Notes Software Engineering

Brendon Mendicino

March 20, 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 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 2.1 – 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 2.1 – 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

2.1 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 2.2 – 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

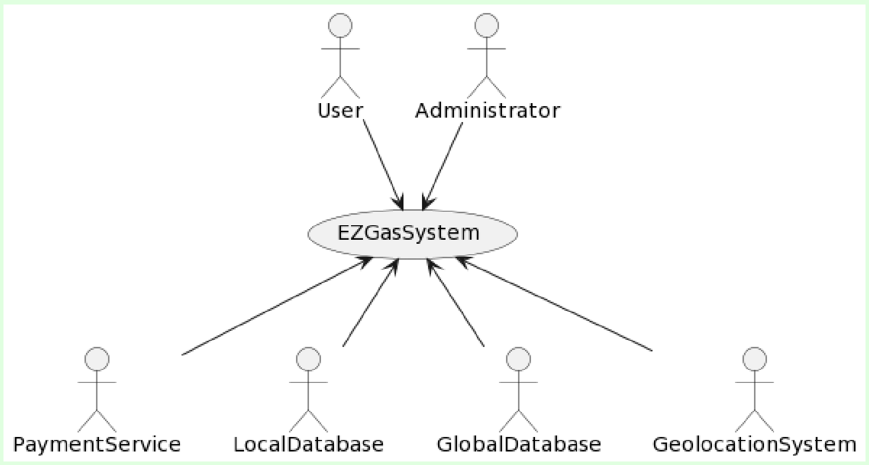

Figure 1: Context Diagram EZGas

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		✓

6. Glossary

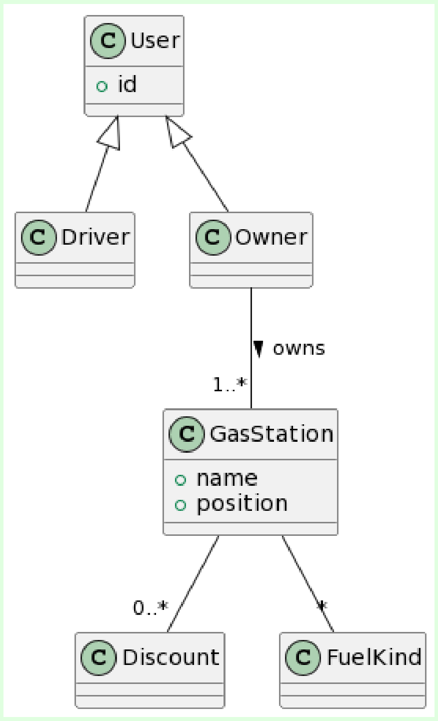

Figure 2: Ezgas Glossary

7. System Design

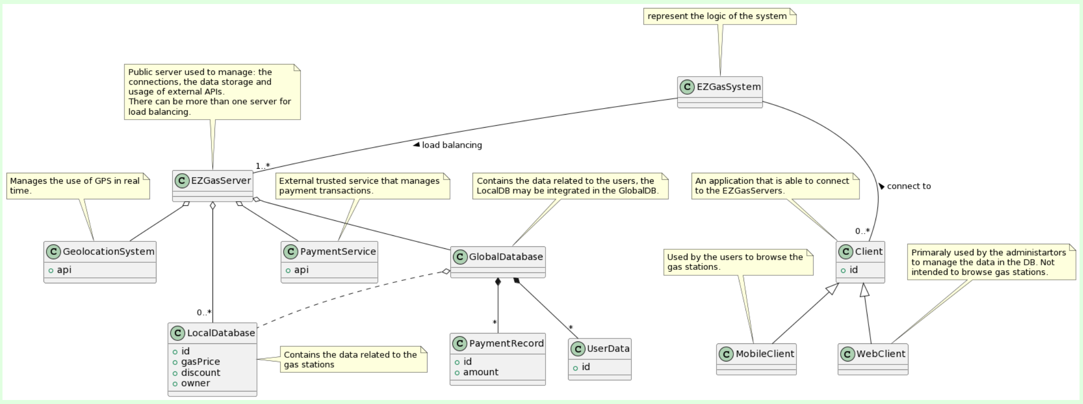
represent the logic of the system
EZGasSystem
Public server used to manage: the connections, the data storage and usage of external APIs.
There can be more than one server for load balancing.
load balancing
1..*
connect to
0..*
Manages the use of GPS in real time.
EZGasServer
External trusted service 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
Used by the users to browse the gas stations.
Client
id
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 3: Ezgas System Design

8. Deployment Diagram

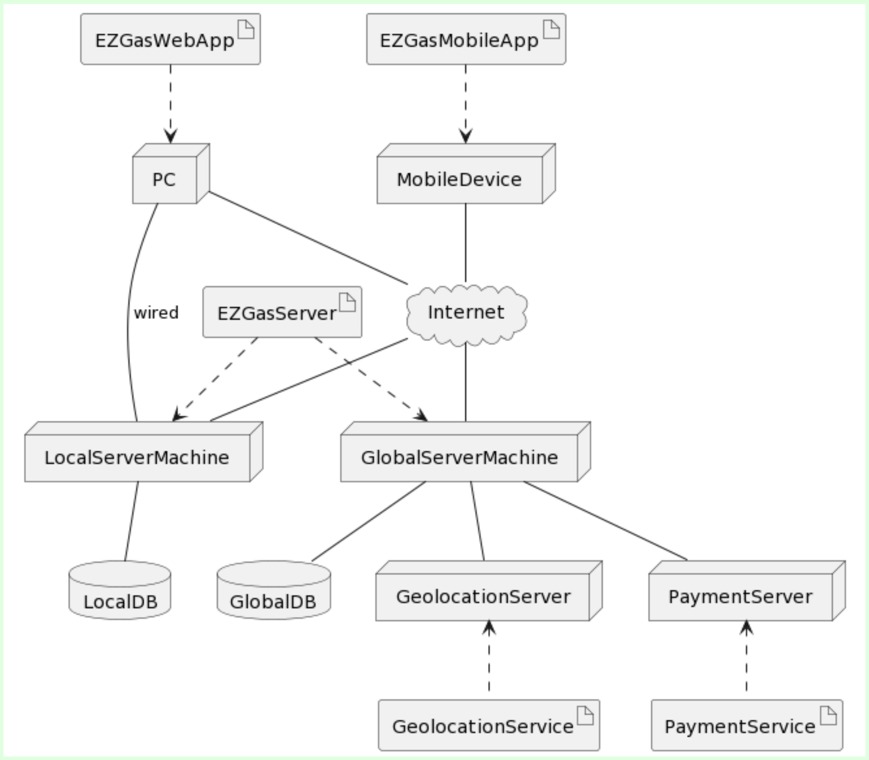

Figure 4: Ezgas Deployment Diagram

Example 2.3 – 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:**

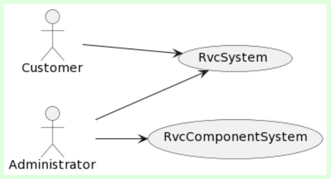

Figure 5: RVC Context Diagram

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

4. Non-Functional Properties

5. Table of Rights

6. Glossary

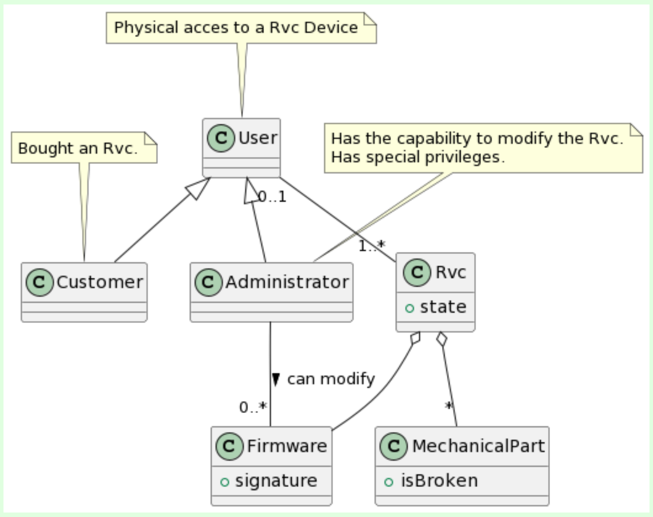

Figure 6: Rvc Glossary

7. System Diagram

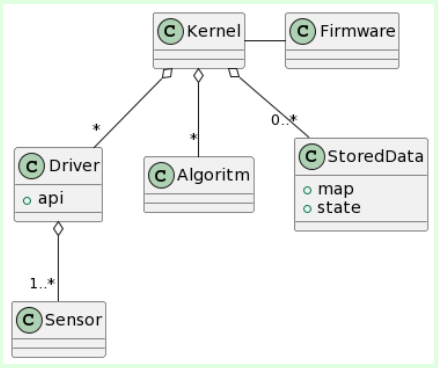

Figure 7: Rvc System Design

8. Deployment Diagram

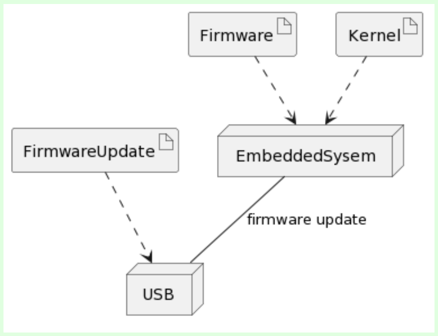

Figure 8: Rvc Deployment Diagram

3 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.

3.1 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:

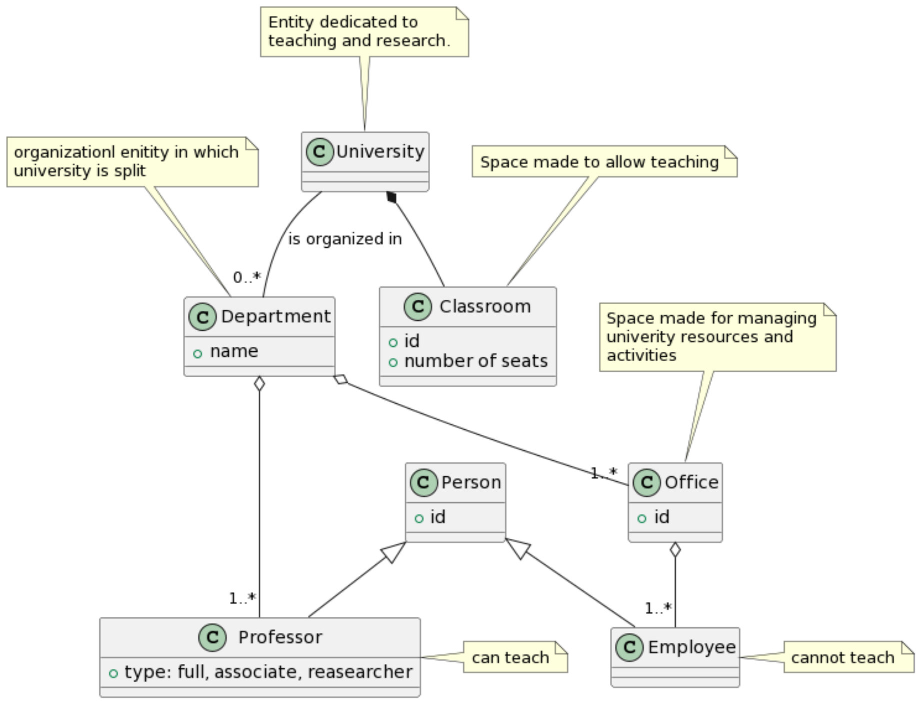
Entity dedicated to teaching and research.
University
organizationl enitity in which university is split
Space made to allow teaching
is organized in
0..*
Department
name
Classroom
id
number of seats
Space made for managing univerity resources and activities
Person
id
1..*
Office
id
1..*
1..*
Professor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

Figure 9: University UML Diagram

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

3.2 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.

3.3 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**.

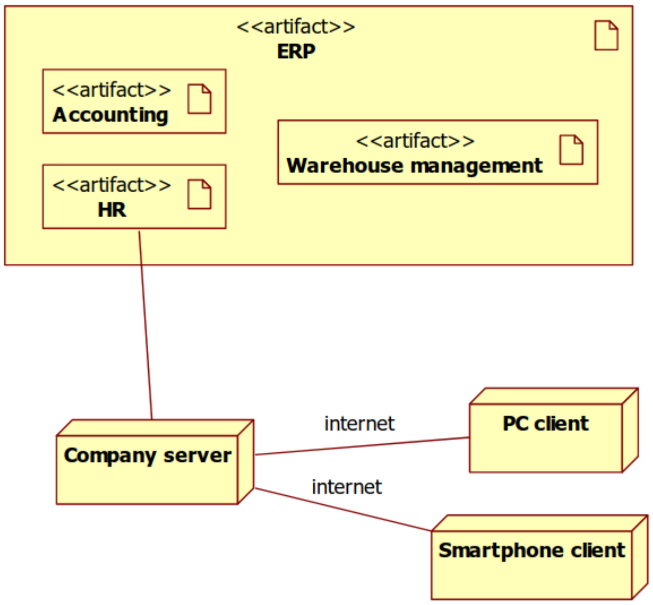

Figure 10: Deployment Diagram Example

3.4 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.

4 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:

5 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
  - $\sim 4 \implies$ number of commits we want to target;
- `$ git merge <other-branch>`: this command tries to merge `other-branch` inside the current working branch.

Example 5.1 – Revert cahnges

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

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

Example 5.2 – 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 5.1 – 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