Finally, on top of the robot there are three switches: on-off, start, learn. The learn button starts a procedure that allows the robot to map the space in the house. With a certain algorithm the robot moves in all directions, until it finds obstacles or gaps, and builds an internal map of this space. By definition the robot cannot move beyond obstacles, like walls or closed doors, and beyond gaps taller than 1 cm. The starting point of the learn procedure must be the charging station. When the map is built the robot returns to the charging station and stops. The start button starts a cleaning procedure. The robot, starting from the charging station, covers and cleans all the space in the house, as mapped in the 'learn' procedure. In all cases when the charge of the battery is below a certain threshold, the robot returns to the charging station. When recharged, the robot completes the mission, then returns to the charging station and stops."

Business Model:

- Private Company;
- Customers buy the RVC;
- Customers can bring RVC to assistance;
- Expert Customers can buy spare parts;

1. Stakeholders:

- Users:
  - Customers;
  - Technicians;
- Administrators:

    - Firmware administrator;
    - Components administrator;

- **Buyers:**
    - Customer;
    - Components Companies;

2. **Context Diagram:**

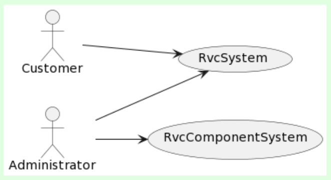


Figure 3: RVC Context Diagram

Actor	**Physical Interface**	**Logical Interface**
Customer	Buttons	-
Administrator	Keyboard	USB

3. **Functional Properties:**

f1	perform action
f1.1	start/stop learning
f1.2	start/stop cleaning
f1.3	start/stop charging
f1.4	return to charging station
f1.5	return to previous action
f2	firmware update
f3	factory reset
f4	read data from sensors

5. **Table of Rights**

6 UML

Thanks to UML (*Unified Modeling Language*) it's possible to represent the relationships between the various objects of the system. UML has many characteristics, some of which are (it the case of *UML Class Diagrams*):

- The Object is the **Model** of an item, every Object *has unique ID*.
- **Class** is the descriptor of a set of items that can be grouped together.
- Classes can be **associated**.
- Every Class has **Attributes**.

In UML different class can be *linked* together, this is called **Association** and represent a logical relationship between them, the association can unidirectional or bidirectional, also classes may have more than one link to represent different logical concepts. When links are formed between classes there can multiplicity of the number of classes dependent:

- `1`
- `0..1`
- `0..n`
- `1..n`
- `m..n`

The **role** is how what a class represent when it has an association. This comes useful when **recursive associations** are present.

Composition is indicated with a *diamond*, it represent when a Class is a *component* of another one, this takes a different approach with respect to inheritance, and there are two kinds:

- When the diamond **is hollow** and *the class A which points to B* means that, *A* is part of *B*.
- When the diamond **is full** the lifecycle are all associated, if a component dies all the other component die too.

Generalization is a way to implement inheritance, the parent class is pointed by a *hollow arrow*. The parent class is a more generalized class, all the attributes of the generalized class are inherited by his children.

Before starting the project with UML it is import to define what are his context, e.g.:

- model of concepts (glossary);
- model of system (system design);

- model of software classes;
- model of deployment (deployment diagram);

It's always important to minimize the number of classes.

6.1 Glossary

The **glossary** or **conceptual diagram** is a way to describe the objects of our system processing. Thanks to *UML Class Diagram* it's possible to represent the relationships between the various objects of the system. The glossary it's different from the **System Design**, in fact during this phase the design of the various software component must not be considered, for example the glossary representation of a university is the following:

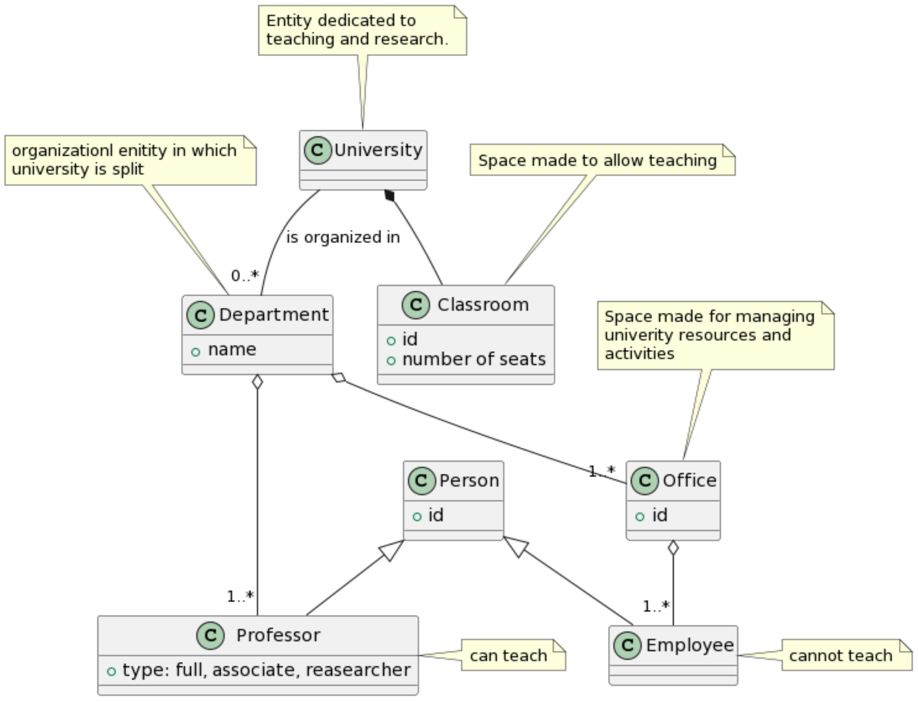

Entity dedicated to teaching and research.
organizationl enitity in which university is split
University
Space made to allow teaching
is organized in
0..*
Department
name
Classroom
id
number of seats
Space made for managing univerity resources and activities
Person
id
1..*
Office
id
1..*
1..*
Professor
type: full, associate, reasearcher
can teach
Employee
cannot teach

Figure 4: University UML Diagram

If the name of the class is not self-explanatory a note is recommended to explain what that is or does.

6.2 System Design

On the other hand a **System Design** shows how the software parts of the system are related to one another, the system design can have different granularity, ranging from the representation of an entire software app reduced to a Class, to represent the real software Classes inside the application.

6.3 UML Deployment Diagram

The **deployment diagram** is a way to represent how the system is deployed on the physical hardware, and it uses *UML Deployment Diagram*, in general every physical device is represented with a **node**, and the piece of software, library, or file that is *deployed* on that physical device is called **artifact**.

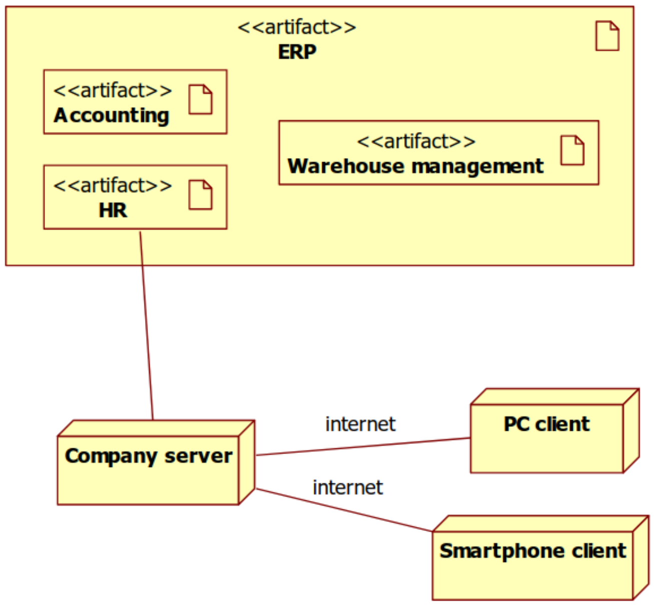


Figure 5: Deployment Diagram Example

Example 6.1 – Lab2 EZGas

6. Glossary

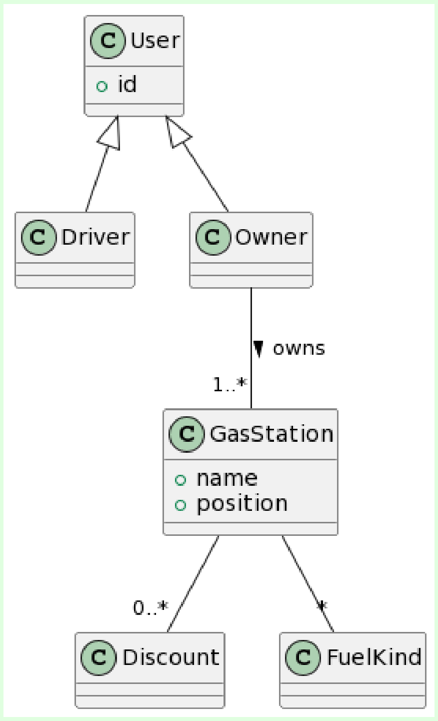


Figure 6: Ezgas Glossary

7. System Design

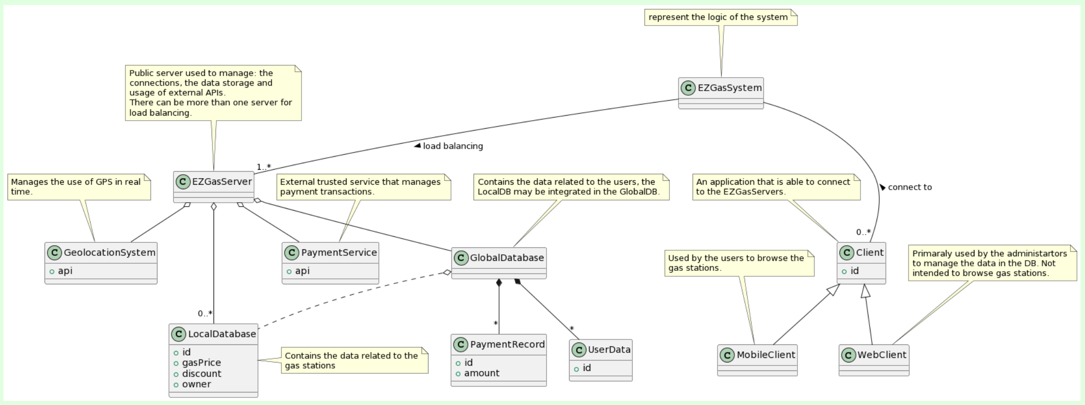

represent the logic of the system
EZGasSystem
Public server used to manage: the connections, the data storage and usage of external APIs.
There can be more than one server for load balancing.
load balancing
1..*
EZGasServer
connect to
0..*
Manages the use of GPS in real time.
External trusted service that manages payment transactions.
Contains the data related to the users, the LocalDB may be integrated in the GlobalDB.
An application that is able to connect to the EZGasServers.
GeolocationSystem
api
PaymentService
api
GlobalDatabase
Client
id
Used by the users to browse the gas stations.
Primaraly used by the administartors to manage the data in the DB. Not intended to browse gas stations.
0..*
*
*
LocalDatabase
id
gasPrice
discount
owner
Contains the data related to the gas stations
PaymentRecord
id
amount
UserData
id
MobileClient
WebClient

Figure 7: Ezgas System Design

8. Deployment Diagram

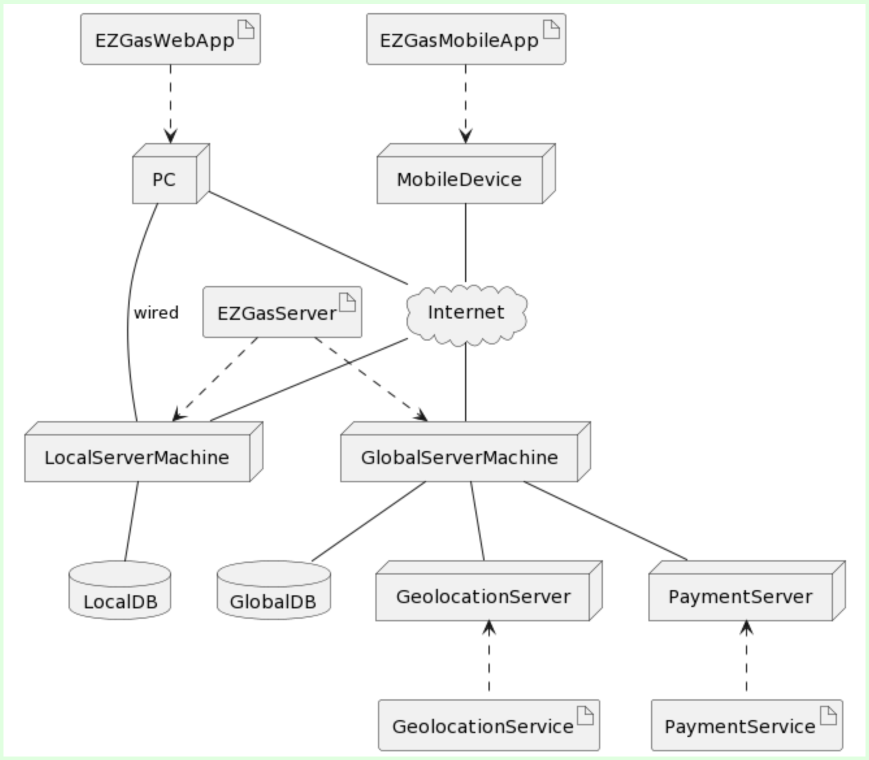


Figure 8: Ezgas Deployment Diagram

Example 6.2 – Lab2 RVC

6. Glossary

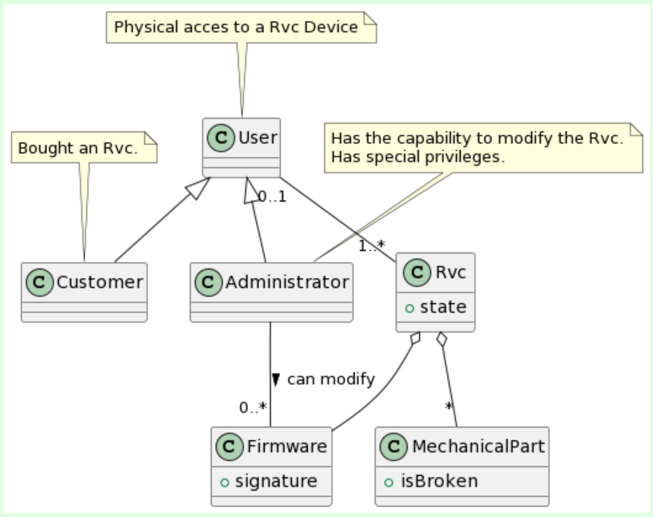


Figure 9: Rvc Glossary

7. System Diagram

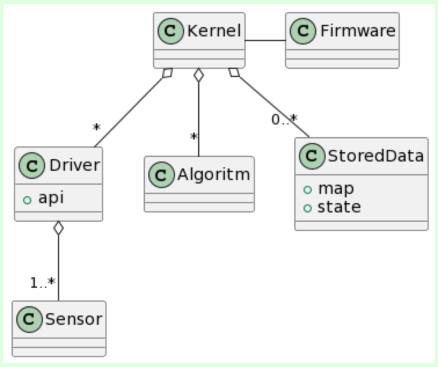


Figure 10: Rvc System Design

8. Deployment Diagram

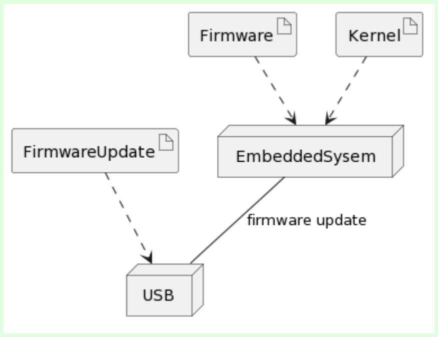


Figure 11: Rvc Deployment Diagram

6.4 Use Cases and Scenarios

A **scenario** is a story. To build a scenario we start from the **context diagram**, and the focus of the scenario is about explaining how the actors interact with the interfaces. Every scenario described is described with steps about what is going to happen.

On the other hand the **usecase** is the visual description of how the actor are related to the functional requirements of the system and the steps that they have to take.

7 Non-Functional Requirements

Non-functional requirements are some kind of properties associated to a functional property. There are some defined standard non-functional properties which are: usability, efficiency,

Usability: usability could be measured in:

- effort to learn: to do this is experiment we can take a number of users (non-technical) and observe their responses;

Efficiency: represent the response time and the resource used (usually computed for the whole application, it's difficult to calculate the efficiency for a single functionality).

Correctness: capability of writing the right functionality, this is always unfeasible.

Reliability: it is the probability to encounter a defect over a period of time, the defects are bugs visible to the end-user.

Maintainability: is the maximum effort to operate a change on the software.

Portability:

Security: protection from malicious attacks.

Safety: the system is safe if it cannot harm any person or get into a hazardous situation, if it does so it needs to have some kind of control.

Dependability:

7.1 EZGas

Scenarios and Usecases

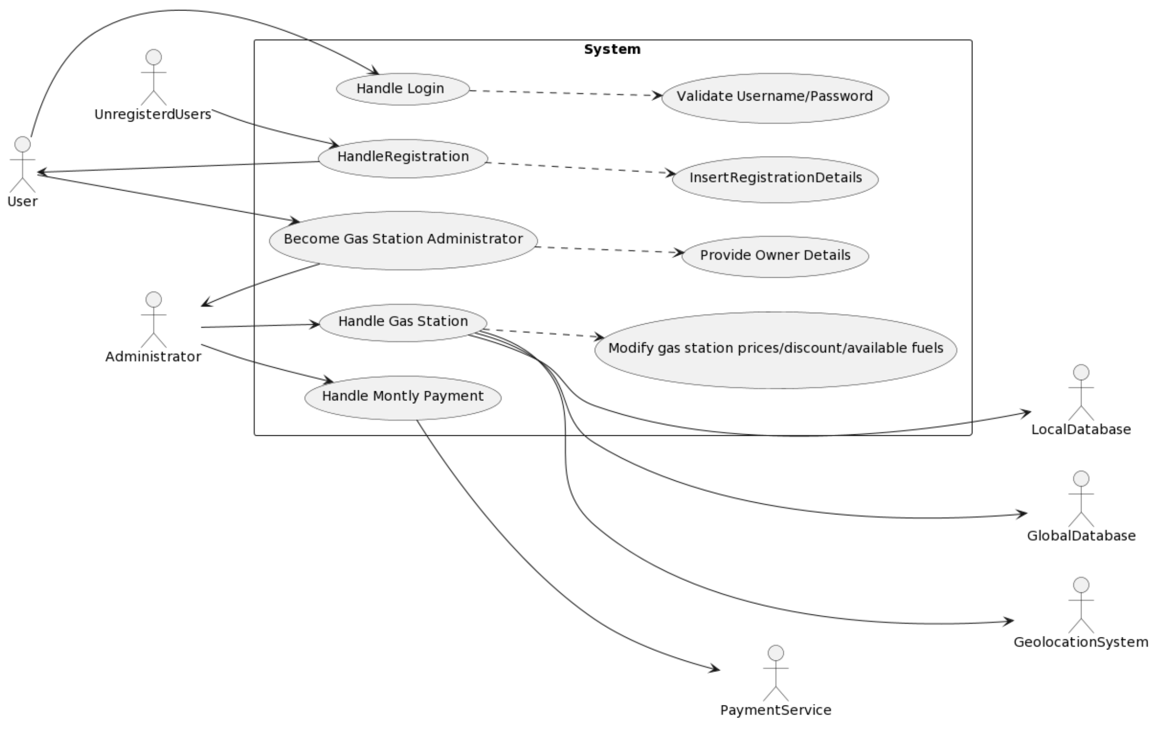


Figure 12: Ezgas Usecase Diagram, Link

- **UC: Login**
  - Precondition:
    - User is not authenticated and not authorized;
    - User must be registerd;
  - Postcondition:
    - User is authenticated and authorized;

1. User enters the app;
2. Inserts username and passwords;
3. Repeat until username and passwords are correct;

4. User is logged in;

- **UC: Registration**
  - Precondition:
    - Download the app;
  - Postcondition:
    - User has an account associated to a username and password;

1. User enters the app;
2. Click the sign in button;
3. Isert his username;
4. Repeat until a valid username;
5. Insert password;
6. Repeat until a valid password;
7. Insert email for password recovery;
8. Verify email;
9. User is now registrated;

- **User Needs Gas Station**
  - Precondition: user logged in;
  - Postcondition: user finds gas station;

1. Users may set custom parameter for searching a gas station;
2. User is prompted with a gas station;
3. User may decline the gas station, if declined repeat previus step;
4. User drives to gas station following the map instructions;
5. User arrives;
6. User completes his journey;

- **User Becomes GasStation Administrator**
  - Precondition: user is logged in;
  - Postcondition: user is now a gas station administrator;

1. User wants to become a gas station owner;
2. User enrolls to become a gas station owner;

3. User provides his details: partita IVA;
4. User pays monthly fee;
5. User becomes Administrator;

- **Administrator Add GasStaion**
  - Precondition: user is an administrator, administartor has acces to local/-global db;
  - Postcondition: a new gas station is added the EZGas system;

1. Admin enters gas station owner area in the application;
2. Admin can add or remove his owned gas stations;
3. Admin adds all the kinds of fuel present in a gas station;
4. Admin can add or remove discounts present for a fuel in a gas station;
5. Changes are submitted;
6. Validation is received from the server;

Non-Functional Requirements

- The app shall be developer for iOS, Android, Web;
- Avarage users should be able to learn how to use the basic functionality of the app in 10 minutes
- The application must satisfy the requirements of GDPR (EU) and CCPA (California) for the satisfaction of the app security.
- A time of 20 hour person (5 persons in a week development) to add a new feature.
- To develop an app for iOS, Android, Web shall take 6 months for the initial deplyment.
- The map must be updated once a week to avoid bringing the users into hazardous situations like: work in progress zone, accident site, ...
- The application server needs to have an *up-time* of 99% during the span of a year.
- The application

8 User Interface

Typcally the user interface represent a human interacting with the graphical interface of the program. Usually when we start to build a UI for our applications we need to start from: **context diagrams and actors**, **function requirements** and esecially **use cases**. When looking at our use cases we can the steps that an actor is performing and the UI should represent this steps in simple way. A key idea to always remember is to *keep the UI simple*, in fact *"A user interface is like a joke. If you have to explain it, it's not that good"*, also another key aspect of implementing a UI is **experimentation**.

The context is also important, in fact nowadays the applications can be developed for many differet devices, like: desktop, web, mobile phone, smartwatch, etc...

As a principle we should use the **simpicity**, we can represent it with three different axes, which are all related to each other, and we should remain as close to the orgin as possible.

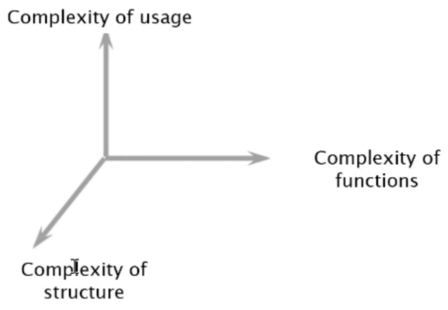


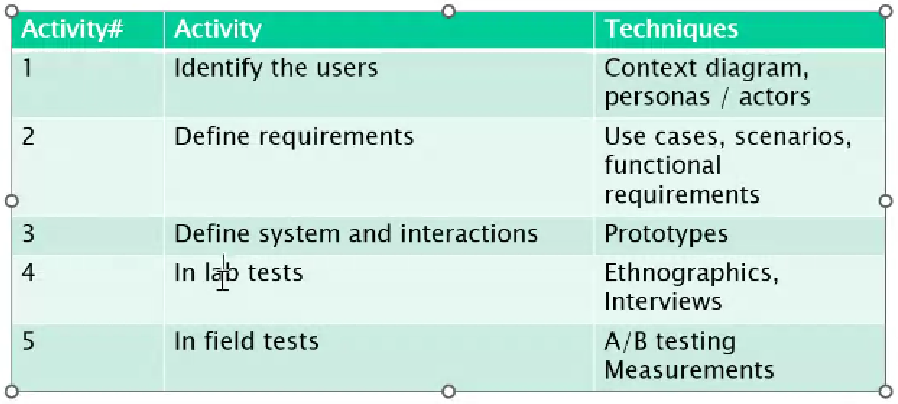
Figure 13: Ui Simplicity

Obviously the simplicity of the interface depends on the funcionality that it is built on, if for example we are building an application to pilot a shuttle to the ISS, it's not a very good idea to implent the controller with three buttons, obviosly shuch application will have thousends of lines of code and hundreds of buttons to control every funcitonality, that's way the context of build a UI is very important.

The main approaches for designing an interface are:

- **Ergonomy**: safety, adaptability, comfort, usability, one entry point, ...
- **Emotional Design**: is about the emotion about using a certain device raise in a user using it, like Apple products.
- **User Experience (UX)**: usability + feelings + emotions + values.

We also need to keep in mind the at the end of day the end-user will interact with the UI and so our choices needs to based on *feedback from the real user*, and not on our personal opinion, this is called **User Centered Design (UCD)**. The UCD has five steps:



Activity#	Activity	Techniques
1	Identify the users	Context diagram, personas / actors
2	Define requirements	Use cases, scenarios, functional requirements
3	Define system and interactions	Prototypes
4	In lab tests	Ethnographics, Interviews
5	In field tests	A/B testing Measurements

Figure 14: User Centered Design Process

8.1 Prototype

A proptotype is a simplified version of the final version of the interface. The inital steps can be made on: paper, postit, sketches, ... (**Low Fidelity**: needs to be done fast). The next step is by producing **High Fidelity** UI, this is done by using a computer, there are many products to do this, like: **Balsamiq**, **Figma**, ... The final step will be to use an *actual GUI Builder*, which will generate all the software calsses and all the methods needed to link the software and his functionality with the GUI.

8.2 In Lab Tests

The test are done via interviews with the users or in a monitored environment.

8.3 In Field Tests

For example a way to get feedback from a deplyed system is the A/B test.

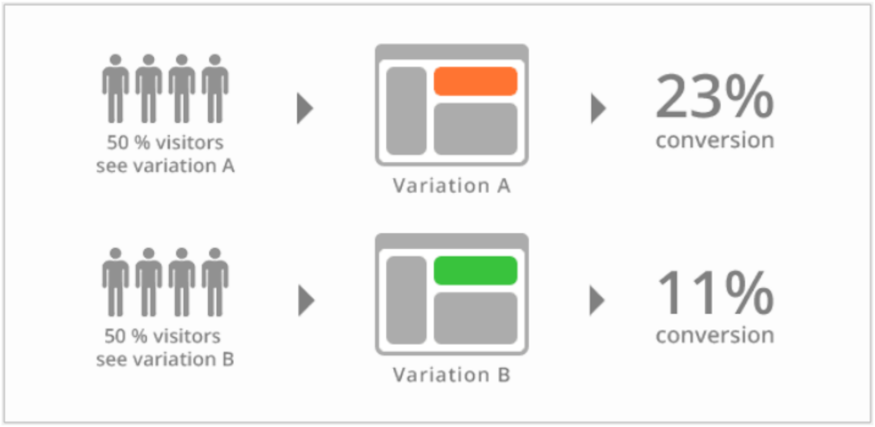


Figure 15: A B Test

This is typacally done: the users are split in two sets which there is a functionality has a variation injected to the two subsets, if the feature has a far greater use in a scenario then that will be the final choice on the application.

8.4 GUI Design

Usability Guidelines ......

After many years there are some conventions used in the design of a product:

- logo at top left;
- click on logo brings home;
- links change color on mouse hover;
- next/back is always in the same place;

9 Project Mangement

Before starting a project there are the Estimantion document which contains the the estimations for when the project will be ready and how much time the project will require, this is what project managemtn tries to answer to.

The planning of the project and the cantract between company and buyer are called **Inception Phase**, this phase is payed by the company which is producing the estimation documents, only if the buyer will accept all the terms then there will somt profit, otherwise the company operates at a loss.

During the **Development Phase**, it's important to keep track of the time and resources pent on the project.

- **Project**: collaborative endeavour to achieve a goal, wiht define limits of time/-money;
- **Program**: managemtn of several related projects (e.g. *Apollo Program*);

A project can be:

- **bespoke**: e.g. polito app;
- **commercial**: e.g. MS Windows;

9.1 Measures

Some of the measurament variants are: estimated value, actual value, targe value (what is desirable), benchmark (what the others do).

Project is basically a way to keep toghter: calendar time, cost, functionality and quality.

The first measure is **Duration**, which is dived in absolute and relative, eventually relative will be transalted into absoute.

Effort is defined as $p * h$, where p is the number of persons and h is the number of hours. What is the realtion between cost and price? The siplest model is: price = cost + margin.

TCO: ...

The **Size** of a sofware product is usually defined as Lines of Code (LOC) even though it's not very precise.

9.2 Techiques

WBS Work Breakdown Structure: you have a certain activity, and then this is divied in smaller activities (divide and conquer), in WBS there is no temporale relationshipt.

Gantt Chart

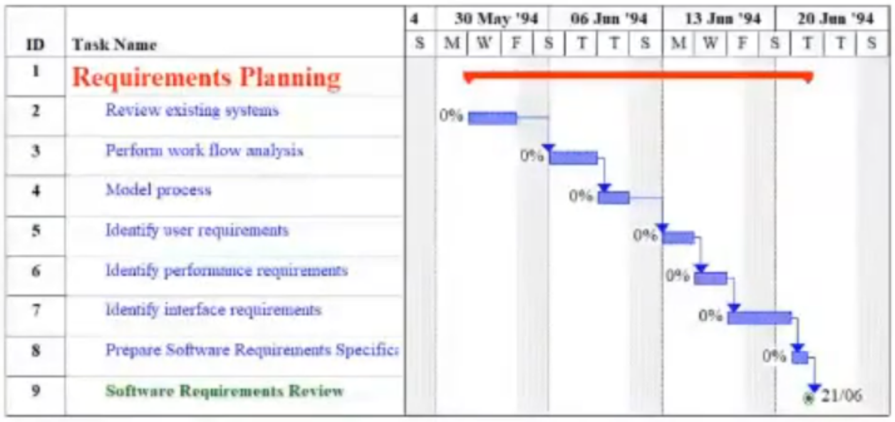


Figure 16: Gantt Chart

On the left there are the various the WBS, on the right each bar is calendar time of each WBS, and they are put with temporal relationship.

The **PBS (Product Breakdown Structure)**, is the heirarchical structure of a project.

The planning process aims to:

- estimate the effort and cost;
- identify acitivities: PBS, WBS
- define schedule: Gantt

9.3 Estimation

Estimating a project is more easyer with more experience.

Other ways (more objective) to estimate a prject are the following techniques:

- **By decomposition**: based on WBS or PDS, then the various parts are summed up to get the totale effort;

9.4 Tracking

How to keep track of the project? The first step is to collect data, to do that we need to collect

9.5 Staffing Profile

Typically, by increasing the number of people working on a single feature it increaces productivity up to a certain point, and then it starts to decrease, this shape follows a Bell's Curve.

9.6 Stories and personas

(Mettere in Requirements)

How to define how the people using our app will behave? One solutoion is using personas, this comes from the technique of fragmentation. For example people can bertitioned by:

- **Domographic:** age, gender, income, religion, level of education, ...

A persona idintifies and describes typical users, and we target the application to a subset of personas, obviously it's not possible to define every subset of people.

9.7 Elicitation

(Da finire: Requiremnt) We take a group of people wich coul represent the personas.

9.7.1 Interview

Interview people with a set of prepared questions

9.7.2 Ethnographics

The data collecotr is not present in the room and the people aare alone in the room, and the collector observers how to peaple behave with the app, this is also done online with tracking tool that observer how we intercat with the applications.

9.8 Exercise

Exercise 1: ...

Exercise 2: A new software project has an estimated size of 210000 lines of code. Past similar projects in the same company had a productivity 30 person day.

1. Esimate the effort for the new project: ouput / effort = LOC / effort 210000 loc / 30 loc/pd = 7000 person day
   7000 / 140 = ... p month

2. Estimate the duration of the porject, assuming to have 4 people available to work on it.

   7000 pd / 4 p = 1750 calendar days = 12.5 calendar months

3. on avarage the cost of a person day is 150 Euros. Estimate the cost of the prject.

   150 * 7000 euro

Exercise 3:

1. Compute the effort (peron hours) estimated for developement, according to the article (assume a person wrokd 140 hours per calendar month):

   5 * 2 * 140 = 1400 ph

2. Compute the cost per person hour, during developemnt:

   300000 / 1400 = 214 euros per person hour

3. Discuss the estimates of time and cost for development.

   2 months seems very short time, cost very high

4. Using the cost per person hour at point 2, compute the person hourse estimated fot eh maintanance phase, per year.

   200000 / 214 = 1000 ph = 7 pm

10 VV, Inspection, Reading

Whatever we produce can contain bugs and errors, this is why there is the last part wiich is the inpection of the work done, ...

Verification and Validation: checking that what we did is correct. Validation: doing the right software system valued by the stackeholders. Verification: is doing correctly the internal transformation.

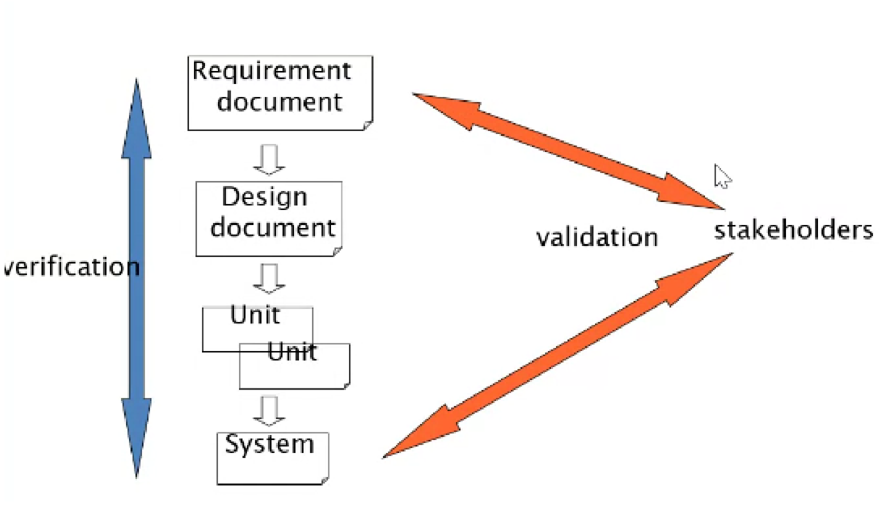


Figure 17: V&V

10.1 Failure, Fault, Defect

Failure is an execution event where the software behaven in unexpected way. The end-user catched the problem

Fault: a feature in software that causes a problem, error in the code or in documents.

A failure is caused by a fault. A failure may never araise, the simple reason is that that part of the code/documnents is unused.

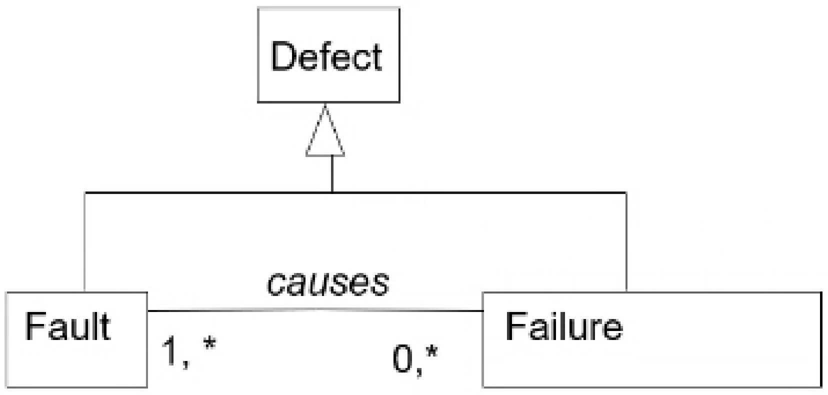


Figure 18: Failure Fault Defect

Someties it's not clear what is the fault and what is the failure so can both call them defects.

10.2 Inspections

This is done for document checking (also applyable to code), is a reading technique and it uses group dynamic, the aim is to **find defects**.

the aim is to find the **taxonomy of the defetcs**. Find:

- inconsistency
- abiguity
- ...

Another tachnique is to do a **checklist**.

11 Architecture and Design

Requirements: what the system shoud do; Architecture: how the system should be built. In general many differnt designs are possible, but there are some common desgin patterns. After many iterations of a disign for a product a **dominant design** will araise.

Software design is defining modules and their interactions so that they satisfy function and non-functional requiremnts (usually the most difficoult to satisfy).

Architecture: high level components are designed with their interactions, a comunication and coordination model is selected, like: process, threads; messages, procedure calls; broadcast, blackboard; so we need to select either one of them.

Design: is about high level decisions inside each component.
There are some non-functioal properties for architecture design:

- testability
- monitorability
- interopelability
- scalability
- deployability
- mdobility

Other non-functional properties are:

- complexity: minimize number of components and number of interations

Coupling: modules shoul have least amount of coupling between them, the coupling is the degree of dependance between two components.
Cohesion: we should put in the same components parts that are related to the component

- higher: Engine: (start(), stop())
- lower: Engine: (start(), stop(), monitorObstacles()), monitorObstables() doesn't fit inside the Engine module, thus causing lower cohesion

...
Cost: some functionalityies are more expensive than others
Schedule: if there are more than one component they can be developed in parallel
Staff skills: the staff needs to be trained to work on a specific component.
...
Performance: minimaze number of operations and comunications, larger components
Security: use a layerd architecture, like a firewall
Safety: more control on seaty-critical features, make the subsystem
Availability: add redunt components
Maintanability: replacable components
Performance vs Security; Availability vs Security, Performance; there conficts we need a trade off, we need to choose which non-functional requirement is more important, defined by the stakeholders.

11.1 Notations

A simple box and line diagram, where a box is a component and an arrow is an interaction. These can also be done with UML class diagrams or packages diagrams. The sequence of function calls should never be displayed in the class diagram, to do that there is the dynamic view, that may correspond to the implementation of a use case.

11.2 Patterns

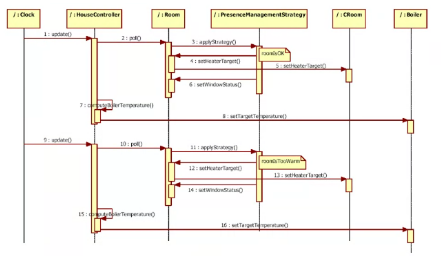


Figure 19: Sequence Diagram

Patterns are similar to dominant design in other engeneering fields, to solve common recurring problems.

11.2.1 Architecture Patterns

Repository style: each interaction happens through a file system

Client server:

ISO/OSI model:

3 tier architecture: persentation, application login, data.

Pipes and Filters: like linux pipes

MVC (Model View Controller): The proble with have a graphical part to display. The simplest way is to have a single class, but everithing gets messy really quickly, and having more than one representation we will have to write everithing from scratch because the logic and graphic part are tightly coupled togheter, one way of extraciting that is using a `Model` of the the grohpical part wich will contain the logic while the `View` will only contain the graphical representation, the `Controller` is what controls the flow of data on logic, for example Android uses MVC, the `Controller` is the generic `View` class, his `View` is the XML file associated, while the `Model` is the data associated with the current XML file. Usually the cycle is: `View` fire events, `Controller` update model, `Model` update view.

Miccrokernel:

Microservices: