Notes Software Engineering

Brendon Mendicino

March 6, 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.

3 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 rabasing 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 trak of every commit where done in different branches.

- `$ git reabase -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 3.1 – 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 3.2 – 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 3.1 – 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