SOTFWARE ENGINEERING (SE)

The topic is not a small code program, but softwares with millions LOC (Lines of Code) for customers who don't know nothing about coding. Companies are becoming softwares companies.

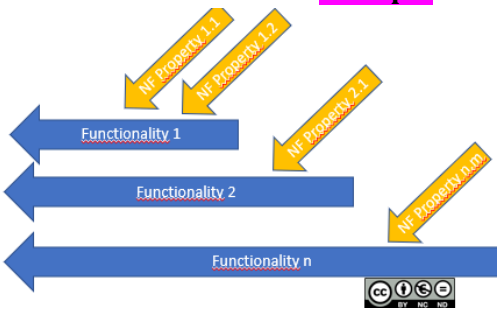
Project must be developed in parallel with course (antiplagiarism tools will be applied, teams are formed by professor) [to be produced: Requirements document, Design document, Code, Test suite – unit level, integration level, Test suite, API level, Change Request]

0) INTRODUCTION

SOFTWARE ENGINEERING = multi person (issues in communication and coordination) construction of multi version (issue in maintenance) software

SOFTWARE = collection of programs, procedures, rules, associated documentation, data, types (embedded, stand alone, embedded in business process [IS],…); software criticality are safety critical (harms people or environment [es. embedded]) or mission critical (harms business)

PROCESS (activities, people, tools) → **PRODUCT** produced (document, data, code…) [quality of product depends on quality of process]; process of software engineering is composed by design software product, design software process, deployment and delivery, maintenance. We use the **example** of traffic light controller to describe the **PROPERTIES** of:



- **SW PRODUCT**
  - **Functional** (**F**)
    - Functionality = the effective function (es. control 4 traffic lights in a road crossing so that…)
  - **Non functional** (**NF**) [more difficult to engineer, often forgotten] **(*)**
    - Correctness = provide the correct functionality in all cases (es. the intended sequence of signals is always satisfied)
    - Reliability (es. the intended sequence of signals could be not as intended with a certain rate)
    - Availability (es. the controller can be down, not providing any sequence)
    - Security = protection and controlled access (es. blocking malicious attack to the controller)
    - Safety = absence of harm to people (es. green in all directions)
    - Dependability = Safety + Security + Reliability
    - Usability = effort needed to learn using the product, satisfaction (es. how easy is to setup the controller)
    - Efficiency = response time, energy used (es. latency in computation of next light values)
    - Maintainability = effort to fix and to deploy on a different platform (es. adding a light to control the left turns)

  ⚠ Both functional and non functional should be clearly defined before development

- **SW PROCESS**
  - Cost
  - Effort (hours)
  - Punctuality (promised date vs actual date)

- Conformance

The software workbench is repository + version and configuration management tools (Git, Subversion, …).
The software tools require context diagram, stakeholders, personas, use cases, functional and non functional requirements, glossary… We divide them in the process phases in which they are used:

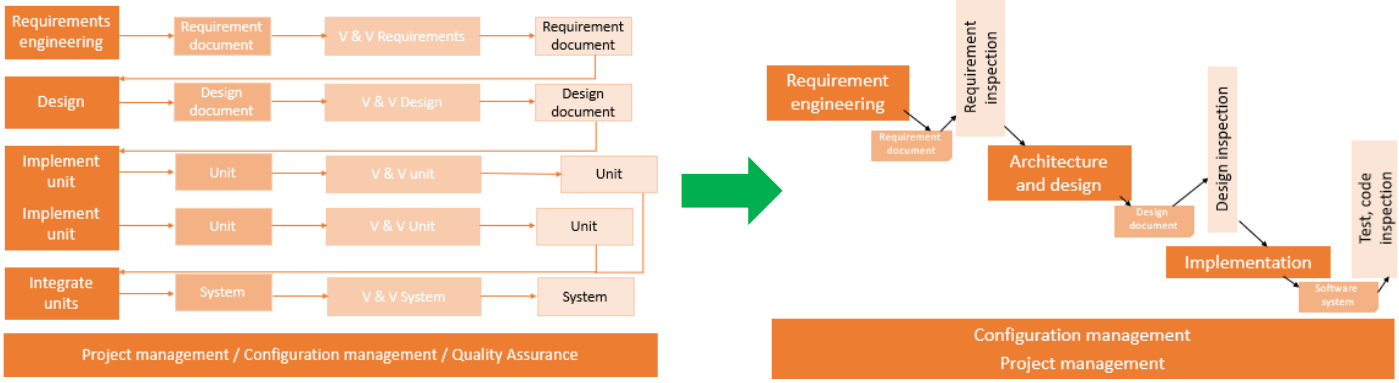
- *Design* = component diagram, package diagram, class diagram, interaction diagram …
- *Development* = VSCode, Eclipse, PyCharm …
- *Test* = unit test (white box and black box) …

LAWS:
- Requirements deficiencies are the prime source of project failures
- Requirements and design cause the majority of defects
- Defects from requirements and design are the more expensive to fix
- Modularity, hierarchical structures allow to manage complexity
- Reuse guarantees higher quality and lower cost
- Good designs require deep application domain knowledge
- Testing can show the presence of defects, not their absence
- An evolving system will increase its complexity, unless work is done to reduce it
- Developer productivity varies considerably
- The process should be adapted to the project
- Conway's Law = structure of a system produced by an organization mirrors the communication structure of the organization

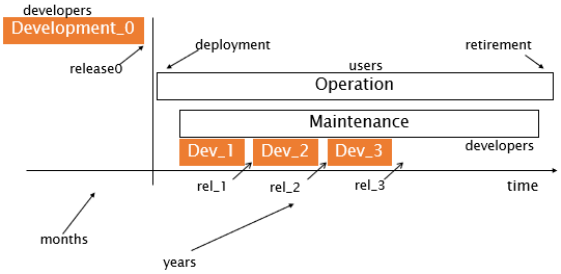
PRINCIPLES:
- KISS (keep it simple stupid)
- Accidental complexity
- Separation of concerns
- Abstraction (given a difficult problem or system, extract a simpler view of it)

1) PROCESS

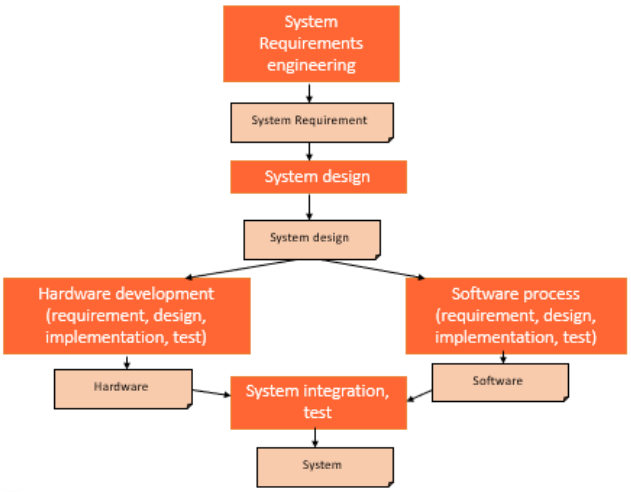
GOAL = produce software (documents, data, code) with defined process properties (cost, duration) and product properties (functionality, reliability …); the approach is **BOTTOM UP** (to obtain the final thing we start from the bottom activities), so REQUIREMENT ENGINEERING [**REQ**; what the software should do] → ARCHITECTURE + DESIGN [**DES**; which units and how organized] → IMPLEMENTATION [**IMP**; write code] → INTEGRATE UNITS. But we have to insert the VALIDATION & VERIFICATION ACTIVITIES (**V&V**) that control requirements, design, code correctness, but also the MANAGEMENT ACTIVITIES (project management and quality assurance):



After **DEVELOPMENT** (**DEV_0** [REQ_0 + DES_0 + IMP_0]), we have **OPERATION** (from the ***users***) and **MAINTENANCE** (from the ***developers*** [the **DEV** phase of each maintenance is usually shorter than the DEV_0 and depends from the DEV before → DEV_1 = REQ_1 based on REQ_0; DES_1 based on REQ_1; IMP_1 based on DES_1]).



⚠️ **System vs Software** Process: stand alone software requires software process, embedded software requires **SYSTEM PROCESS** that is:



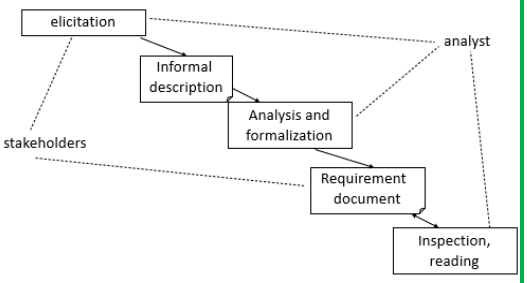
There are **3 approaches to SE** (other than "cowboy programming" that is just coding):

1. **Document based, semiformal** (semiformal language for documents [UML], human transformations)
2. **Model based, formal** (formal languages for documents, automatic transformations)
3. **Agile** (limited use of documents, emphasis on code and tests)

⚠️ We focus on 1!

2) REQUIREMENT ENGINEERING (RE)

REQUIREMENT ENGINEERING is about defining the product properties (functional + non functional) before starting development (without it product properties are unclear and testing is not feasible). The output is the Requirement Document. The ***steps*** of RE are →



⚠️ Requirements are requests, that may or may not become properties of the product (es. requirement = response time < 0.5s; property = response time between 1s and 3s)

As we can see from the image, usually the starting point is an **informal description** by a client or a potential user to the developer; but problems arise when requirements are ambiguous and can be interpreted in different ways by developers and users. So in theory, requirements should be both:

- **complete** → include descriptions of all features required
- **consistent** → no conflicts/contradictions in the descriptions of the system features

Requirements defects are:

- Omission/Incompleteness
- Incorrect fact
- Inconsistency/Contradiction
- Ambiguity
- Extraneous information (overspecification)
- Redundancy

TECHNIQUES FORMALIZATION:

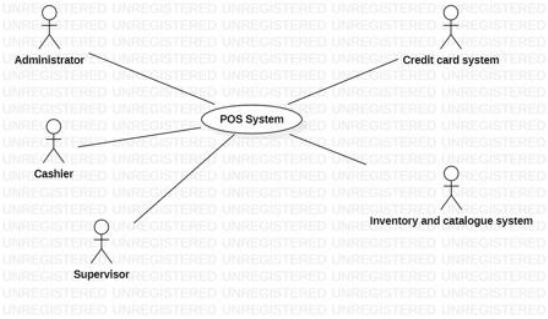
- **STAKEHOLDERS** = ask the client an informal description of the product to build; it's an entity involved in the project, who may affect/be affected by the output of the project (important because consider all relevant points of view); example with POS (Point of Sale) in a supermarket:
  - User → uses the application (es. Cashier at POS [profile 1], Supervisor [profile 2], Customer [indirectly through cashier])
  - Buyer/Commissioner → pays for the application (es. CEO and/or CTO or CIO of supermarket)
  - Admin (es. POS app admin [profile 3], IT admin [profile 4], security manager [profile 5], DB admin [profile 6])

- Analyst → expert in RE and/or in the domain
- Developer

- **BUSINESS MODEL** = how the application is funded? (basing on the option chosen, there are requirements for the implementation of services [es. payment system, ads …])
  - organization pays for development and operation + maintenance, users use it for free (Polito App)
  - organization pays for development and operation, users pay to use it (Netflix)
  - users don't pay but must watch ads, organization is paid for each ad shown (Facebook)

- **PERSONAS**, **STORIES** = informally define what the application should do (for mass market application) basing on profiling/segmentation (classify people to define and sell targeted products/services) [profiling also happen in website that monitors our activity: es. first time visitor = % discount on firts order; returning customer = propose similar products to ones previously bought]

- **CONTEXT DIAGRAM** = define what's inside the application to be developed, what's outside (entity outside = actor with *actor* ⊂ *stakeholder*) + defines interfaces between inside and outside. Image left below
- **INTERFACES** → Image right below



Actor	Physical interface	Logical interface
Cashier	Screen, keyboard	Graphical User Interface (to be described, ex slide 39)
Product	Laser beam	ReadBarCode (to be specified, ex slide 37,38)
Credit card system	Internet connection	API description (ex https://developer.visa.com/docs for ViSA APIs
Administrator	Screen, keyboard	Graphical User Interface + command line interface

3 types of interface may have to be defined: user interfaces (GUI), procedural interfaces (Java), data exchanged (XML). Formal notations are an effective technique for interface specification

⚠️ Context diagram and interfaces are essential to agree on what is the scope of the system (and which requirements/functions should offer/use from outside)

- **REQUIREMENTS** (F & NF):
  - Functional → description of services/behaviors provided by the system; clearly separated and hierarchical numbered requirements (**F_number**) for better recognability of the F requirement

Requirement ID	Description
F1	Handle sale transaction
F1.2	Start sale transaction
F1.3	End sale transaction
F2	Authorize and authenticate
F2.1	Log in
F2.2	Log out
F2.3	Define account

  - Non functional → constraints on services; application vs domain. They are divided in **Product**, **Organisational** & **External** requirements. They must be measurable (speed, size …), but are not testable. The system should be easy to use, maintainable and portable. Are described before **(*)**:
    - ***Usability***
    - ***Efficiency***
    - ***Reliability***
    - ***Maintainability***
    - ***Portability***
    - ***Security***
    - ***Safety***

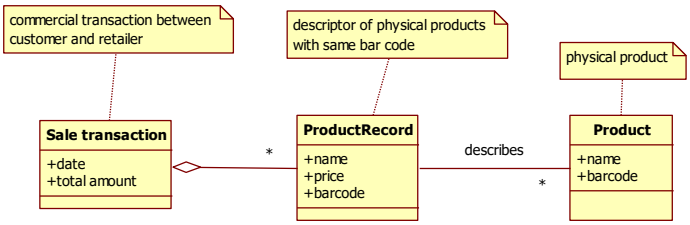
USABILITY	Application is simple and readable
EFFICIENCY	Response time for F2.1 < 1 ms
RELIABILITY	The application is resistant to big number of users (< 15k per time)
MANTAINABILITY	Application bug fixes done in 5 hours on average
PORTABILITY	Application can be used on PC or smartphone
SECURITY	Password requirements for sign in
SAFETY	Sending email to confirm password change and location of access

- **TABLE OF RIGHTS**:
  - COLUMN = actors (those who are in the context diagram and interface)
  - ROWS = primaries F requirements
  - SINGLE CELL = Yes/No if the actor is involved in the F requirement

	Gas station owner	driver	admin	Application store	Payment system	Google map
FR1	YES	YES	NO	NO	NO	NO
FR2	NO	YES	NO	NO	NO	YES
FR3	YES	NO	NO	NO	NO	YES
FR4	YES	NO	YES	NO	NO	NO
FR5	YES	YES	NO	NO	YES	NO

⚠️ Key stakeholders must decide the ranking of NF requirements and it's a business decision (not a technical one) and shouldn't be taken by developers. There are also domain requirements (*external* requirement; derived from the application domain) that can be new F requirements or constraints on existing F requirements; problems of domain requirements are understandability and implicitness

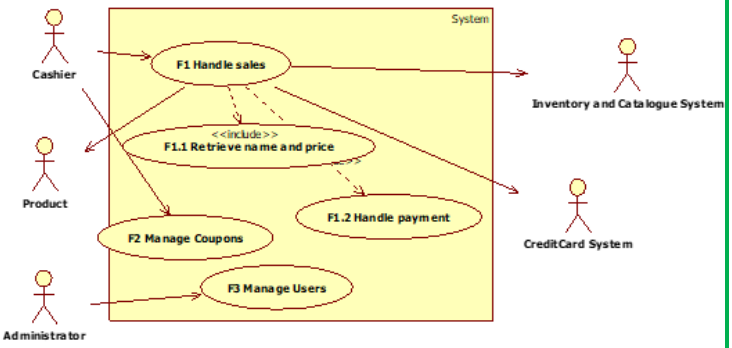
- **GLOSSARY** (glossario) → object name + definition of the object (es. Sale = commercial transaction between customer and retailer, Product = …). We can use also Glossary with Class diagram (**UML**):



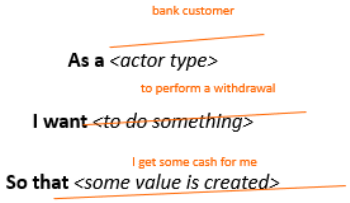
- **SCENARIOS (SEQUENCE DIAGRAMS)**:
  - **SCENARIO** = sequence of steps that describe an interaction between a user and the system, with a precondition [have to be satisfied *before* steps] and postcondition [satisfied *after* steps] (es. precondition in the image = "cashier is authenticated"; postcondition = "sale completed with payment"). **Time** is only part of this model (**not** requirements, Glossary, Context diagram, Use cases…)

Step	Description
1	Start sales transaction
2	Read bar code X
3	Retrieve name and price given barcode X
	Repeat 2 and 3 for all products
4	Compute total T
5	Manage payment cash amount T
6	Deduce stock amount of product
7	Print receipt
8	Close transaction

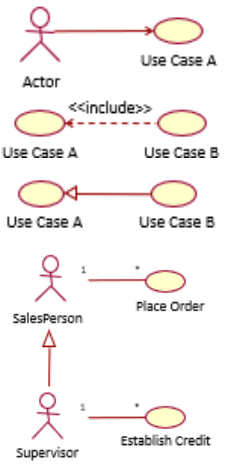
- **USE CASE (UCD)** = set of scenarios with common user goal (es. Use case = "handle sales"; Scenarios → Scenario1 = "sell 2 products", Scenario2 = "sell 3 products, abort sales because no money", Scenario3 = "…"). Let's see an example of USE CASE DIAGRAM (in which the primary actor initiates an interaction with the system to accomplish some goal, and the system responds protecting the interests of all the stakeholders). The diagram is composed by:



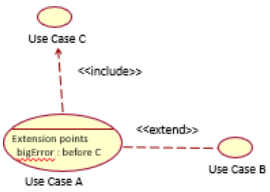
  - ***Actor*** involved (type of user that interacts with the system) → external to system; they can be humans or machines/systems. *Primary* (start the interaction with the system) and *Secondary* (passive
  - ***System*** being used (as a black box)
  - ***Functional*** goal (that actor achieves during the system) → structure =>



  - ***Use case*** (the ellisse) = a functionality of the system; **RELATIONSHIPS** can be:
    - Associations (between actor and use case [es. actor triggers use case A])
    - Include [es. functionality A is used in the context of functionality B]
    - Generalization [es. functionality B as a specialization of functionality A]
    - Generalization with also actors [es. a generalization from an actor B to an actor A indicates that an instance of B can communicate with the same kinds of use-case instances as an instance of A]



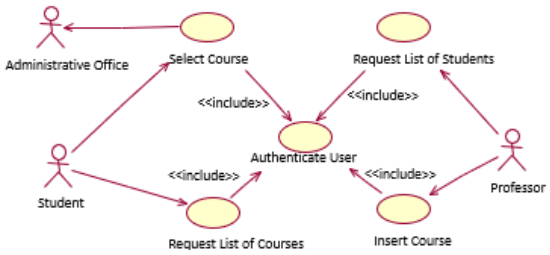
- Extension [es. an instance of use case A may be augmented by the behavior specified by B]



⚠ Functional requirements are finer-grained than use cases. High level requirement (es. F1, F2…) are translated as use cases that includes other use cases, while low level requirement as use cases <included> by other use cases

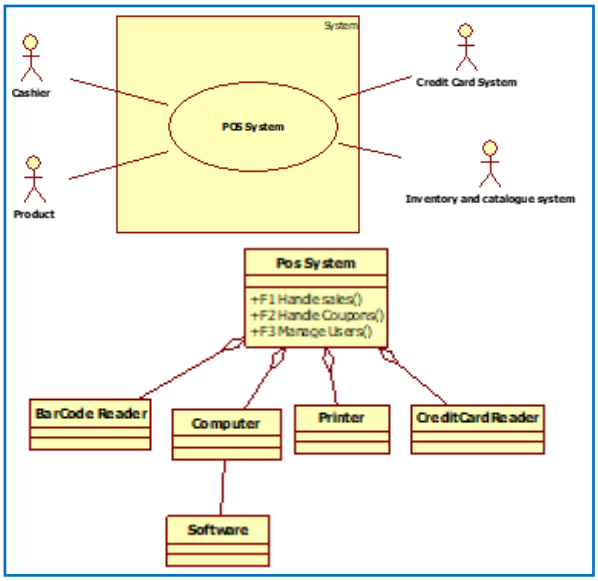
Example (student management):

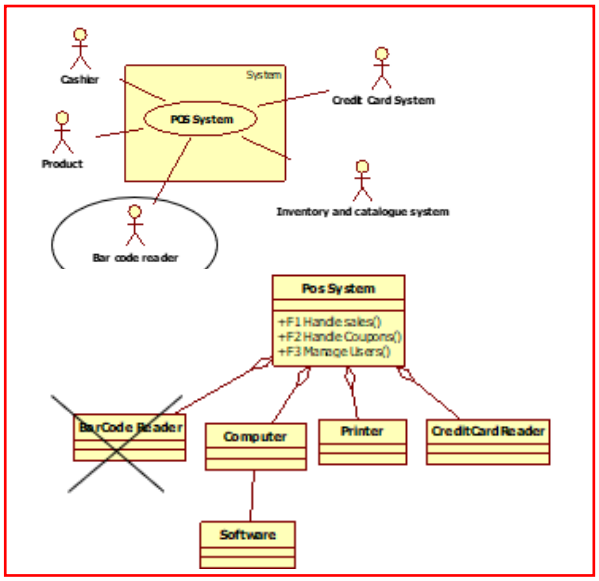
- students select courses
- professors update the list of available courses
- professors plan exams for each course
- professors can access the list of students enrolled in a course
- professors perform exams and then record issues of exam for students (pass/no pass, grade)
- all users should be authenticated



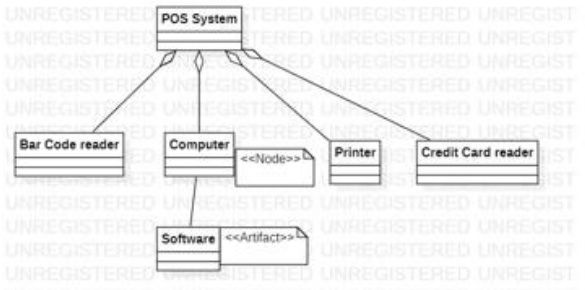
⚠ Use Case diagram and Class diagram must be consistent (actor may become a class, use case must become an operation on a class, interaction is not represented)

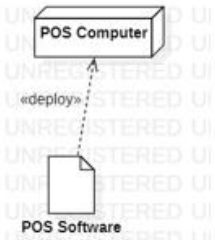
- **SYSTEM DESIGN** = subsystems (software and computing) that compose the system. Let's see an example of what changes if an actor (BarCode Reader) is before **internal** of the system (POS System) and after **external** between the Use Case diagram and the System Design diagram:





Classes that represent a computing component are Nodes, classes that represent a software component are Artifact. Let's also see the difference between system design and deployment:





The result of RE is the **Requirement Document** (that formalizes the requirements). It's composed by:

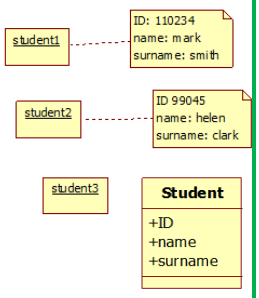
- **Overall description** (text)
- **Stakeholders** (text)
- **Context diagram and interfaces** (UCD + text)
- **Requirements (F, NF, Domain)** (type + numbering)

- **Use case diagram** (UCD or UC briefs)
- **Scenarios** (tables + text)
- **Glossary** (text or UML Class)
- **System design** (UML Class)

⚠ Other RE techniques are focus group, questionnaire, interview, ethnographics (researcher hidden in an environment observing from the intern [expensive, long, risk of being invasive]), V&V (inspections, prototyping, iterations, model checking on formal languages) [we see V&V later in the course], MVP (minimum viable product [try it with end users]).
But also structured presentation (text), form-based specifications (function, inputs, outputs, other entities, pre-condition, post-condition, side effects), tabular specification (condition → action)

⚠ Some definitions:
- **Domain** = collection of related functionality (or collection of applications with similar functionalities) [es. banking, telecommunication…]
- **Application** (or system) = software system supporting a set of functions; belongs to 1 or more domains

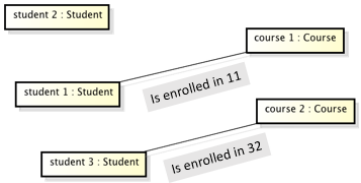
3) UML (Unified Modeling Language)



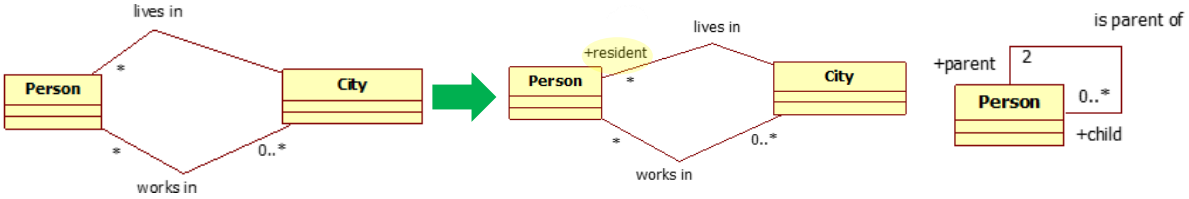
UML includes several diagrams (**Class** diagram, **Activity** diagram, **Use Case** diagram…):

- **CLASS Diagram** → language composed of:
  - Object (istance of a Class) = model of item [es. a student, an exam…] characterized by identity, attributes (properties), operations (behavior) and messages (it receives)
  - Class = describes set of objects
  - Attribute = elementary property of classes, characterized by a name and a type (es. Name: String)

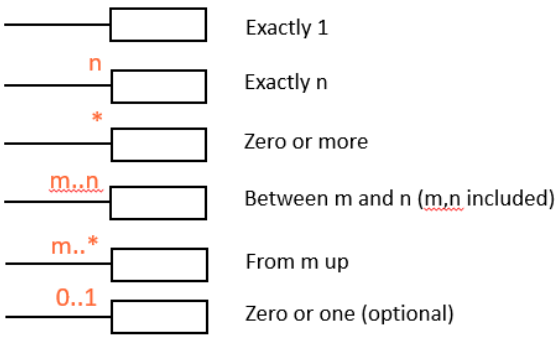
  ⚠ We use Class diagram in model of concepts (GLOSSARY), model of system (hw+sw, SYSTEM DESIGN) and model of software classes (SOFTWARE DESIGN). A class in the conceptual model (UML Class diagram) corresponds to entities in software application

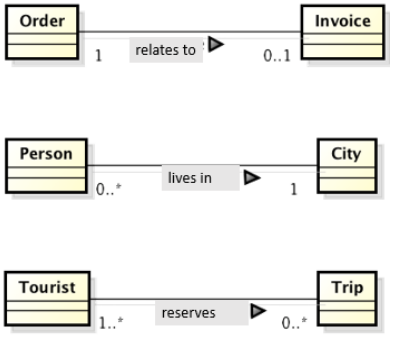


  - Link = property between objects
  - Association = set of links between objects of different classes or pairs of objects (1 per class) [es. {is enrolled in 11, is enrolled in 32} or {student1 – course 1, student1 – course3, student3 – course2}]. We see that we can have a role in an association and we can have also recursive associations:



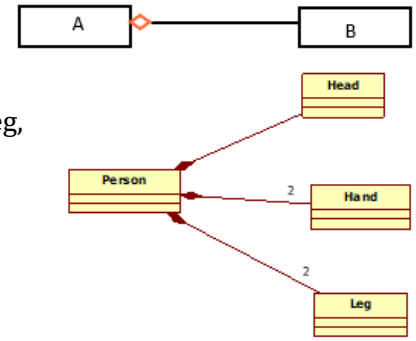
  - Multiplicity = max and min number of links in which an object of a class can participate (should be specified for each class participating in an association) [0 = optional, 1 = mandatory, 2, 3, …, * = many]



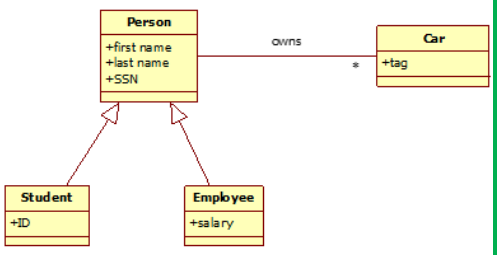


There are **3 cases of associations**:

- Aggregation (B is-part-of A; A has B)

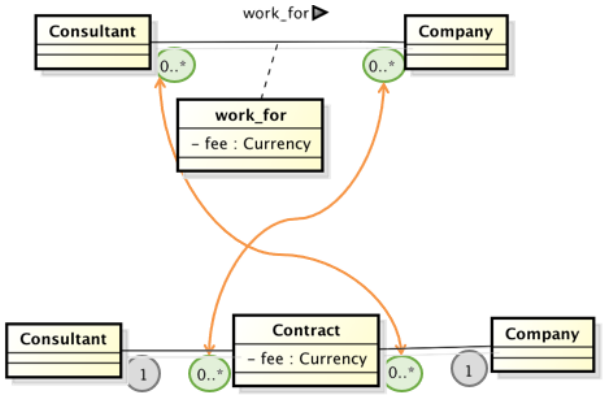


- Composition (aggregation more strict; es. if Person disappears, the objects Leg, Hand and so on disappear too)
- Generalization/Specialization (B is-a A [Employee is-a Person]): terminology of class involved:
    - Superclass, Ancestor class, Base class = 1 or more above
    - Parent class = above
    - Child class = above
    - Subclass, Descendent class, Derived class = 1 or more below



⚠ With this specialization, a Student can't be an Employee (and v/v) but both are Person; if we want to have Student as also Employee (and v/v) we can use instead roles in an association

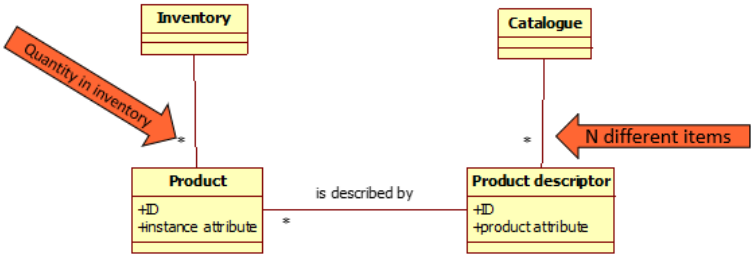
The **link** of an association can also have an attribute (this is an Association Class); an alternative is using an Intermediate Class (so that we can have more than 1 value for a link):

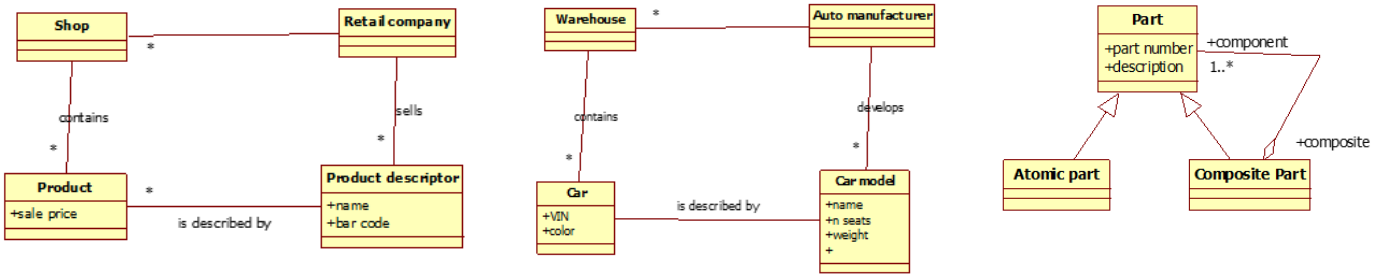


⚠ In UML Class diagram don't:

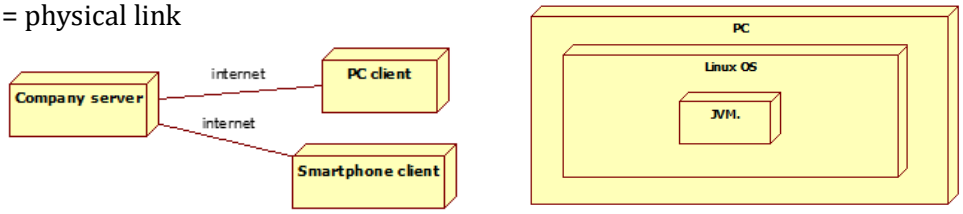
- Use plurals for classes (person, not people)
- Forget multiplicities
- Forget roles/association classes when needed
- Use class as an attribute (es. Address contains CAP, city, number; so we can't use an attribute address in the Citizen, because it's better to model it as a Class)
- Use attribute that represents more objects (es. using grades as an attribute in Student; it's better to create the Class Exam and putting grade in it, linking Student with Exam)
- Use transient relationships that represents transient events (es. creating 2 links, 1 with "enter" and 1 with "exit"; it's useless)
- Use external keys (es. repeat CourseID in Student; just put ID in Course and link with Student)
- Use loops in relationships
- Model classes that belong to SOFTWARE DESIGN

Let's see **some patterns**:

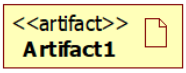


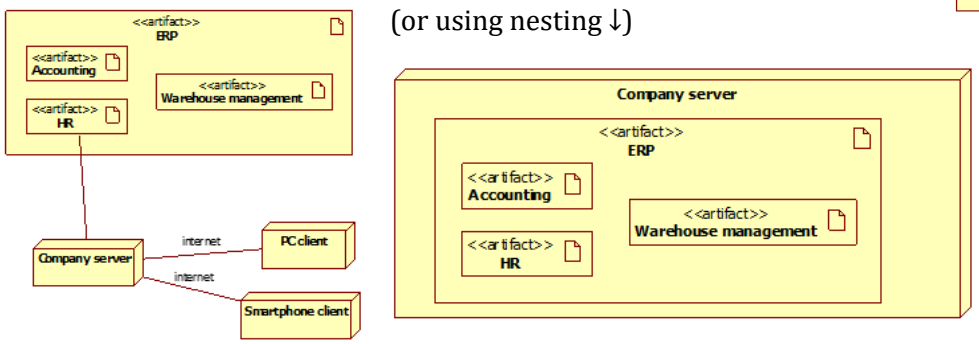


- **DEPLOYMENT Diagram** → design the hw + sw configuration of applications (same as IS):
  - Node = physical entity or software entity capable of processing; can be nested
  - Association = physical link



  - Artifact = file, library, db table (for us, mostly artifact == application); can be nested (or using nesting ↓)



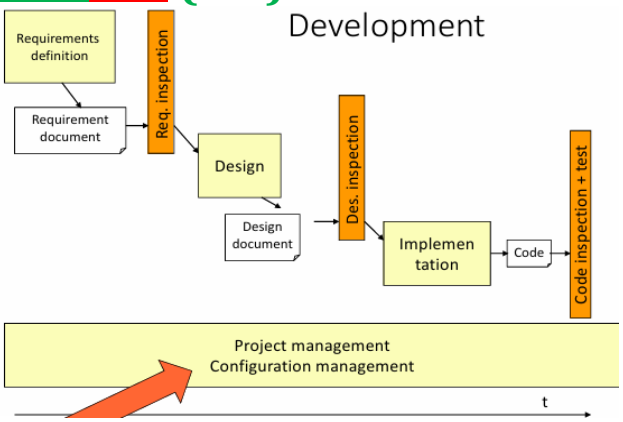


TEAMS & TEAMWORK: when considering a team, we need to consider also its single members:

- **SINGLE MEMBER** = a member (person) has 5 personality traits (Myers "Big Five"), each fit for a specific job:
  - Neuroticism
  - Openness
  - Coscientiousness
  - Extraversion
  - Agreableness
- **TEAM** = perfect size is 4-7 (more can cause communication problems, sub-groups, formalization, less participation). Characteristics are:
  - Team norms:
    - Productivity → defines what is too much and what is too little work
    - Protest → defines how and when to express conflict/contrary ideas
  - Specialization → defines if all team members can do all tasks (no specialization) or not (yes specialization)
  - Leadership → define who does what or who represents the team (better to have it emerge through democratic decisions and not at priori

⚠ *Summary* → working in teams is a must: conflict is necessary to have different ideas between different people and recognize defects in our product more easily (if everybody thinks the same, it's harder to recognize if we're failing or not). A good team performs more efficiently than single people!

4) CONFIGURATION MANAGEMENT with GIT (CM)

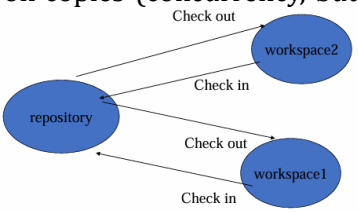


CM is useful because:

- Many persons need to work on the same documents for long time
- Documents have dependencies (usually not formalized), that are Requirement, Design item, Code fragment, Use case, Test case, Test result

CM concepts:

- **Configuration Item** (**CI**) → single unit under CM (1+ files)
- **Configuration** → set of CIs
- **Repository** → place where CIs are
- **Versioning** → capability of storing/rebuilding all past versions of a CI or configuration
- Change control → capability of granting right of reading/writing a CI to a certain user; 2 models:
  - **Lock-Modify-Unlock** = 1st user locks the CI, no other user can access it until 1st user unlocks (no concurrency, but serial work)
  - **Copy-Modify-Merge** = many users can checkout in parallel and work on copies (concurrency, but conflicts)
- Check-out → notification that a CI is being accessed by a certain user
- Check-in → notification that a CI is no longer accessed and has been modified ("***COMMIT***")



CM choices are what becomes a CI (not all documents), Lock or Copy, fequency of commits and what **CMS** (CM System) tools? A **CMS** is a software application for versioning CIs and changing control.

Taxonomy:

- Local CMS = 1 machine; repository and workspace on the same machine
- Centralized CMS = server contains repository (all CIs versions), clients contain copy
- Distributed CMS = server contains repository, client mirrors the repository locally (also all the versions) [**Git**]

Storage models:

- Deltas → for a CI the root version is stored, for next versions only differences are stored (less space, but more time required)
- Full copies → for a CI a full copy is stored (more space, but each version available in zero time) [more popular]

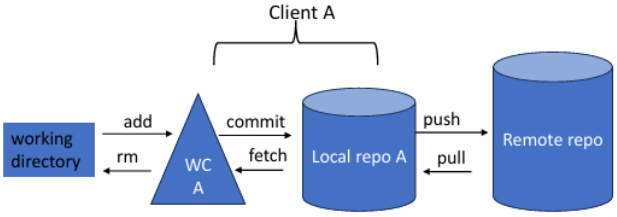
CM models:

- Differences → commit by a user changes only the version of a CI (so like "Deltas")
- Snapshot → commit by a user defines a new snapshot (all CIs in the project; if a CI didn't change version, a link to the previous one is used) [**Git**]; is identified by a unique ID (commit hash) computed at commit

GIT is a **Distributed CMS** that use **Snapshots** as CM model; has **Local operations**, is **Additive** (info only added, never deleted) and based on **Integrity** (everything is check-summed before it's stored, any change is recognized).

Basics are:

- **Working directory** (or *tree*) = files and folders
- **Working Copy** (**WC**, or *index* or *staging area*) = files and folders to be committed, but not versioned yet
- **Local repository** = files and folders versioned; is modified by commits (***commit*** = atomic operation with a log msg)
- **Remote repository** (server-side) = shared by all clients:
  - **`push`** = uploads snapshot from Local repo to Remote repo (`sync`)
  - **`pull`** = download snapshot from Remote repo to Local repo



CIs states are:

- Untracked = in the working directory, but not tracked by Git
- Staged = in WC, tracked by Git (**`-add`**)
- Committed = in Local repo, versioned (**`-commit`**)

COMMANDS:

- `git config -global user.name` *name*
- `git config -global user.email` *email*
- `git init` → initialize an empty local repo in the current folder (creating a .git directory inside it)
- `git remote add origin` *http:…* → add a new remote repository
- `git status` → show which files are in which state
- `git diff` → show as `git status` + lines changed
- `git log` → show history commits of `master`
- `git log` `name_branch` → show history commits of a branch
- `git log -all` → show history commits of all flows
- `git ls-files` → show tracked files
- `git add` *file* → add file to WC (now tracked)
- `git rm` *file* → remove file from WC (not tracked anymore) and from file system
- `git restore` *file* → restore file in file system
- `git mv` *file* → rename file (no metadata is stored in git to do this)
- `git commit` → commit changes from WC to local repo
- `git commit -a` → commit all tracked files (no need to add them to staging area, direct commit)
- `git pull` → fetch (gets changes from repo) and merge remote repo on local repo
- `git push` → upload local repo on remote repo
- `git help` *command* (or `git` *command* `-help`)
- `git branch` → show all branches
- `git branch` *name_branch* → create a new branch (ramificazione a parte del lavoro indipendente)
- `git checkout -b` *name_branch* (or `git switch -c` *name_branch*) → create a new branch and switch to it
- `git checkout` *name_branch* (or `git switch` *name_branch*) → switch to an existing branch
- `git branch -d` *name_branch* → delete a branch (merging to main)

⚠ Git stores a commit object that contains:

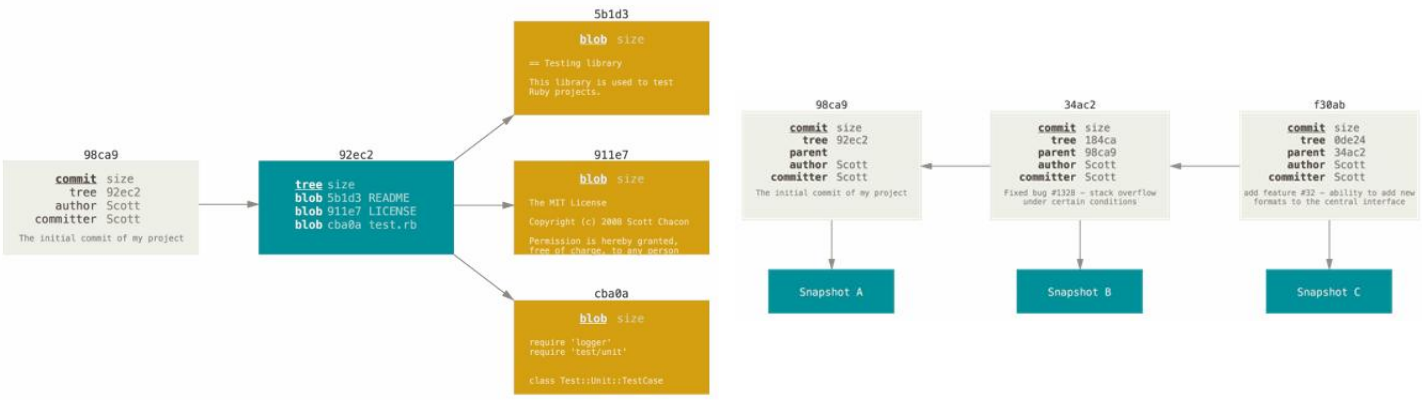
- a pointer to the snapshot of the content staged
- pointers to the commits that directly came before this commit (its parent/parents)

⚠ Staging the files:

- checksums each one
- stores that version of the file (***blobs***)
- adds that checksum to the WC (*staging area*)

Example: given a directory containing **3 files**, we stage them all and commit → now git repo contains **5 objects**:

- 1 blob for the contents of each file
- 1 tree that lists the contents of the directory
- 1 commit with the pointer to that root tree and all the commit metadata (if we do others commit, they will point to the commit before [img dx])



In Git, **HEAD** points to the current branch (the default branch to point is **master**, the *main branch*); with `checkout` and `switch` we change the branch which HEAD is pointing to.

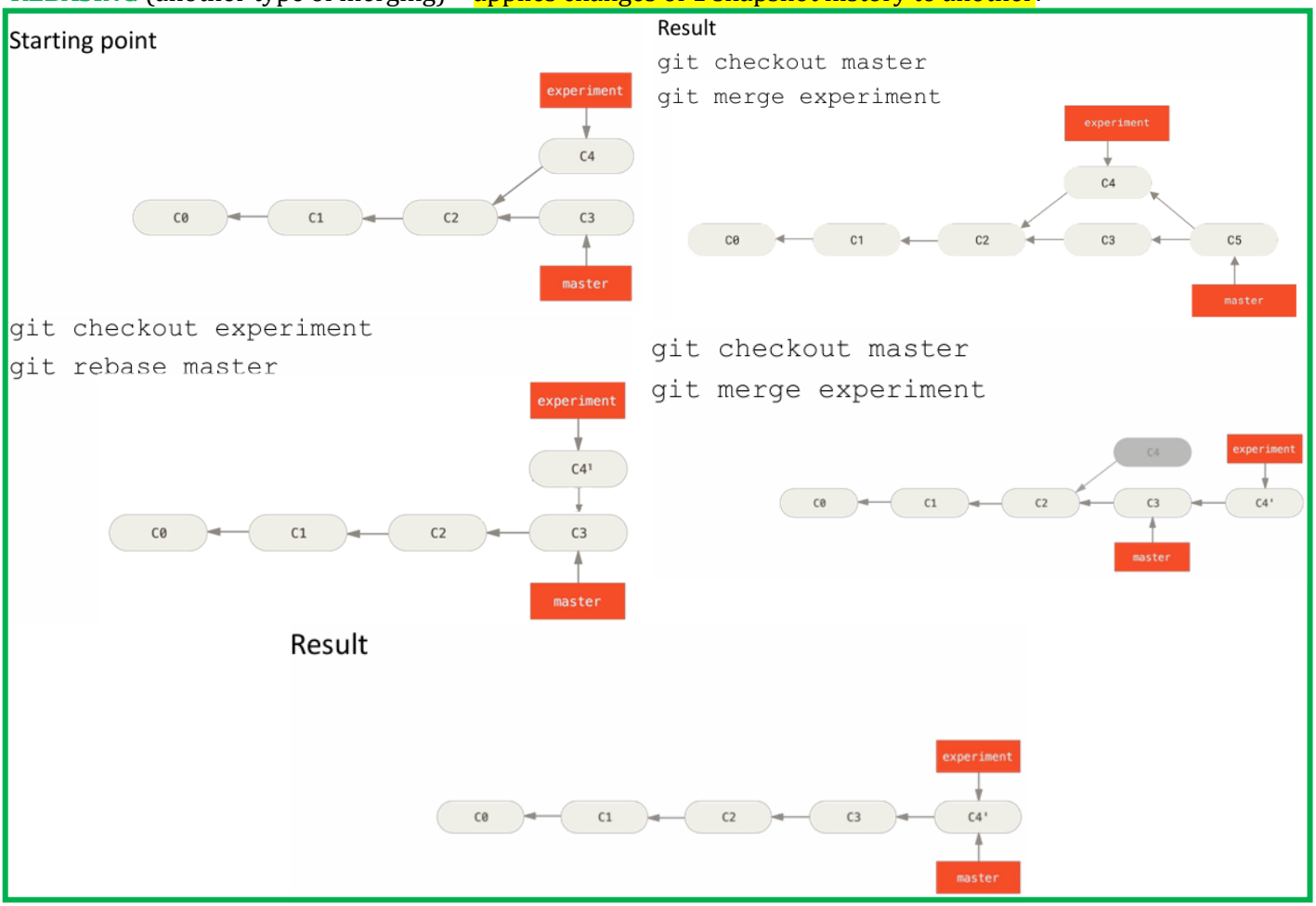
MERGING BRANCHES (3-way merge):

- **NO DIVERGENCE** (the branch and the master aren't in different branches) → `git merge` *name_branch*: merge a branch with `master` (after we can delete the branch)

- **DIVERGENCE** (branch and master are in ≠ branches) [MERGE COMMIT] → `git checkout master; git merge name_branch`

⚠ `git mergetool` → helps in managing conflicts

REBASING (another type of merging) = applies changes of 1 snapshot history to another:



With the rebase we can also edit commits: `git rebase -i HEAD~n` (`-i` = interactive mode; `~n` = number of commits we want to target). After this we can edit our commits and commit messages (we can change `pick` with **`reword`** for editing the commit message, change `pick` with **`squash`** for merging commits, …)

⚠ **Merge** vs **Rebase**:

- **Merge**: combines commit history of 2 branches, creates a new merge commit [safer] (better for integrating changes from different branches)
- **Rebase**: creates a linear commit history, rewrites the commit history of the feature branch (better for keeping a cleaner commit history)

⚠ `git cherrypick commit-hash-id` → apply a commit from one branch to another without merging the branch

⚠ FORK = copy of a repo; make changes to a project without affecting the original repo (but we can also interact with the original repo with pull/merge requests) [we will work in a branch (1 per team) and deliver through merge requests to main]

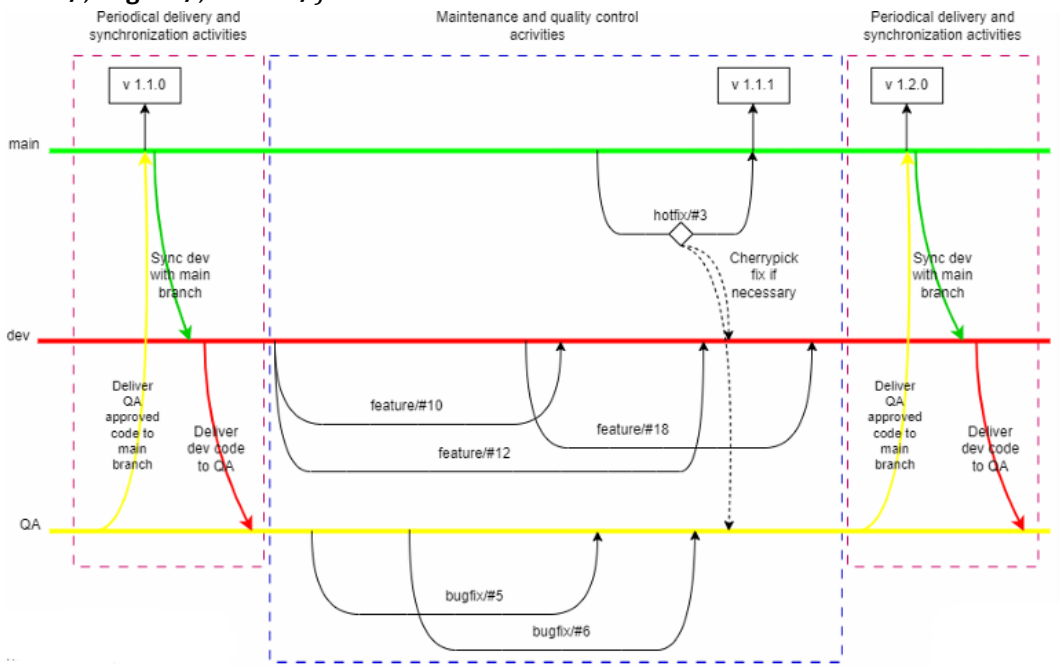
We will also use **GitFlow** (a branching model designed to structure and streamline software development [clear workflow]); common variations are Classic GitFlow (with release branches), Simplified GitFlow (no release branches) and Trunk-based development (everything on main).

Gitflow separates stable code from ongoing development (< risks), allows features to be developed in parallel on isolated branches (> efficiency), integrates testing before release (> quality, stability), facilitates collaboration (using merge requests for code review) and tracks releases using tags and versioning.

It has **Persistent Branches**:

- **`dev`** branch → new features; working branch (it has **`feature/`** branches for single features, then merged); it is periodically merged in `QA` for integration testing
- **`QA`** branch → testing and integration (regression testing + integration testing) (fixes for bugs are handled using **`bugfix/`** branch); once testing is complete, it's merged into `main` and a new release tag is created
- **`main`** branch → latest production-ready/stable code (fixes for bugs are handled using **`hotfix/`** branches)

⚠️ We prefer to use sub-branches (ephemeral branches) instead of writing code directly in the persistent branches (**`feature/`**, **`bugfix/`**, **`hotfix/`**)



5) PROJECT MANAGEMENT (PM)

First step of SE process (even before development) is **PM** (Project Management) = **initial estimation + tracking during process** (**how much** = estimation + tracking [***TCO*** (Information Systems)]; **when** = estimation + scheduling + tracking), but also it continues during the entire process:

- **INITIAL ESTIMATION** (**PLANNING**):
  - **Initial proposal** → start with planning and estimation of how long and how much (often not realistic: no initial estimation can ever be accurate [anything can happen during the development process])
    - an approximation of size estimation can be done depending on the units we will be using
    - estimation becomes more accurate as the process continue
    - it's important to have a limit to the project (goals, Parkinson's Law)
  - **Estimation by decomposition** (efficient + convenient):
    - by ACTIVITY → the work is divided in sub activities, then an amount of person hours is estimated for each sub activity
    - by PRODUCT → identify products, estimate effort per product
    - vendors usually overpromise so that the buyer is then stuck to his product during maintenance
    - inception → development of the idea and defining the requirements
    - dimension → is the product a service or does the user have full ownership of the product?
    - constraints → safety, domain… (they effect deeply the sw is developed)
- **DEVELOPMENT + TRACKING** = different decisions:
  - Project structure
  - Resource allocation

- Time planning

- **DEPLOYMENT + POST-MORTEM**:
  - Post-deployment meeting, collecting project data and understanding total effort (estimated and actual)
  - Goal → make the next development processes better

Planning and estimations are based on the concepts of resources and activities to complete-milestones:
- Milestone = key event in the project
- Deliverable = prototypes delivered to the user to test functionalities and receive feedback

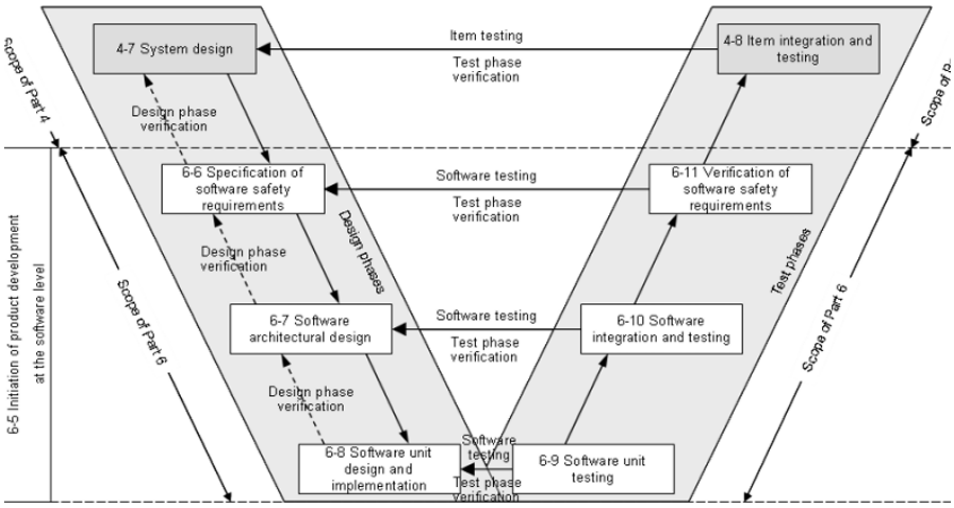
⚠ The *initial planning* (especially time) is often based on the benchmark value of other companies and products

PLANNING TECHNIQUES (Agile/Waterfall; Cost, size and quality; GANTT chart) = 2 WAYS:

1. **WATERFALL** → SEQUENTIAL activities; **focus on every part of the process**. FIXED PRICE model and STATIC APPROACH (once one activity is done, proceed to next):
   - **requirements → planning → define GANTT → development**
   - suited for bigger projects (especially with hardware/safety dependencies)
   - even if sequential, prototypes can be built directly during requirements phase → requires team skills to develop and maintain throughout the process
   - Estimations are based off effort (`cost*ph` [person hours])

   Pros: requirement document very compliant with user needs, better for large products and large teams
   Cons: changes have heavy impact, so better to avoid changes (worst are design and requirement changes). Contracts usually defined at start of development and changes in product development may require changes in contract (price, time…)

   ⚠ **V-MODEL** → similar to waterfall, focus on V&V (Validation & Verification) activities. Tests are written right after design phase



2. **INCREMENTAL (RUP)** → PARALLEL activities; **partial emphasis on documents**. Similar to waterfall, but the intgration phase is INCREMENTAL and REPEATED: every loop of integration produces a part of the system that is reviewable by the end user and can be used to fix defects in the next iteration (**early user feedback**). Mostly used for new developments, but also good for large projects and large teams

3. **AGILE** → PARALLEL activities; **no emphasis on documents**. TIME & MATERIAL model (*DYNAMIC PRICE*). DYNAMIC APPROACH = more repetitions of same phases to provide a more accurate product with less defects:
   - requirements and design are short at beginning because they are **adjusted every iteration and at every defect found** (by user or by developer)
   - be in touch with the user and adjust the product to his needs
   - when an error/inconsistency/ambiguity is detected, **go back and repeat** steps
   - solving an error could cause other errors (**REGRESSION**) → once fixed, check if no other errors are present (otherwise repeat)
   - more suited for smaller projects
   - different repetitions may take longer than expected
   - Estimations are based off effort `(cost*ph)*iterations`

AGILE **PRINCIPLES**:

- **Communication** = problems can arise from miscommunication about something important (PM)
- **Simplicity** = simplest solution is often the best one [simple design, runs all tests]
- **Feedback** = customer satisfaction + on-site customer + small releases
- **Courage** = doing the right thing, without fearing failure [software quality, refactoring]
- **Coding** & Development Standards

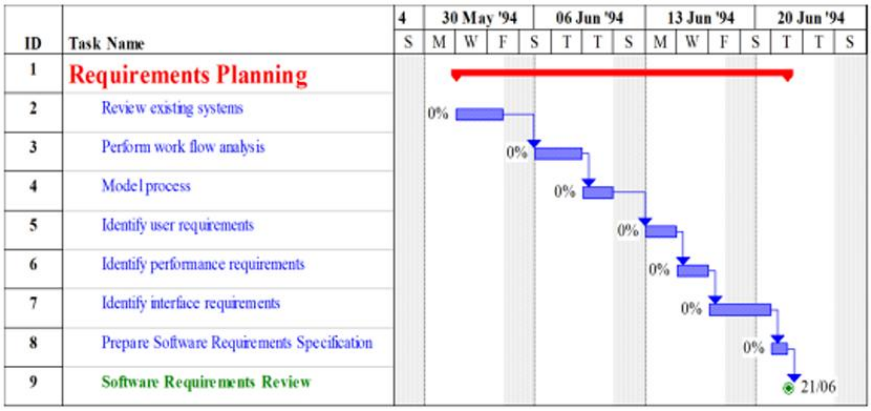
AGILE **ADDITIONAL METHODS**:

- **Scrum** → team leader + developer team + user. Team has a meeting and defines how to achieve a goal (sprint) and when to have it ready to show to the user, then proceed to show the demo to the user
- **XP** (eXtreme Programming) → write tests first, write code accordingly. Less iterations & fast delivery; no upfront design
- Pair programming → 2 people code: 1 code, 1 try to make it better (less efficiency: using double the effort over the time unit)

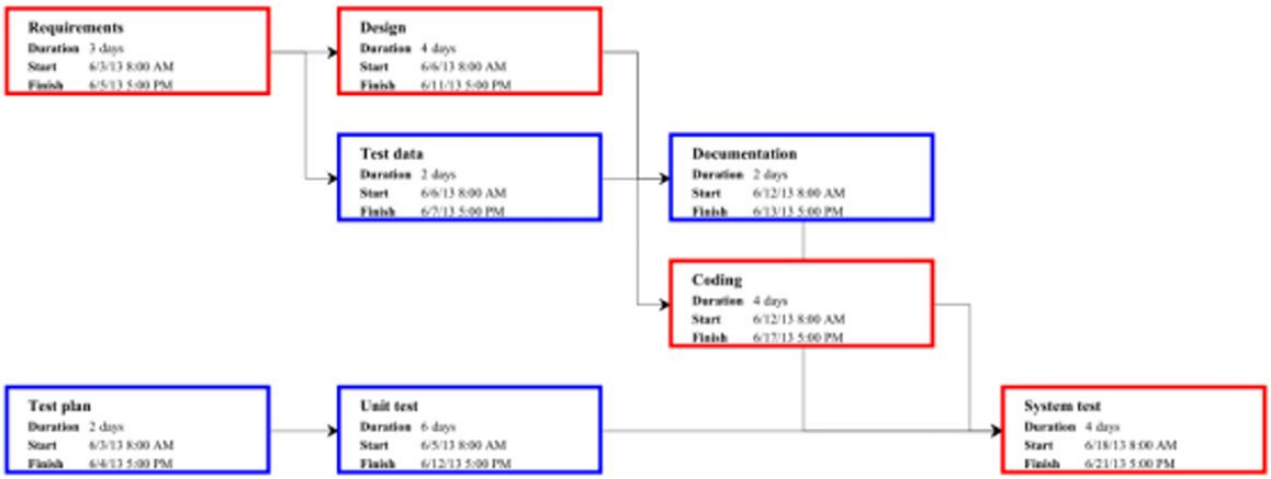
4. **CITY MODEL** → ALWAYS-ON (**max reliability**), HORIZONTAL functions (es. Twitter): the product is always-on, what teams work on is different functions that don't interfere with the main functionality of the product (**frequent changes**)

WBS (Work Breakdown Structure): **planning by decomposition/activity** → the work is divided into sub activities, then a total of person hours is estimated for each activity (part of RE):

- **GANTT CHART** = defines work division + temporal planning + tracking of what is actually done; GANTT chart + **STAFFING** → estimate how many people are going to work on certain activities



- **PERT CHART** = **directed acyclic graph**; shows different steps of development process in order (arrows indicate what needs to be executed before certain tasks)



FUNCTION POINTS = A*(External Inputs)+B*(External Output)+C*(External Inquiries)+D*(External Interface Files) + E*(Internal Logical Files); function points are adjusted by factors as **TCF** (Technical Complexity Factor encountered in the project) and **ECF** (Environmental Complexity Factor)

⚠️ **COST, TIME & QUALITY** are linked together:

a) **COST** = depends on hardware, software and staff (person hours spent) [> time spent on the process → < important is the cost of acquisition for a resource]
   - Relative (month 1, month 2…)
   - Absolute (precise date)

b) **EFFORT** = `Duration * Resources` = time taken by a resource to complete a task, measured in **ph** (person hours) with `1 person day = 7 ph`

c) **SIZE** = size of project:
   a. source code → LOC (lines of code)
   b. documents → pages
   c. tests

   Productivity = `total size/total effort` (**ph**)

TRACKING TECHNIQUES (timesheet, list of closed activities (milestones)): the goal is, while collecting project data, to track the effort spent by resources (**ph**) and size of the project → the goal is to make the **initial estimation** as accurate as possible throughout the project:
1. **Timesheet** = keep track of how many ph have been spent on a certain activity
2. **Activities closed** = list of activities completed

6) V&V = VALIDATION & VERIFICATION

The goal is PREVENTING, FINDING AND MINIMIZING DEFECTS. We have **STATIC** and **DYNAMIC analysis** → inspection, code analysis and tests. We define:
- **VALIDATION** → is it the most effective solution to the user's problem? External correctness, user based:
  - depends on stakeholders'/user's needs
  - a product can be fully functional, but it's not valid if it's not what the user wanted (even if a product functionality is simple, as long as it solves a user need, it is valid and does not need to be more complicated)
- **VERIFICATION** → is our solution implemented correctly? Internal correctness (done by engineers)

⚠️ Coupling = degree of connection of 2 modules (es. share of a variable, message passing, function call…)

- **FAILURE** = unexpected behaviors by product
- **FAULT** = malfunction by the system that causes failures
  - after release, 10% of modules cause 80% of total faults
  - earlier detected → less resources needed to fix them
  - a fix to 1 fault could cause other faults (leading to continue changes in our products) → **REGRESSION**, especially if fixing a fault later in the process:
    - Regression can be avoided by fixing the faults while using the same tests (keeps product coherent) [**Regression testing** = after fixing a defect, same tests are repeated to ensure that the new change hasn't introduced any new defects]
    - Regression is more likely to happen in case of multiple dependencies (when dependencies are based off a unit with a defect)

V&V TECHNIQUES (**find as many defects as possible**) [tests need to be performed on a variety of data, not always on the same ones] → TEST CASES should be **documented** (*formal testing*) and **automated** because, as any other activity, testing could require more resources than expected:

1) STATIC
   a. **Inspections**

i. Reading documents [while test find 1 defect at a time, reading documents can find many defects at once]
ii. Earlier inspections, less rework
iii. Done by groups of people
iv. Checklists can be used to set standards

b. **Source code analysis**
i. Compilation analysis (syntax, types, semantic) [by compilers]
ii. MISRA-C rules = guidelines for software products, created to make the C language safer
iii. Bad smells = bad programming patterns and usages (duplicate code, long methods, switch statements, unused code)
iv. Data flow analysis = debugging the value of certain variables to find anomalies

2) **DYNAMIC** (**TESTING on a product prototype** to detect differences betwwen actual and required behavior): testing is finding defects, while debugging is finding faults (errors in implementation); both can lead to spot design failures. We have ***SYSTEM TEST*** (entire system) and ***UNIT TEST*** (functions/units) [we'll see later]
A **TEST SUITE** is a set of test cases [test case = test performed under certain conditions]:

a. **ORACLE** → expected behavior of a program/function for a given case; can use the requirement document to know what to expect, but easier said than done

b. **CORRECTNESS** → correct output for all possible inputs (exhaustive testing): it's not possible in reality (infinite test cases) and no system is perfect (we have to find defects, not demonstrate there aren't)

c. **COVERAGE** → how much of the product the tests are covering ($entities\ tested/all\ entities$)

d. **Testing and Object-Oriented classes** → observability (how to read the internal values) and controllability (how to manipulate the internal values)

Test **CLASSIFICATION**:
- How to analyze each item?
  - **Unit** = individual modules
  - **Integration** = different units together, 1 test per dependency
  - **System** = all modules together, 1 test per functional requirement (FR)
- What approach to use?
  - Black Box → testing input-output without knowing internal code (Unit, Integration & System)
  - White Box → testing input-output starting from source code (< time, > precise) (Unit)
  - Reliability & Risk based (System)

Let's see this tests in details:

- **UNIT TESTING**:
  - **Black Box**:
    - **Equivalence classes:**
      - Criterion = defines an attribute
      - Predicate = specifies the value of a criterion
      - Boundaries = ranges of values of a criterion, valid and invalid, which can be passed to the function we are testing
      - Coverage requirement → at least 1 test case per partition/boundary

      ⚠ Specifying all possible criterions and all values for them (predicates) gives the best coverage. Tests are written by taking as input a combination of all possible predicates and boundaries specified; but **how to write test cases with equivalence classes + boundaries**?
      1) Define criterions and predicates properly (define valid and invalid ranges of values)
      2) Write boundaries
      3) Write tests picking values from each boundary range

    Example:

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

In this example, sex can be only **0** and **1**, so its boundaries are *0*, *1*, *[2, inf]* and *[-inf, -1]* (so we will write test cases where we consider *0*, *1*, then a random value between *[2, inf]* and one between *[-inf, -1]*)

Let's see another example (**sx: criteria, predicates, boundaries; rx: equivalence classes + tests**):

Criteria

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

Predicates

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

Boundaries

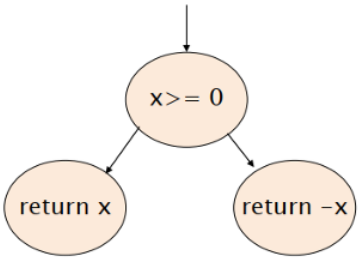
Criterion	Boundary
C1	-1,0, 250

Equivalence classes and tests

Test cases with B are test cases considering boundary elements.

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

- **White Box**:
  - **Coverage requirement** → all statements. Based on coverage: we transform our program in a control flow chart where **each node is a statement** [function becomes graph], with the goal to cover 100% of the statements [nodes] with our tests (we want 100% also in edge coverage [not only nodes]) [so ***node coverage*** ≡ ***statement coverage***]. How to do this for every part of the program?



    - ❖ **LOOP** coverage = consider 3 cases:
      - loop is not entered
      - loop is entered once
      - loop is entered more than once
    - ❖ **PATH** coverage = consider all the possible paths that a function can take:
      - this applies to loops → if a loop iterantes *n* times, I have *n* different possible paths (so it's basically impossible to compute all the *n* possible paths if n big [$n \sim$ maxint])
      - this applies also to other flow control constructs (if/else, boolean conditions…)
    - ❖ **MULTIPLE CONDITION** coverage = given a boolean condition composed of multiple boolean conditions, consider all possible combinations of these (so if I have n boolean statements and all are possible as TRUE and as FALSE, I must have **2^n tests to cover all combinations**) [if one combination is not possible, we need to specify it]

So we have to:

1) define **control flow chart** of function
2) define **test cases** for our function
3) write **TEST TABLE** and select **best combination of test cases to obtain 100% coverage**

Compute coverage for the given test cases

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

- **INTEGRATION TESTING** (“**incremental**” **testing**) → based on relationships between units, we need to test:
  - **dependencies**, using stubs (“fake unit”, simpler) simulating single functions values
  - **functions** and **units**, first tested separately and then tested by their integration (easier to detect errors and solve defects) [so, **avoid “*big-bang integration*”** [*]]

Incremental integration is an example of this technique (finds defects fast, but a lot of tests to write [effort]); 2 versions (both good, usually they are mixed; **AVOID *“big-bang testing”*** [*]):

- **top-down** = efficient for early versions of a product → can’t test single units, detect design problems
- **bottom-up** = starts from single units, doesn’t deliver high-level functionalities at first → easier detect errors and know where they are coming from

- **SYSTEM TESTING** (**requirement and scenarios coverage**; for the coverage requirement, at least 1 test case per requirement) → test of all units composing the product, considering all aspects:
  - functional/non-functional **properties**
  - **PLATFORM** (development, production) = defines where the product receives its:

        - Development
        - Production (tests are usually performed here)
        - Target [where is going to be used] (test also here, but don't use the same target as the users)
    - **PLAYER** (developer? tester? end user?) = can be done both by the user (beta testing) and developer:
        - Acceptance testing = performed by users, written by acquirer
        - Beta testing = before the product is released

  System tests are done by:
    - covering functional properties/requirements
    - covering use cases/scenarios
    - **USAGE PROFILES** = defines the most common usages of the product ("*usage patterns*"), testing the more used functions

- **RISK-BASED TESTING** → identify risks + rank them based on "probability-effect" relation
    - **Mutation testing** = used to verify test validity: keep same tests + insert errors in code → if the test catch errors, tests are valid

ECONOMICS FOR TEST AUTOMATION → **COST** of **TESTING =** (Invent + Document + Run) the test case.
Automated test cases require Invent and Document test once; but **Non-automated** tests require more resources:
- Automating a test is worth if it's executed many times
- All types of tests need to be changed if a defect is found (so testing requires many resources)

7) SOFTWARE DESIGN (Architecture & Design)

Most defects come from this phase. We define:
- **ARCHITECTURE** = high level description of the product's construction (interaction within components)
- **DESIGN** = low level description of each component:
    - Given a set of requirements, many design choices are possible
    - The choice of which design to adopt is based off the team's skills, creativity and experience

The **design process** is different based off the kind of product we are developing:
- simple software → requires only 1 software design phase
- entire system → requires system design + software design (defining how the system interacts with each sw)

⚠ Architecture and design can be described **informally** (simple tables [better for showing the user how the product behaves]) or **formally** (UML diagrams [better for developers])
⚠ **Pre-existing Patterns** can be also used (re-usable solutions to recurring problems)

PROCESS design = **ARCHITECTURE** (relations between components), **then DESIGN** for each component. Define:
- Coupling = degree of dependency between 2 components
- Cohesion = degree of consistency between functions of a component

Process desing and NF requirements, what to do?
- Performance = localize critical operations + minimize communications (when a error occurs, it's contained)
- Security & Safety = use layered architecture to protect critical assets and operations
- Availability = use redundant components to make sure the data is always available
- Maintainability = use replaceable components

⚠ Obviously, not all properties can be defined: decide necessary tradeoffs (es. deciding whether to have less layers of security but better performance)

So in the PROCESS DESIGN we have:

1) **ARCHITECTURAL PATTERNS** → real system is influenced by more than 1 type of pre-existing pattern:
    a. **Repository** = 1 central data holder or multiple ones that exchange between each other: **centralized management** (subsystems must have common data model)
    b. **Client-Server** = stand-alone servers provide data to specific services: **straightforward distribution** + **easy new integrations** (no shared data model between consumers)

c. **Abstract-machine** = system organized in different **sub-layers**, each with its **set of services**
d. **Pipes & Filters** = **data in** → **pipe-in** → **filter** → **pipe-out** → **data out** (es. **API** or **Compilers**)
e. **MVC** (Model View Controller) = model contains data, view shows data, controller handles interaction between the 2 (to keep shown data consistent with model): **separation** of responsibilities, but more **complex** (less performance)
f. **Microkernel** = **different APIs**, **1 main hw** component (need to maintain portability and consistency)
g. **Microservices** = **independent services** based on their http **APIs** (RESTFul APIs) each with its own data model (less efficient)

2) **DESIGN PATTERNS** → identifies the key aspects of a common design structure, to create a **reusable object-oriented design**. A component can be for example a class:
   a. **Choices** can be made staring from glossary and context diagram
   b. **Class patterns** = common class usages (es. defining `public` / `private`, `getters` / `setters`, `types`)
   c. **Relationships** (association, generalization, specialization…)

   Possible design patterns are as follow:
   - **CREATIONAL PATTERNS** = manage the instantiation of new objects to avoid errors and create too many low-level functions:
     - **Singleton** → useful with class that represents a concept that requires a **single instance**
     - **Abstract Factory** → useful when an app must be developed for **different OS**
   - **STRUCTURAL PATTERNS** = organize classes and objects in order to form larger structures:
     - **Adapter** → **required class** doesn't implement required interface: class is intermediate between required function class and required interface
     - **Composite** → represents a series of objects that belong to a **hierarchy**
     - **Façade** → represents a **single** object that contains **multiple** objects functionalities (without accessing the objects themselves)
   - **BEHAVIOURAL PATTERNS**:
     - **Observer** → monitors a particular object to **apply changes to other objects**
     - **Strategy** → performs a specific action (**algorithm**) between 2 or more objects

8) SOFTWARE PROCESSES

Who **executes** them, what are their **results**? Processes are divided into:
- **Primary**
  - Acquisition/Supply (acquiring resource suppliers and customers)
  - Development
  - Operation (deployment)
  - Maintenance
- **Supporting** → documentation, configuration management, V&V (quality assurance) [not directly related to the product]
- **Organisational** → project management, monitoring, training [regard developers and product utilization]

Process models = used to decide the activity organization in terms of temporal and materialistic constraints (what, who, when constraints). It's only a part of the software process (among with activities [requirement doc, design, implementation] and roles). Process models can be **created from scratch** or **based off standards**:
- Waterfall → 1 iteration; longer duration for each activity
- Agile → multiple iterations; shorter duration for each activity

An example of a process model is based off building a starting version of the product, then showing it to the user and fixing it on its requirements → this can work for small project, but for bigger projects it's a waste of resources. This means that the choice of the process model is based off different factors:
- ***Product constraints*** (national, safety, mission-critical)
- ***Size of end product*** (constraint)

Processes reuse = using multiple times open/closed source components because:

- ➢ easily **available** + **higher quality** with **low cost**
- ➢ but **not owned by developers** (no control over their evolution)
- ➢ **reusing components** is a good option during both the requirements and design phase (also considering constraints that follow these components' usage)

MAINTENANCE → based off the concept of **change** (defects + modifications to functions + new functions). **Every change requires the product to maintain its original architecture over time and its market sustainability**:

- for this reason, only a part of changes is actually implemented, even if corrective
- changes can be **required** or **created** by either users (who signal faults / failures), by developers or automatic failure detection systems
- 3 types of changes during maintenance: corrective, enhancement, evolutive

DevOps = is about the **relation** between development team and operation team: the development team knows the product layout, information, requirements etc… The goal is remove communication and knowledge barriers.

Synch & Stabilize (**S&S**) = context where there are many defects that need to be fixed, but can't be fixed at the moment: so, the procedure is **split in small teams** where each focus on a certain amount of tasks and synchronizes with the others frequently → this creates an iterative process where product is **fixed incrementally in a stable way** (all teams converge in a single version of the application every iteration). How is done? 3 phases:

1) PLANNING → product managers **define goals, priorities and team formations** [manager + devs + testers]
2) DEVELOPMENT → **split in groups**: 1 subproject is analyzed by more teams; each team produces code, debugs it and tests it
3) STABILIZATION → when a subproject is finished, the code/product developed by each team is **joined in a single stable component**

⚠ **S&S solves the problem of DevOps** by having multiple teams on the same subproject (product component) and interacting with each other consistently during the stabilization part

PRODUCT QUALITIES → how to pick process model? Different factors:

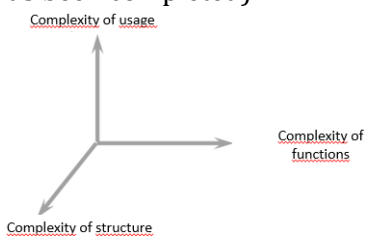
- **Criticality + Domain** = if product has certain constraints, **sequential** is better
- **Size** = large project size is better with **waterfall/S&S** process (to avoid inconsistent product versions and too many resource-wasting repetitions)
- **Technical debt** = the effort we don't spend today, has to be spent later + interest → developing a starting version of the product fast might require more time to fix later on, so **waterfall/sequential** is better
- **Relationship with developers** = if developers are external, might need a process model that bases itself more on documenting the product requirements (**sequential**)

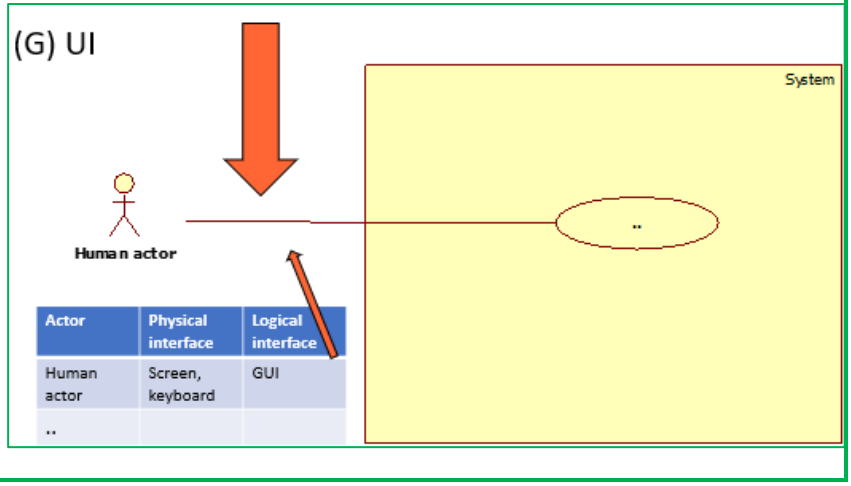
⚠ A process model can also be picked based on **what we want to guarantee to the end user**:

- ➢ **Reliability** = a reliable product is easier to develop in **waterfall/sequential** process
- ➢ **Market driven** = a product that is able to adapt to the market is easier to implement by **Agile methods** and overall methods based off **multiple iterations**

→ USER INTERFACE (UI)

When human actors are involved, designing the **User Interface** (**UI**; **GUI** = **Graphical UI**) is a key design choice (assuming that RE activity has been completed):





UCD = User Centered Design → process for mass market products:

- Prototypes **low fidelity** (**low-fi**) = sketches, paper, post-it… [analyzed by ergonomy experts]
- Prototypes **high fidelity** (**hi-fi**) = mockup on computer (ppt, figma, …) [tested by real users]

Techniques of **Feedback**:

- **Etnography** = users perform normal daily activities using the prototype (observers don't interfere)
- **Interviews** = users are interviewed after using the prototype (open/closed questions)
- **Focus Groups** = users discuss the product in a group after testing it (guided by moderator)

9) TYPESCRIPT, ORM & TYPEORM

TypeScript is a strongly typed superset of JavaScript that adds optional **static typing**; is **transpiled** to JavaScript through ***tsc*** (TypeScript compiler), making it executable in any JavaScript runtime (es. browser, Node.js). It enables developers to express constraints on data, improving correctness and maintainability.

```
// TypeScript file.ts (or file.tsx)

const sum = (a: number, b: number): number => a + b;

// Transpiled to JavaScript file.js (or file.jsx)

const sum = (a, b) => a + b;
```

⚠️ All JavaScript code is valid TypeScript code, **but not the reverse;** `const result = 1 + true` // in JS result = 2, while in TypeScript error; instead, `const result = '1' + 1` // both in JS and TS is string concatenation

TypeScript need to be installed on project (not globally) with **`npm i typescript`**. To generate a **`tsconfig.json`** file prepopulated with recommended settings, we can use **`tsc -init`**; when we write **`tsc`** command, TypeScript will compile using the configuration specified in the nearest `tsconfig.json` file.

In TS we have **7 primitive types** (so basic types, no objects and no associated methods): **`string`**, **`number`**, **`boolean`**, **`null`**, **`undefined`**, **`symbol`**, **`bigint`** (so `int > Number.MAX_SAFE_INTEGER`); all these types are **immutable** (their values can't be modified once assigned) [`let var1: number = 12;`].

There's **Type Inference** (`let x: string = 'x;` // Explicit typing → `let y = 'x;` // Type Inference to string), also in return values of functions (es. `function add(a: number, b: number): number { return a+b; }` is equal to `function add(a: number, b: number) { return a+b; }`).

A **Literal Type** is a single element set from a collective type; they are **`number`**, **`string`**, **`boolean`** and they are only used by the **`tsc`** (es. `type O = 'a' | 'b' | 'c';`). **Enum**s are compiled into actual **JS objects**. There also Union and Intersection; there are **Generics** (used to write code for every type) and there is the fictious type **`any`** to write variables without specifying the type at coding time.

3 examples of **`@Watch`**, **`@Serializable`** and **`@Required`**:

```
function Watch(target: any, key: string) {
  let value = target[key];
  const getter = () => value;
  const setter = (newVal: any) => {
    console.log(`Property ${key} changed to: ${newVal}`);
    value = newVal;
  };
  Object.defineProperty(target, key, {
    get: getter,
    set: setter,
  });
}

class Person {
  @Watch
  name: string = "";
}
```

```
function Serializable(target: any) {
  target.prototype.toJSON = function () {
    return JSON.stringify(this);
  };
}

@Serializable
class Book {
  constructor(public title: string, public author: string) {}
}

const b = new Book("1984", "Orwell");
console.log(b.toJSON());
```

```
function Required(target: any, key: string) {
  if (!target.__required) target.__required = [];
  target.__required.push(key);
}

function Validate(instance: any) {
  const required = instance.__required || [];
  for (const key of required) {
    if (!instance[key]) throw new Error(`${key} is required`);
  }
}

class User {
  @Required
  username!: string;

  constructor(data: any) {
    Object.assign(this, data);
    Validate(this);
  }
}
```

TypeScript also supports **object-oriented class-based programming**. As Java, we have **`public`**, **`private`** (only in class) and **`protected`** (class & subclasses) access modifiers. When we define a class, there's a default constructor, but we can define a new one (the default will be deleted).

⚠️ We can also have **`static`** variables/methods (it mantains the value through objects [remember ASE]).

```
interface Person {
  name: string;
  surname: string;
  vat_code?: string; // optional property (this notation can also be used with parameters)
}
class Student {
  private email : string;
  constructor(public name, public surname) {
      this.email = name + "." + surname + "@email.it";
  }
}
var user: Person = new Student("John", "Doe");
```

⚠️ In a class a method can be declared without implementation through the word **`abstract`**; in interface, instead, all methods are abstract by definition (so the word abstract is not required) [no values and no implementations]

In an interface we can also have a **`readonly`** attribute/property (`readonly ssn: number`), so that its value can't be reassigned after initialization.

Examples of **interfaces for array and dictionary types**:

```
interface NumberArray {
  [index: number]: number;
}
let arr: NumberArray = [10, 20];

interface Dictionary {
  [key: string]: string;
}
let d: Dictionary = { TS: "TypeScript", JS: "JavaScript" };
```

⚠️ Interfaces can be extended through other interfaces (`interface IEmployee` **`extends`** `IPerson`); a class can implement an interface (`class Employee` **`implements`** `IEmployee`)

TypeScript supports **Function Overloading** (we can have 2 function signatures with the same name, but different parameters) [as Java]. TypeScript support both **Polymorphism** (with inheritance and interfaces, as we have seen before) but also structural polymorphism (without inheritance, but with other constructs as **union** → `type Media = Book | Movie;`).

TypeScript supports also the creation of **Module**:

```
module MathModule {
  export class Expression {
      sum(term1: number, term2: number) {
          return term1 + term2;
      }
  }
}
var expr = new MathModule.Expression();
var res = expr.sum(1,2);
```

⚠️ It is not possible to check TypeScript types at runtime (the types are erased after compilation); to do this, we can use the "**tagged union**": the property **`kind`** is used to distinguish at runtime between objects in JS; but also throught **`instanceof`** operator.

In TS there are also **Promise**s (represents the result of an asynchronous operation; can be in 1 of 3 states: **pending**, **fulfilled** and **rejected**), but also the **async**/**await** syntax that we have seen in Applicazioni Web 1:

```
async function getUser(): Promise<void> {
  const response: Response = await fetch("/api/user");
  const data: { name: string } = await response.json();
  console.log(data.name);
}
```

We can also use **Promise.all()** to run multiple async tasks in parallel. Let's an example of safe error handling with **instanceof** error:

```
async function loadData(): Promise<void> {
  try {
    await fetchData();
  } catch (err) {
    if (err instanceof Error) {
      console.error(err.message);  //Safe access
    } else {
      console.error("Unexpected error", err);
    }
  }
}
```

We define **Persistence** (we want that program's data to outlive the scope of current process → **persistent** data = stored in disks, DB or ROMs; **in-memory** data = stored in RAM [volatile]); **Direct Data Access** (interacting with a DB relying only on a language native database driver to connect to the DB [*SQL*]).

The **Object Relational Mismatch** is mismatch between relational data model and object-oriented paradigm:

- **Granularity** → object models often have more classes than tables in the DB
- **Subtypes** (inheritance) → relational model doesn't have inheritance
- **Identity** → relational model has primary key equality, object model has object equality and value equality
- **Associations** → bidirectional in relational, unidirectional in object (the ORM remove this difference; we have also relationships cascading, that allows propagation of operations [`persist`, `update`, `delete`] from a parent entity to its related entities)
- **Data navigation** → in object from 1 object to the next object (ORM has lazy loading [only loaded when accessed] + eager loading [loaded with the entity])

An **ORM** (Object-Relational Mapping) is a technique for **managing database interactions using object-oriented programming**, converting data between a relational database and the memory. SQL queries can be automatically generated based on object models. It abstracts connection handling reducing manual configuration, but it has less control over query optimization compared to optimized SQL

⚠ When basic SQL querying aren't enough, we can also have advanced techniques in ORM

ORM Entity is a structural component that represents a persistent object mapped to a DB table; each entity instance corresponds to a row in the table. An entity defines fields (mapped to table columns) and relationships with other entities, but must have at least 1 primary key to ensure uniqueness. There's **conversion** between object model types and relational model types (es. **`String`** → **`VARCHAR`**). For each entity field we can define the constraints typical of a relational db schema (es. `foreign key`, `nullable`, `unique`, `default values`…).

There are **schema creation** policies:

- **`Create`** = generates the schema if doesn't exist
- **`Drop-and-Create`** = deletes and recreates the schema on each run
- **`Update`** = updates the schema to match entity changes
- **`Validate`** = checks if the schema matches the entity definitions but doesn't modify it
- **`None`**

We define **Session** (or persistence context) an intermediary temporary environment where an ORM tracks and manages entities during their lifecycle, ensuring that objects remain synchronized with his corresponding records in the DB. Its **scope** and **lifespan** must be carefully managed to avoid performance issues. Operations:

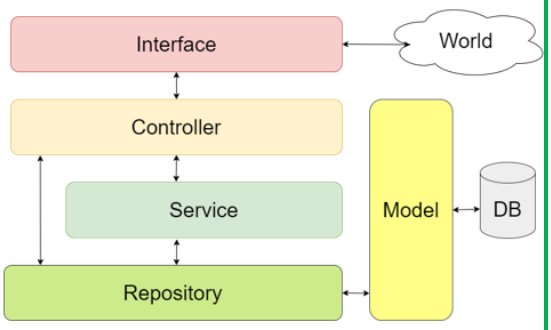
- Fetching entities = when an entity is retrieved using an ORM query, it is **automatically added to the session**, its state is tracked and modifications will be synchronized with the DB when session is flushed/committed
- Detaching entities = removing it from the session (not from the DB), keeping it in memory (no more tracking)
- Removing entities = entity marked for **deletion** and will be removed from the **DB**

We have "short-lived" (new session for each operation, that close immediately after) and "long-lived" (sessions remains open for a longer period, entities can be reused without repeated database queries) sessions.

We define **Transaction** a sequence of 1 or more DB operations executed as a single unit of work; it follows the **ACID** properties (no partial updates). We can also have multiple transactions in concurrency; but we must follow the "isolation level" required (how the transactions interact between them).

Repository Pattern = provides an abstraction layer between the application's logic and the DB. Using repository for **CRUD** operations ensures a consistent API across the application. We have basic and custom queries

Designing a Layered ORM-Based App → organizes an application into multiple, distinct levels, each with a specific responsibility; each layer communicates only with adjacent layers, maintaining a structured flow of data



TypeORM is an Object-Relational Mapping (**ORM**) library for TypeScript and JavaScript; is **fully written in TS**. Uses decorators to define entities and relationships and supports multiple DB engines. Includes automatic migrations and schema synchronization; allows lazy and eager loading for relations. A single istance of **`DataSource`** is the core object that manages the connection to the DB. We use **SQLite** because is fully supported by TypeORM (**`db.sqlite`**). It's in-memory (runs entirely in RAM, losing all data when app stops). We have:

- **`@Entity`** → defines a table
- **`@Column`** → defines table fields (`nullable`, `unique`, `default`)
- **`@PrimaryColumn`** → defines primary key field of an entity
- **`@RelationshipDecorator(() => TargetEntity, (target) => target.property, options)`** → `TargetEntity` is the class of the related entity, `target.property` is the property in the related entity that references this entity (only needed in bidirectional relations), `options` defines how the relationships behave
- **`Cascade Options (cascade: true | ["insert", "update", "remove"])`**
- **`Delete Behavior (onDelete)`** → we can have **`CASCADE`**, **`SET NULL`**, **`RESTRICT`**

We can have both **`eager`** and **`lazy`** as loading strategy. About multiplicities:

- **`@OneToOne`**
- **`@OneToMany`**
- **`@ManyToOne`**
- **`@ManyToMany`**

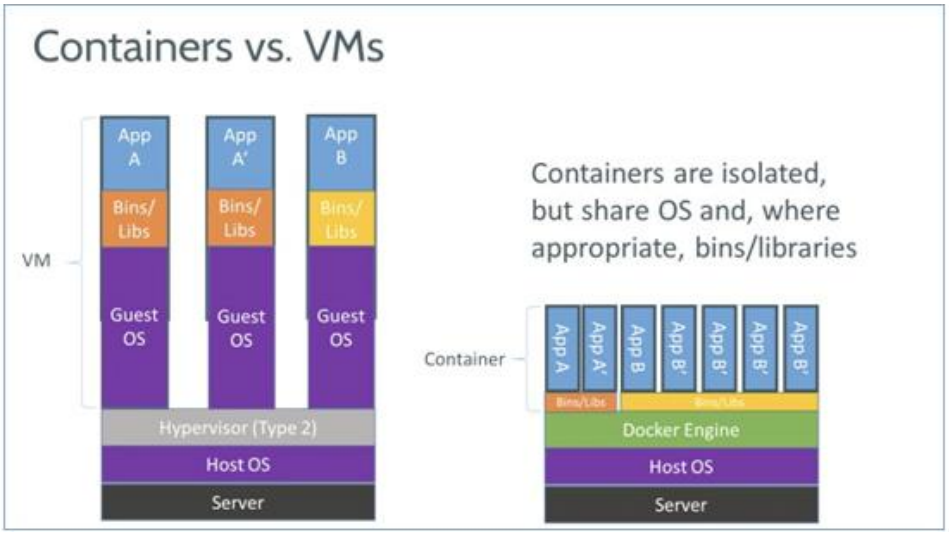
⚠️ TypeORM does not support eager loading on both sides of a bidirectional relationship

TypeORM implement the Repository pattern for each entity; can be retrieved via **`DataSource.getRepository(Entity)`** with methods `find`, `findOne`, `save`, `remove`, `delete`… **We can work entirely in TypeScript, without writing any SQL**; **then Never use the application's real database during testing**.

10) DOCKER & Gitlab-CI/CD

Docker is a software platform that simplifies building, running, managing and distributing apps. It virtualises the OS of the computer on which it is installed, enabling the efficient deployment of containerised applications. It's useful if we have a project with multiple web apps on a single server and different frameworks with different dependencies for each app.

Docker Containers are logical entities created by Docker Host (virtual copy of process table, network interfaces and file system mount points); the OS isn't in container because the kernel of host OS is shared across all containers:

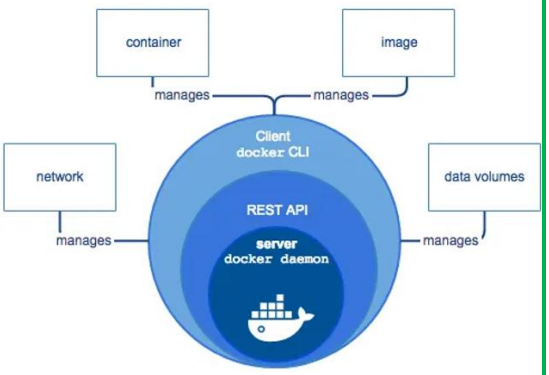


We have:

- Bridge Network (default) → containers on the same network can communicate through container names
- Host Network → container shares the host's network stack (> performance, < isolation)
- Overlay Network → containers communicate across docker hosts

Docker Engine manages docker's operations and has 3 main components:

- Server = operates a daemon named "dockerd" (which handle creating and managing docker images, containers, networks and volumes)
- REST API = enables apps to interact with Server
- Client = command-line interface that lets users communicate with docker



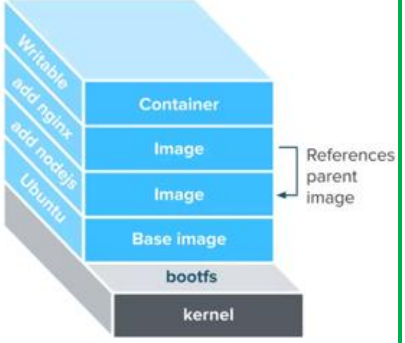
Docker Images are templates containing the app and all the dependencies require to run the app on docker [images are static (as OS images) = snapshots of app and its environment, containers are dynamic = running instances of images]

⚠ Docker Hub = online repository for docker images; GHCR (Github Container Registry) = enables Github users to host and manage Docker images (allows users to store their container images in a central location)

Building docker images = each instruction in the building file of the docker image creates a layer (these layers are read-only [**R/O**]); then at the end the layers are stacked together and form an image ("*union file system*").

Creating docker containers = docker containers are made by creating a read-write (**R/W**) layer on top of docker image ("container layer"); it's based on COW (Copy-On-Write → copy before modify) and contains all changes to the containers.

⚠ **R/O** layers (*image layers*) can be shared between containers starting from the same image; **R/W** (*container layers*) are obviously unique per container



How to store data? (should be isolated from containers):

- Volumes = stored in the host filesystem and managed by docker
- Bind mounts = stored in the host filesystem and need a path
- Tmpfs mounts = available only on Linux and not persisted on the host filesystem

Dockerfile = **text file containing instructions to build a docker image** (base image + app code + dependencies + commands to be executed to set up the environment and run the app); it's used by the docker engine to automate the building of the image. Es (sx) and Es of **multi-stage optimized builds** (dx):

```
FROM node:latest
WORKDIR /app
COPY . .
RUN npm install
EXPOSE 3001
CMD ["npm", "start"]
```

```
# Build stage
FROM node:18 AS build
WORKDIR /app
COPY . .
RUN npm install && npm run build

# Production stage
FROM node:18-slim
WORKDIR /app
COPY --from=build /app/dist ./dist
CMD ["node", "dist/server.js"]

```

Create a docker container → **`docker build -t my-app-image`**

Run the node.js app in the container, exposing it on port 3000 and mounting the local folder to the container under the /app directory → **`docker run -d -p 3000:3000 -v /path/to/local/folder:/app my-app-image`**

TAG = name + version reference for an image (required to push a registry) (es. **`docker build -t my-app:1.0.0`**)

Pushing a docker image (! you must have write access to target repository):
1) tag the image with full registry path
2) log in to the registry (es. **`docker login ghcr.io`**)
3) push the image (es. **`docker push ghcr.io/myusername/my-app:1.0`**)

Pulling a docker image (es. **`docker pull ghcr.io/myusername/my-app:1.0`**) [automatically done when docker run an image that doesn't exist locally (es. **`docker run -d mysql`**)]

Docker Compose = **define and manage multi-container docker apps** (they can be defined in **`yaml`** and **`JSON`**); the **`docker-compose.yml`** file is a configuration file that defines: Services (container running the app + image used + ports to expose + environment variables), Networks (allow containers to communicate [see above]) and Volumes (persistent data shared between containers [see above]) [when run docker compose up, `docker-compose.override.yml` is used]

⚠ **YAML** file is a human-readable data serialization format used for configuration files and structured data (differently from JSON it supports comments) [is a mix between JSON and MD] [sensitive to indentation]

Environment variables in Dockerfile (sx) vs Docker Compose (dx):

```
FROM node:latest

ENV SERVER_PORT=3001
ENV APP_ENV=production

WORKDIR /app
COPY . .
RUN npm install
EXPOSE ${SERVER_PORT}
CMD ["npm", "start"]
```

- `ENV` sets default values used during image build and container runtime
- `${SERVER_PORT}` is used in `EXPOSE`, but can be overridden in Compose
- You can use these variables in your application code as well (e.g., `process.env.SERVER_PORT` in Node.js)

Pass values to containers:

```
services:
  frontend:
    image: app-frontend
    container_name: app-fe
    environment:
      SERVER_HOST: localhost
      SERVER_PORT: 5000
      APP_PORT: 5173
    ports:
      - "5173:5173"
```

- Compose-defined variables override those in the Dockerfile
- Compose-defined variables are injected at runtime and readable from code
- Useful for environment-specific customization
- Anchors help maintain consistency across services

⚠ Environment variables expose sensitive data, so for sensitive data we can use **Docker Secrets**

We can use Docker to test the app in a secure environment without needing any dependency or risking compatibility issues; we can launch docker with **`docker compose up -d`** and stop it with **`docker compose stop`**.

GITLAB CI/CD → **CI** = continuous integration of code changes in a shared repository (stops at **TEST**); **CD** = continuous delivery (+ manual approval before deploy; stops at **RELEASE**)/deployment (automated deploy; stops at **DEPLOY** = end). We have artifacts (file generated during pipeline) and cache (reusable dependencies; > speed)
⚠ CI pipelines (**`.gitlab-ci.yml`** file) are executed by **`git push`** and in our project run only in the **`main`** branch