POLITECNICO
DI TORINO

Software Engineering

Course Handbook 2017

Table of Contents

I INTRODUCTION

1.1 Motivation: Importance, diffusion and complexity of software

These times, software has a very prominent role inside economy and society. The economies of all the developed nations are highly dependent on software: financial operations, manufactoring, distribution and entertainment are, at least to some extent, aided by software and automated.

Therefore, ICT market has experienced an enormous growth and is still growing in recent years, as shown by the followign graphs: figure 1 shows the continuously ascending revenue (in millions of dollars) of the ICT market; figure 2 highlights the fact that revenues from ICT market have actually overtaken the ones of the automotive market.

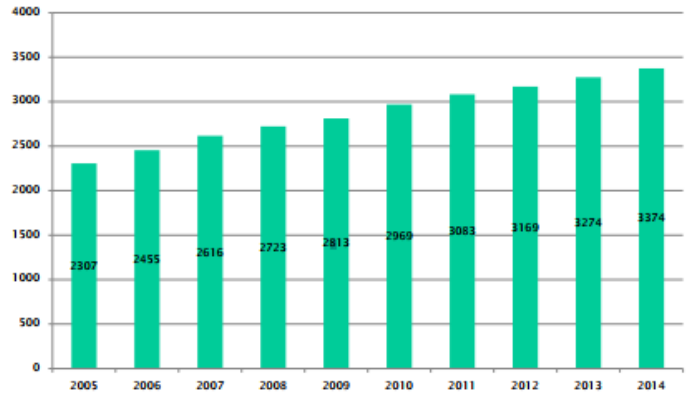

Figure 1 - ICT market revenues ($ millions)

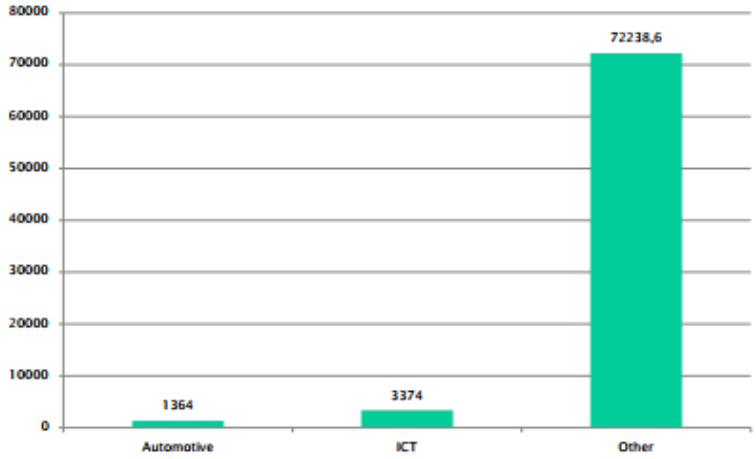

Figure 2 - ICT market revenues compared with automotive domain

A correlation has also been proved between the wealth of countries and the diffusion of ICT in them. A positive correlation is identified between the usage of technology and the per capita GDP (Gross Domestic

Product); ICT has also revolutionized the world of work, with the creation of new kinds of employment and a significant improvement of productivity.

In a 2015 report about ICT made by WEF (World Economic Forum) it is stated that *"as a general-purpose technology, ICTs hold the potential of transforming economies and societies. They can help address some of the most pressing issues of our time and support inclusive growth".*

WEF ICT report gives a classification of countries according to a Network Readiness Index (NRI) which is a key indicator about how countries are performing in the digital world. The NRI index is based on a set of key factors about the readiness of the network infrastructure, the usage of networks (by individuals, companies and governments), and the social and economic impacts of ICT in countries. Figure 3 shows the top 15 countries by NRI, in 2015.

As shown in figure 4, there is an evident positive correlation between the GNI (Gross National Income) per capita, and the NRI.

Rank	Country/Economy	Value	2014 rank (out of 148)	Income level*	Group†
1	Singapore	6.0	2	HI	ADV
2	Finland	6.0	1	HI-OECD	ADV
3	Sweden	5.8	3	HI-OECD	ADV
4	Netherlands	5.8	4	HI-OECD	ADV
5	Norway	5.8	5	HI-OECD	ADV
6	Switzerland	5.7	6	HI-OECD	ADV
7	United States	5.6	7	HI-OECD	ADV
8	United Kingdom	5.6	9	HI-OECD	ADV
9	Luxembourg	5.6	11	HI-OECD	ADV
10	Japan	5.6	16	HI-OECD	ADV
11	Canada	5.5	17	HI-OECD	ADV
12	Korea, Rep.	5.5	10	HI-OECD	ADV
13	Germany	5.5	12	HI-OECD	ADV
14	Hong Kong SAR	5.5	8	HI	ADV
15	Denmark	5.5	13	HI-OECD	ADV

Figure 3 - The Networked Readiness Index 2015

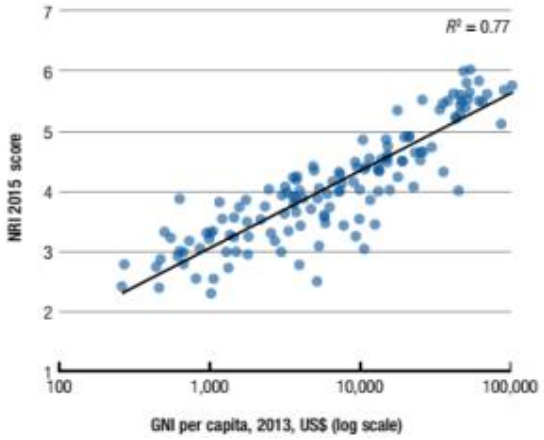

Figure 4 - Networked Readiness and Income

WEF also defines a Global Competitivess Index (GCI), which measures the quality level of institutions, politics and other factors determining economic prosperity on mid-long term. GCI is also influenced by innovation

and business sophistication, which are identified as keys for innovation-driven economies. Figure 5 shows the top 15 countries by GCI, in 2015.

As shown in figure 6, there is an evident positive correlation between the GCI and the level of income of countries: this is another prove of the positive influence that the massive dissemination of ICT has on economies.

	GCI 2014–2015			
Country/Economy	Rank (out of 144)	Score (1–7)	Rank among 2013–2014 economies*	GCI 2013–2014 rank (out of 148)†
Switzerland	1	5.70	1	1
Singapore	2	5.65	2	2
United States	3	5.54	3	5
Finland	4	5.50	4	3
Germany	5	5.49	5	4
Japan	6	5.47	6	9
Hong Kong SAR	7	5.46	7	7
Netherlands	8	5.45	8	8
United Kingdom	9	5.41	9	10
Sweden	10	5.41	10	6
Norway	11	5.35	11	11
United Arab Emirates	12	5.33	12	19
Denmark	13	5.29	13	15
Taiwan, China	14	5.25	14	12
Canada	15	5.24	15	14

Figure 5 - Global Competitiveness Report 2014-2015

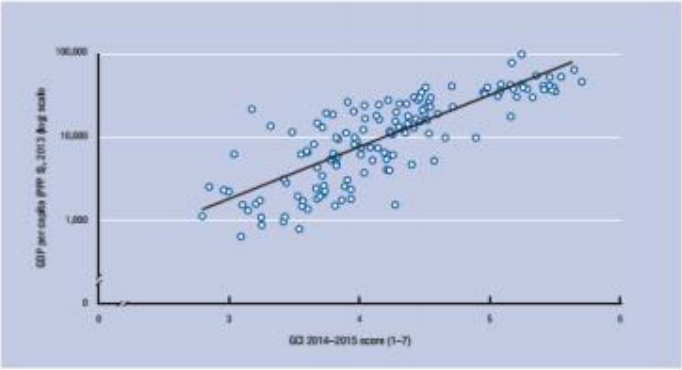

Figure 6 - Relationship between the GCI and level of income for 143 economies

Software has become highly pervasive, with a great number of high-complexity applications and services being used daily by people with their personal devices, and with a very high level of interconnection between systems. Figure 3 shows the growth in demand of new-generation devices (tablets and smartphones) by end users during last years. It is worth mentioning that the yearly shipping of smartphones has largely overtaken the demand for desktop computers. Software systems have a very high amount of variability: there are many types of software systems, and each of them must be developed, deployed and managed in different ways.

Software, in addition to being more and more widespread, also has an increased complexity. Software costs dominate the costs of the computer systems, and also high costs of maintenance are needed. Today complex systems have to be built and delivered quickly, and reliability of software is mandatory when sensible

information and critical operations are aided (or wholly performed) by computers. Most typical software problems are an excessive cost (up to a factor of 10 with respect to expectations), a late delivery (up to a factor of 2 with respect to deadlines), and a lack in compliance with user expectations.

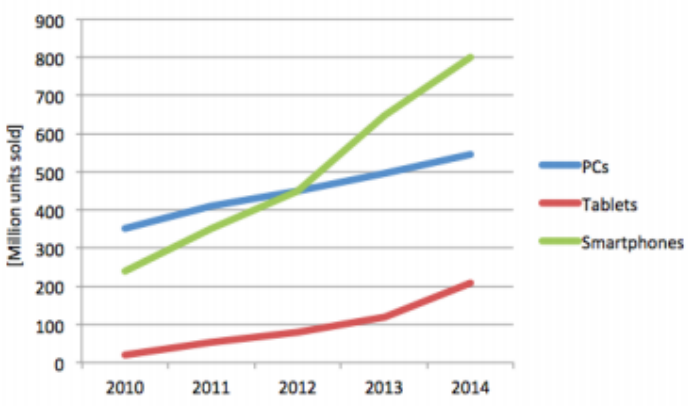

Figure 7 - Demand of technologic devices by end users

Software engineering coves all the aspects of software production, from development to management of software and production of theories that can aid software development. The discipline of software engineering allow to produce more trustable software, and the adoption of software engineering theories can lower significantly the cost for development, testing, and maintenance of software systems on which people, societies and economies rely.

1.2 Definitions and Concepts

This section defines a set of keywords that are crucial for the discipline of software engineering,

1.2.1. Definition and types of software

Software is a collection of computer programs, procedures, rules, associated documentation and data. Software development covers all the elements related to software, and hence is more than merely the development of software programs. Software incorporates documents that describe various views of the same software elements for various stakeholders (for instance, users or developers).

According to “The Mythical Man Month” (Brooks, 1975), for solving a given problem, producing software is approximately 10 times more expensive to produce than a simple program. On average, in software development 10 to 50 LoC are produced per person day, and the number can be lowered down to 7 LoC in critical systems.

Many different types of software exist, and the techniques for developing and maintain them are very different. Some possible software types are:

- *Stand Alone applications*: software that can be run locally on a device, and that does not require network connections to perform their functionalities (e.g., Notepad);

- *Embedded software:* software that is installed on hardware devices, to control them (e.g., ABS, digital cameras, mobile phones);
- *Process support software:* software that supports production processes (industrial automation) or the execution of business processes (management automation).

The eterogeneity of software and the emergence of new software types goes along with a massive **diffusion** of software. In 1945-1980 software was confined to scientific communities, military forces, banks and large private organizations. From 1980 onwards software became global, with a huge impact on everyday's life of common people.

1.2.2 Criticality and Complexity of software

Software can be classified according to a set of parameters; **criticality** identifies the relevance of software inside a company, and the possible damage (to people or things) resulting from software malfunctions. Different levels of criticality can be identified for software:

- Safety Critical (or Life Critical) software: a software which failure or malfunction may result on death or serious injury to people, loss or severe damage to things, or environmental harm (for instance, aerospace, military, medical software);
- Mission Critical software: software that are essential for a company or project, and that manage important data, equipment of other resources (for instance, banking software, logistics, industrial production control);
- Non Critical software: other kinds of software which malfunctionings are not harmful neither for people wellness or business performance (for instance, entertainment and games).

As a human artifact, software can be characterized by its **complexity**, defined as the number of "Parts and interactions among parts" (H. Simon, The sciences of the artificial, 1969). Software systems are probably the most complex human artifacts: if the line of code is identified as their atomic part, some systems can be composed by several millions of them, way more than an IKEA table (5-10 components), a bycicle (20-100 components), a car (about 30.000 components), an airplane (about 100.000 components). Table 1 reports some examples of software complexity.

Software system	Number of Lines of Code
Cell phone, printer driver	1 million
Linux 3.2 (2012)	15 million
Windows 7	50 million
F-22 Raptor (Us Air Force jet fighter)	1.7 million
F-35 Joint strie fighter (onboard systems)	5.7 million
Boeing 787 Dreamliner	6.5 million
Premium-class automotive software	100 million

Table 1 - software complexity examples

1.2.3 Misconceptions about software

Software is free: even though some typologies of software may be released without any costs for the final users, it must be kept in mind that software is never free, and its production is very labor intensive. Assuming

a productivity of about 200 to 1000 lines of code per person month, software costs from $8 to $40 per single line of code. A medium sized project, of about 50.000 lines of code, costs between $400.000 and $1.600.000 in personnel. For companies, the cost of software is dominant with respect to the cost of hardware, as shown in figure 8.

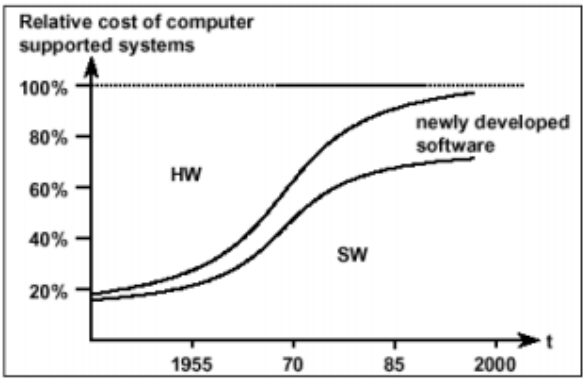

Figure 8 - Relative cost of hardware supported by software (Boehm 1976)

Software is soft: Even if at a smaller extent than for hardware, changing and managing software is difficult and often very costly, and change always happens. If the lifespan of a software system is long, the maintenance costs become way higher than the development costs. At a certain point, due to *architecture erosion*, the maintenance of software becomes nearly impossible.

Software is produced. Software is not mass produced, like machines are. The main difference between software and hardware costs lies in the fact that the replication of software is almost effortless, while the manifacturing of hardware is the most critical costs to be substained. Software is *developed* in a non-deterministic, creative process due to human involvement that can introduce defects (which cannot be introduced by production), and that can be controlled in a probabilistic manner only. In the case of sotware, the description of the solution itself *is* the product.

Software ages. As opposed to hardware, software does not break as it ages, hence the hardware reliability concepts cannot be applied to it. Failures in software do not occur due to material fatigue, but due to the execution of logical faults. However, software cannot be perfect, and a chance of software faults is always present: all software faults may not be removed before the execution of software. In addition to that, software is not stable and may change, due to requirements changes, platform changes, and defect correction; each change (even corrective ones) may add new faults to software.

1.2.4 Functional vs. Non Functional requirements

Functional Requirements describe functional characteristics of software, and behaviours that the system must perform. In general, functional requirements describe *what the system is supposed to do.* For instance, a functional requirement for a software may be:

- Add two integer numbers.

Non functional requirements specify how the system and its operations can be evaluated, rather than behaviours of the system. In general, non-functional requirements describe *how the system should be*.

Examples of non-functional requirements are:

- User interface must be usable by not-computer expert;
- Precision: relative error < 10-9, absolute error < 10-8;
- Reliability: sum must be correct 99,9999999% times;
- Performance: sum must be performed in less than 0,01 milliseconds;
- Efficiency: sum must use lessthan 10 kbytes of RAM in memory.

Non-functional properties are sometimes harder to espress than functional properties, and in particular they are harder to designe into software. They are rather emerging properties that depend on the system as a whole, and that can be typically measured only on the finished and running system.

1.2.5 Software Engineering vs Solo Programming

Software Engineering supports professional software development, rather than individual ("solo") programming: "multi-person construction of multi-version software", according to a definition given by Parnas.

Solo programming can be sufficient only for solving a small niche of simple problems. In these cases, the size of the projects is typically small, so that one person can manage all the development. The developer is typically the user itself, so there are no communication problems in the definition of the requirements and in the acceptance of the software, and typically only functional requirements (i.e., what the software *must* do) are taken care of. Solo-programmed sotware have a limited cost (typically they are free) and they do not need to be mantained for a long period of time.

On the other hand, projects needing practices of software engineering are typically large-sized, with teams of developers that need to share documentation, and that must face communication and coordination problems. User is not the developer, hence there are 3rd party requirements that must be communicated with the user, and non-functional properties (e.g., performance and usability) must be taken care of.

Table 2 summarizes the differences between projects that are compatible with solo programming, and projects that need software engineering techniques.

	Solo programming	**Software Engineering**
Size	Small	Large
User	Developer	Not the developer
Lifespan	Short	Long
Cost	Development	Development, operation, maintenance
Properties	Functional	Functional, Non Functional

Table 2 - Differences between solo programming and software engineering

The characteristics of projects managed by software engineering projects lead to a set of se-related issues.

- Large projects required team based development: efficient communication and coordination between team members is needed;
- Long lifespan: maintenance, and communication between developers and maintainers, are required;

- Third party requirements: communication is required between computer specialists, and non-computer specialists; non functional properties become essential, to measure the outcome of the developers' work.

1.3 Process and Product properties

A process is a defined set of activities, people and tools that are needed to obtain a certain product. Products (that may be documents, data, or code, in the case of software production) are the outcome of a given process. The quality of the product depends on the quality of the process. Parameters exist for characterizing the quality of both process and product.

Figure 9 - Process and product

1.3.1 Process properties

Cost: direct and indirect costs of a given projects, in terms of money.

Effort: work that must be completed during the execution of a process, in terms of person-hours or machine-hours.

Punctuality: capacity of the process to be on time, reaching its completion before a fixed deadline.

1.3.2 Product properties

Functionality: the product has a set of functions that currectly satisfy the needs at the base of its production (both explicitly stated and implied needs).

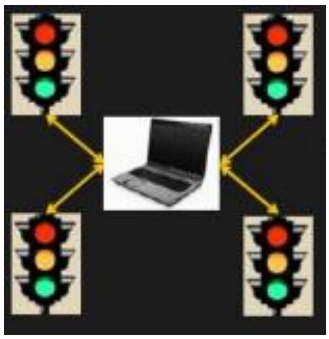

Figure 10 - Traffic light controller example

Example: Traffic Light Controller. The device has to control 4 traffic lights in a road crossing, so that:

- When green light is shown in one direction, red light is shown in the other. Duration of green signal is x seconds;
- While flashing yellow light is shown in one direction, red light is shown inthe other. Duration of yellow signal is y seconds;
- Total duration of red light in a direction is z seconds.

Correctness: the capability of the product to provide the intended functionalities and perform their exact tasks, as given in the specification, in **all** cases.

Example: for the Traffic Light Controller, Correctness would be the certainty that the intended sequence of signals is **always** shown by the device.

Reliability: the ability of a system or component to perform its required functions under stated conditions, for a specified period of time. Even though failures may be acceptable, a system or software should have a long mean time between failures to be considered reliable. Reliability parameters can be expressed in terms of the probability or count of failures during a given period of time.

Example: the Traffic Light Controller is considered reliable if the intended sequence of signance is given with high probability (e.g., P = 99.9%) during a year. Or, if there is at most one failure every year.

Performance: measures *how well* the product carries out its main functionalities. In general, performance measures pertain time (e.g., speed or delay in performing a function) or space (e.g., memory or CPU required by a software to perform a specific function).

Safety: capability of a product to avoid hazards, that can compromise the security of people or other things.

Example: the Traffic Light Controller should never allow the contemporary exhibition of green light in both directions.

Robustness: the capability of a product to exhibit a reduced set of all its functionalities when adverse conditions are present, limiting the damages that such conditions can cause.

Example: in case of broken cable, the Traffic Light Controller should provide a safe behaviour, so not to harm people in the street (for instance, only red lights, or only flashing yellow lights).

Usability: the ease of use a given function, measured in terms of the effort needed by a non-trained person in using it. It is an assessment made by the user about using the product.

Figure 11 - Devices with different levels of usability

1.4 Principles of software Engineering

The principles of software engineering are fundamental, broad coverage ideas, that are capable of producing positive and useful effects in the production of software.

1.4.1 Separation of concerns

Separation of concerns is the ability of splitting a large and difficult problem in many independent parts, that can be considered and resolved individually. The concept is analogous to the “divide and conquer” approach in war strategy.

In Software Engineering, the Separation of concerns translate in the concept of Software process: concentrate first on what the system should do, then on how the system should be done, finally make the actual system.

Separation of concerns can also apply to functionalities of the software itself. Software Engineering provides design patterns for the production of software, that can help keeping separated the components of software pertaining to different functions.

1.4.2 Abstraction

The ability of extract a simpler view of a difficult problem/system, aviding unnecessary details. Then, reasoning is made only on the simpler view, which is called model.

Abstraction is a technique that allows to cope with the complexity of systems, and is one of the most important principlesin object-oriented software engineering, for which it translates to the ability of identifying atomic software artifacts (*object*s) that are able to model the problem domain.

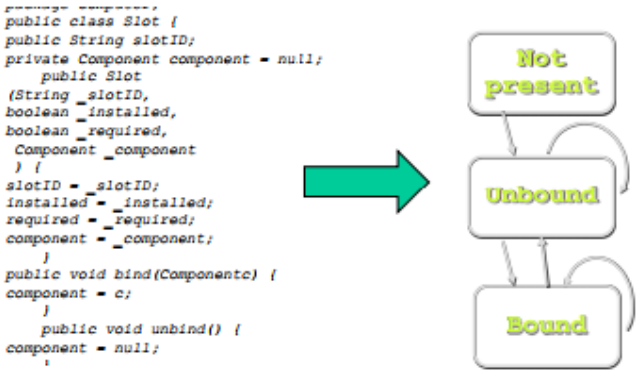

Figure 12 - Abstraction: from plain code to models

1.4.3 Modularity

The ability of dividing a complex system in models, with high cohesion (i.e., the degree of similarity of elements that are inside the same module) and low coupling (i.e., the degree of similarity of elements that are in different modules).

In software development, modules can be divided based on their functionalities, and can be substituted with other ones without affecting the functionalities of the whole system (e.g., developers can use prewritten

code). Having separate modules also allows to implement new functionalities in an additive way, without taking care of the already present ones.

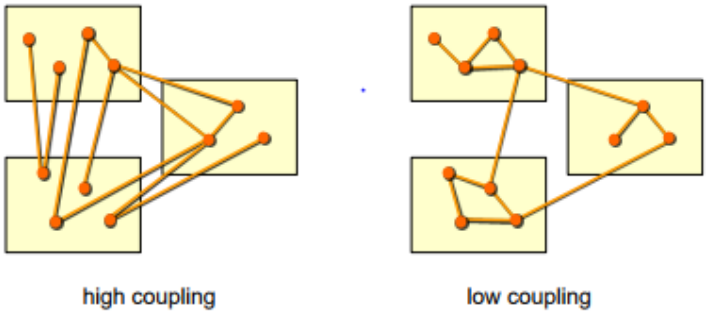

Figure 13 - Highly coupled and uncoupled modules

1.4.4 Information Hiding

The ability of hiding the internal mechanisms and design choices of each module to other modules. It can be considered as another form of modularity, since less information exchanged between modules leads to minor coupling. As modularity, information hiding provides additional flexibility.

As an example, a calculator software should expose the currect functionalities and the abilities of giving inputs and receiving outputs (i.e., the calculation results) to the user, but the algorithms used for the computations remain hidden.

II THE SOFTWARE PROCESS

2.1 Activities of the software process

The software process can be defined as a structured set of activities that are required for the production of a software system. Many software process exist, but they however present similar activities.

2.1.1 Production Activities

The goal of the software process is to produce software with defined and predictable process properties (cost, duration) and product properties (functionality, reliability, and so on). Code is not the only outcome of the software process, since also documents and data are produced.

Software is made especially by executables, that are the final outcome of any software process. Developers do not produce directly the executables, but the first intermediate artifact of the software process: the source code. Hence, the **coding** phase is, from the bottom up, the first activity of the software process.

Source code is often large, with many lines of code, and needs to be organized in several physical units (like files, directories, devices) and also logical units (functions, packages, subsystems, classes in object oriented programming). The **design** phase of the software process is the activity of defining and organizing the units of source code that must be developed.

To know exactly what software is made for and what it should do (e.g.: add numbers, count cars, forecast weather, control mobile phones, support the administration of a company…) **requirements** must be looked at.

Requirements Engineering	What the software should do
Architecture and Design	Which units, and how to organize them
Implementation	Write source code (executables) and integrate units

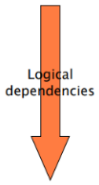

Table 3 – Three steps of the production activities

Thus, the production activities are divided in three steps: collecting the requirements and writing it in a formal way (requirements engineering); identifying and organizing the units in which the software is subdivided (architecture and design); writing source code, integrating different units and obtaining the executable code (implementation).

Logically, each of the production activities depend on the previous ones: to create the design of the software, requirements must be known in advance; to implement the code, both design and requirements are necessary.

A first, simple approach to the production activities of the software process is to perform them in sequence (see Waterfall Model). However, each phase may provide feedbacks to the previous ones, so that the process may cycle again over phases that have already been performed.

The production activities create a set of documents, especially for what concerns the requirements and design of the software system.

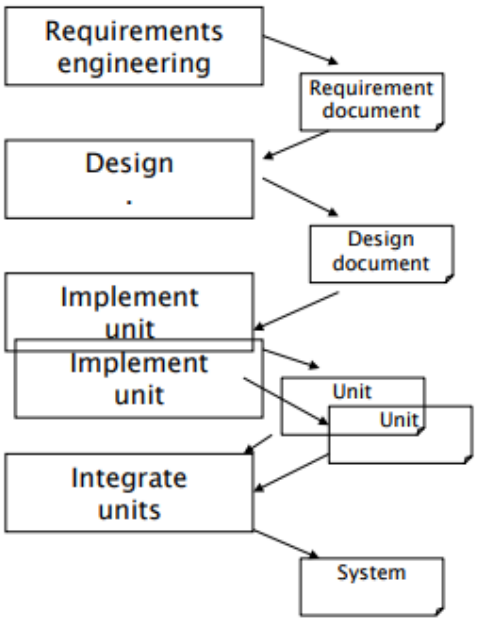

Figure 14 - The sequence of production activities

Figure 14 shows all the production activities of a software process, and the intermediate items that are produced and exchanged between them. The Requirements engineering produces requirement documents, that are used by the Design phase. Once the architecture and design is defined, design documents are created. Based on the subdivision in units made in the Design phase and detailed in the design document, the first step of the implementation activity is to separately implement the units of the software system. Finally, the units are integrated to obtain the final software system.

2.1.2 Verification & Validation (V&V) Activities

The software process must include activities to verify the outcome of the production activities. Once the units of the sotware are developed, questions similar to the ones following should always be answered.

- Does it work?
- Is it doing what it should do?
- Did we understand the requirements correctly?
- Did we implement the requirements correctly?

After each step, it is necessary to do the verification and validation (V&V) activities. Those are mandatory for all steps, especially for the initial ones. In fact, an error in the first phases of the software process will influence all the following phases.

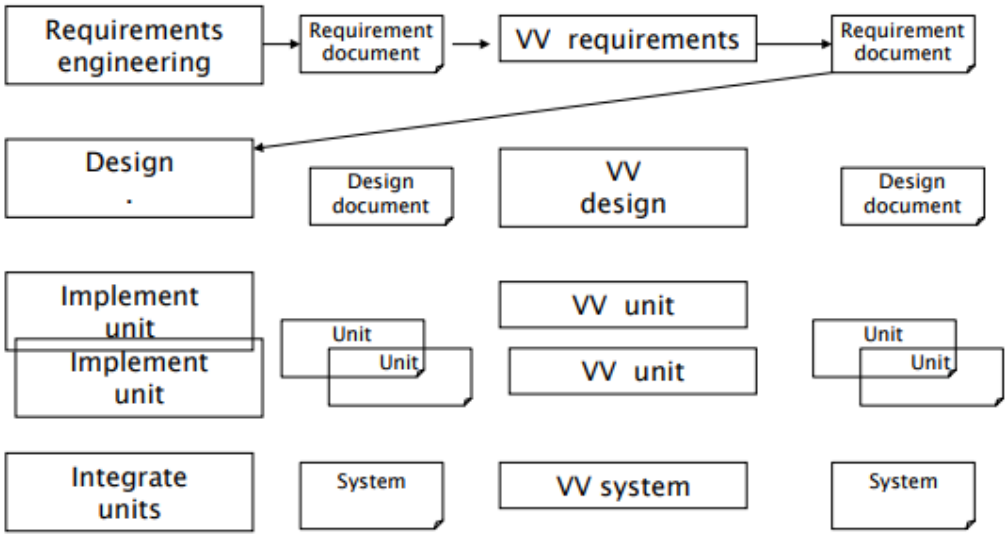

Figure 15 - Production + VV activities

Verification and Validation means that it must be controlled that the steps were correct both from the external (if they are coherent with the previous phases or with stakeholders' wills) and internal (if they are inherently correct, e.g. if the coded units are behaving correctly) points of view.

	Objective of V&V	**Externally**	**Internally**
Requirements engineering	Control that the requirements are correct.	Did we understand what the customer/user wants?	Is the requirements document consistent?
Architecture and design	Control that the design of the system is correct.	Is the design capable of supporting the requirements?	Is the design consistent by itself?
Implementation	Control that the code is correct.	Is the code capable of supporting the requirements and the design?	Is the code consistent (syntactic code checks)?

Table 4 - V&V Activities

2.1.3 The management activities

The final issue that is raised about the development of a software system is about the personnel that will perform the activities, and about the organization of the workforce.

- Who does what, and when?
- Which resources will be used?
- How much will the project cost? When will it be finished?
- Where are the documents and units? Who can modify that?
- Are we doing it state of the art?

All those questions pose the need for management activities.

Project Management. Organization of all the activities related to a software project and its parts, in terms of the assignation of work to people and monitoring of their progress, and estimation and control of the available budget.

Configuration Management. Software projects evolve and change during time. Configuration management is the activity of identifying and storing all documents and units pertaining to a software project, and keeping track of the software versions and updates. Configuration management can also be called configuration control.

Quality assurance. The activity of verifying whether the specifications are met from the finished software. Measurable quality goals must be defined for the software project, and must be controlled (through V&V activities) once the project is delivered. Also how the work will be done is defined within this activity.

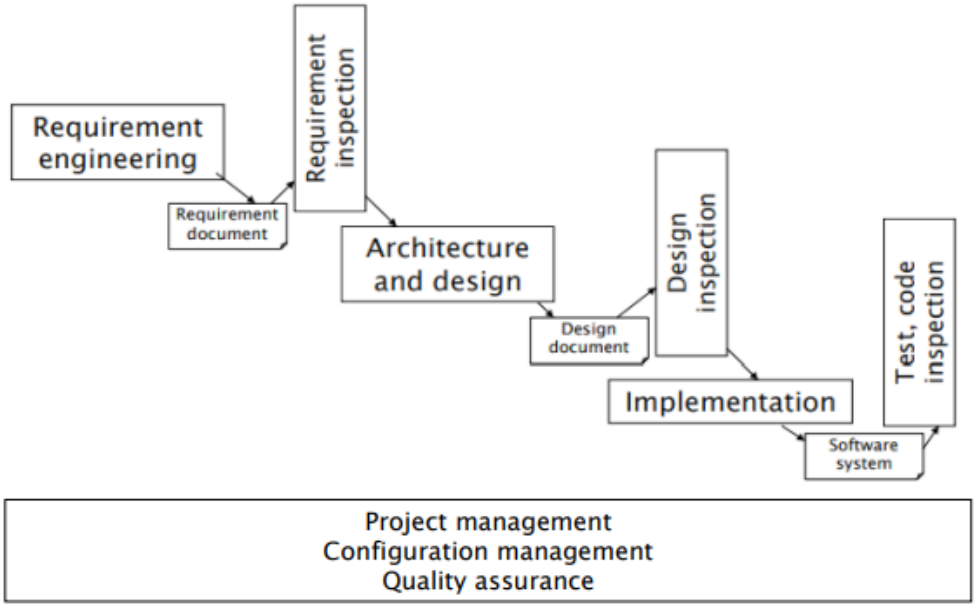

Figure 16 - The whole picture

2.2 Phases

Development of software is just the first part of the game. In fact, the software must be first of all deployed and put in an operative condition; then, the software must be maintained, and modified according to incoming needs; finally, the when the software can not be maintained anymore, it must be retired.

Operate the software	**Deployment, Operation**
Modify the software	**Maintenance**
End up	**Retirement**

Table 5 – Aims of the phases of the software process

Process activity:
1. Requirement engeneering
2. Design and architecture
3. V&V

Management activity:
1. Project management
2. Configuration management
3. Quality assurance

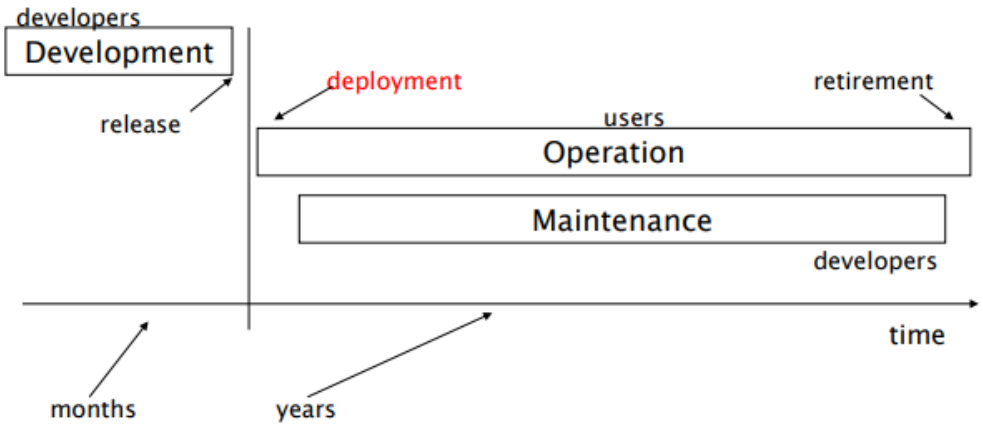

Figure 17 - Main phases of the software process

The aim of the **Maintenance** phase is to improve the software quality (e.g., for what concerns performance or other attributes), meet changing customer needs, or perform corrective actions on the deployed software to fix defects. Maintenance can be seen as a sequence of development phases (*dev_1, dev_2, …. dev_N,* with *dev_0* being the initial Development phase). Development and maintenance perform the same activities (requirements, architecture, design, coding). At the end of each development phase, a **release** of the software is obtained.

Indeed, the first maintenance is usually the longest, and the following are constrained by the previous ones. In *dev_0*, *requirements_0* are derived from scratch: this gives total freedom in defining the requirements for the first development phase. *design_0* and *implementation_0* are based on *requirements_0* only. In the next development phase, *dev_1*, the requirements *requirements_1* are derived from *requirements_0, design_0,* and *implementation_0*. In the same way, the requirements of *dev_N* will have to be derived from the requirements, design and implementation of all previous development phases, from *dev_0* to *dev_(N-1)*.

For instance, if in the *dev_0* phase the programming language chosen is Java, all other development phases will have to use Java. Likewise, if in *dev_0* the client/server model is adopted for the software, all further developments will have to keep it.

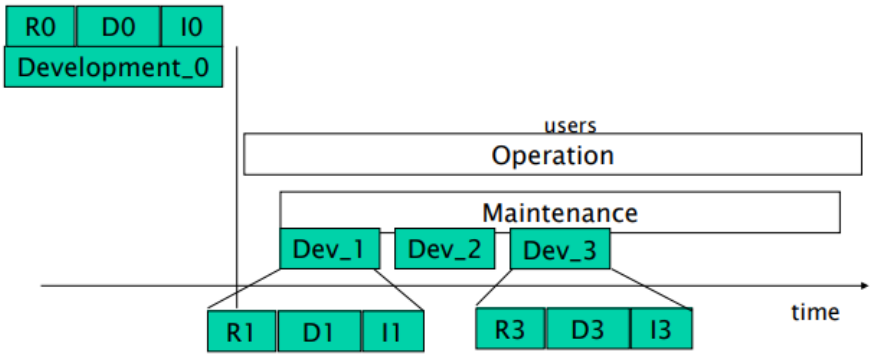

Figure 18 - Maintenance as a series of development phases

After years of operation of the software, the constraints posed by previous maintenances become too many, and making changes becomes almost impossible. Maintenance is usually the most expenive phase of the

software process: for instance, in software that supports businesses, the cost for maintenance can be up to 60% of the overall costs.

Scenario	Development	Operation	Maintenance
IT to support business	Several months	Years	Years, up to 60% of costs
Consumer sw (games)	Months	Months (weeks)	Virtually no maintenance
Operating System	Years	Years	Years, up to 60% of costs
Commercial OS (MS)	2 – 3 years.	Several years.	Several years, with daily patches and major releases (service packs) at long intervals. Parallel development of a new release.

Table 6 - scenarios in development

2.3 Comparison with Traditional Engineering

The software process is not an invention that has been created recently for software development: indeed, it is just the *application* of the engineering approach to software production.

As a case study, in the following what is done by aeronautics engineers is shown, so to highlight the equivalence with the individual phases of the software process described before.

Production Activities	**Requirement definition ("what")**	Airplane, civil usage; Capacity > 400 people; Range > 12000km; Parameters about noise level, consumption, acquisition and operation cost…
	High level design ("how")	Blueprints of the airplane; Definition of subsystems (avionics, structures, engines); Mathematical models (structural for wings and frames, thermodinamic for the engines).
	Low level design	Definition of further subsystems (in several cases subcontracted or acquired, e.g. for the engine).
	Implementation	Implementation of each subsystem.
Test Activities	**Unit test**	Verification that the subsystem complies to its specification.
	Integration	Put subsystems together (e.g., wings + frame).
	Integration test	Test the assemblies.
	Acceptance test	Does it fly?
Management Activities	**Project Management**	Project planning; Project Tracking; Budgeting, accounting.
	Configuration Management	Parts and assemblies; Change control.
	Quality Management	Quality handbook; Quality plan; Roles management.

Table 7 - Aeronautic Engineering Process

It is evident that the engineering process for a traditional manufacturing sector follows the same phases that are applied to software development: a preliminary phase in which requirements are defined; an architecture and design phase in which the elements of the final output of the production process are specified; an implementation and integration phase in which the individual elements are created and then put together; a validation and verification phase; finally, a set of activities in charge of managing the product and the process itself.

The difference between traditional engineering and software engineering lies principally in the fact that software engineering is still a relatively new discipline: while traditional engineering techniques are hundreds of years old, software engineering has been around for just about 50 years. The consequence of this fact is that the maturity of customers and managers, that is ensured for traditional engineering, is highly variable for software engineering.

In addition to that, traditional engineering bases its theories on a set of *hard* sciences, ranging from physics to law, mathematical models and so on. By converse, software engineering is based on a limited set of theories and law, and to some extent can be considered more a kind o *social* science.

2.4 System and Software Process

Since the nature of software may vary, also the nature of the process behind it varies accordingly. For instance, a stand-alone software may be developed by a software process; on the contrary, the development of an embedded software, which is tightly coupled to the hardware on which it is installed, requires the execution of a **System Process**.

The phases of a **System Process** are initially close to those of a normal software process, with a first **System Requirements Engineering** phase in which the requirements for the whole system are obtained, and a **System Design** phase in which the architecture of the whole system are defined.

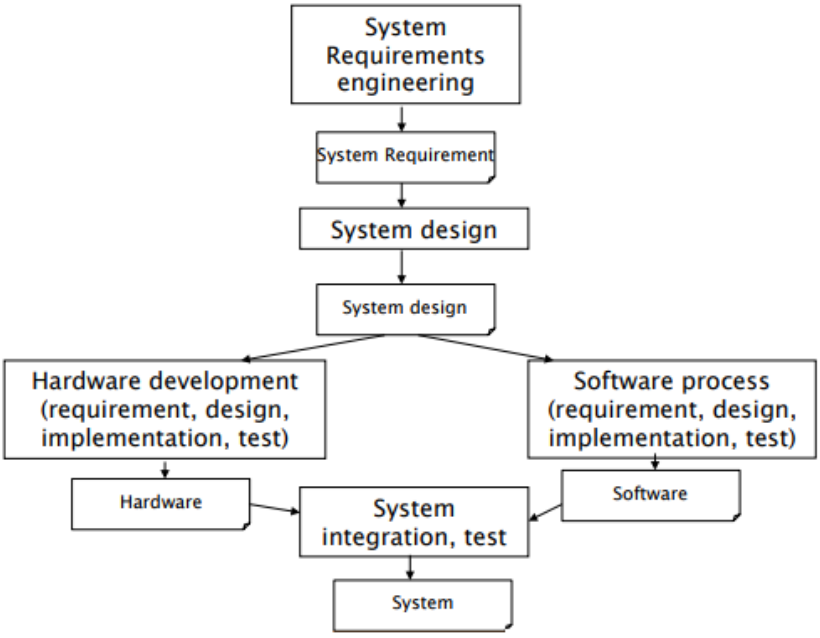

Figure 19 - The System Process

The process then forks, and two new processes start: an **Hardware Development** process, and a typical Software Process. Both are full-stack processes, with their individual requirement, design, implementation and test phases. The outcome of the two processes are the hardware and software component of the system, that are then integrated in the final **System Integration** phase to form the system, and tested together.

2.5 Software Engineering Approaches and Recent Trends

Software engineering proposes a set of different approaches to the software development process. In general, the activities that must be carried out for the development of any software do not change much, with the typical phases of production, verification and validation and management that are always present. In any software development process, information and decisions must be written, shared and controlled, typically with the use of documents.

Software engineering codifies ways in which the information and decisions about software are written and shared (with formal languages with which documents can be written) and offers techniques and models that help supporting all the development activities and the process as a whole (e.g., CMM, CMM-I, Iso 900-3, and so on).

2.5.1 Main approaches to software development

Even though the ways of putting together all the activities of a software process can be many, at least three fundamental approaches can be recognized, in addition to the basic solo programming.

Cowboy Programming. Cowboy coding just focuses on code, with the philosophy that any other activity is time lost, that should not be done by real programmers. It is a completely undisciplined approach to software development, in which the control of the development process is fully given to the programmers.

Cowboy coding focuses only on quick development and fixes, with the objective of getting a finished product as fast as possible. No formal process is adopted for requirements elicitation and testing, thus leading to a higher possibility of errors and failures after deployment. The absence of clearly defined requirements and design makes the cowboy-programmed software very difficult to integrate with other software, and to maintain over time.

This approach to programming is still applied, especially when there is lack in resources or the project deadlines are very tight.

Document Based, semiformal, UML. A semiformal language (UML) is used for writing documents. Transformation and controls are not made automatically, but by humans.

This approach is typically adopted in normal industrial practice, by mature companies and domains that have to manage large projects.

Formal / Model-Based. Formal languages are adopted for all the documents and specification of the software to be developed. Controls and transformations of the deployed software is made automatically, and not by humans.

This approach has a limited adoption in critical domains. It is typically used not for entire projects but for small part of them that are then integrated: it does not scale up to large industrial projects.

Agile. The agile approach focuses on code and tests, trying to limit the use of documents: it is a light-weight methodology, that encourages rapid iterations of the life-cycle of a project.

It is the latest approach being defined (introduced in 2001 in the *Agile Manifesto*) and there is still debate about the advantages and disadvantages it offers. The adoption of agile methodologies is still limited, but increasing.

2.5.2 Recent trends in sotware engineering

Component-based software engineering. Instead of being focused on just building the parts of the system to develop, component-based development evaluates the possibility of buying and then integrating components, either commercial or open source, i.e. the off-the-shelf components. The technique encourages the reuse of components, and highlights the phase of definition of the software architecture. The key drivers for the technique are saving time and money when building complex systems, and improve the quality of the developed system by using reliable and tested components.

Offshored outsourcing. The process of moving the development and maintenance of software abroad, for reasons related to time and money saving or different skills available. Offshoring guarantees enhanced cost and time effectiveness, at the expense of a possibly lower quality and security of the outcomes of the development.

2.5.3 Business models for software

Many business models are available for companies to monetize the software they develop, in addition to the traditional scheme of selling licenses for an unlimited use of a software.

Traditional (on premise). The software is paid with a one-time license fee, i.e. the right to use a software. Additional expenses may be asked for support and maintenance.

Open Source. The software is free, and the user is charged for support maintenance only.

ASP (pay per use). Software is run on the provider's machines, and accessed by the users through a network, typically internet. Users pay for using the software, rather than purchasing it (e.g., mySAP.com).

Freeware and pro versions. A light version of the software is distributed free of any charge to the user. The professional version is charged.

Shareware. The software is distributed freely, to facilitate a trial use. Users pay for it if they decide to keep and use it (e.g., WinRar).

Adware. The software isfree. The interface shows advertisement banners, that are refreshed by internet (e.g., Eudora).

III REQUIREMENTS ENGINEERING

3.1 Basic Concepts and Definitions

Requirements Engineering aims to collect and formalize the stakeholders' wills. Also, this process involves the analysis of the requirements, to be sure that they are correct and well formalized.

Requirements Engineering is a key phase of software development: most of the defects of the final product come from this phase, and they are the most disruptive and expensive to fix. To avoid such errors, validation and verification of the requirements is essential.

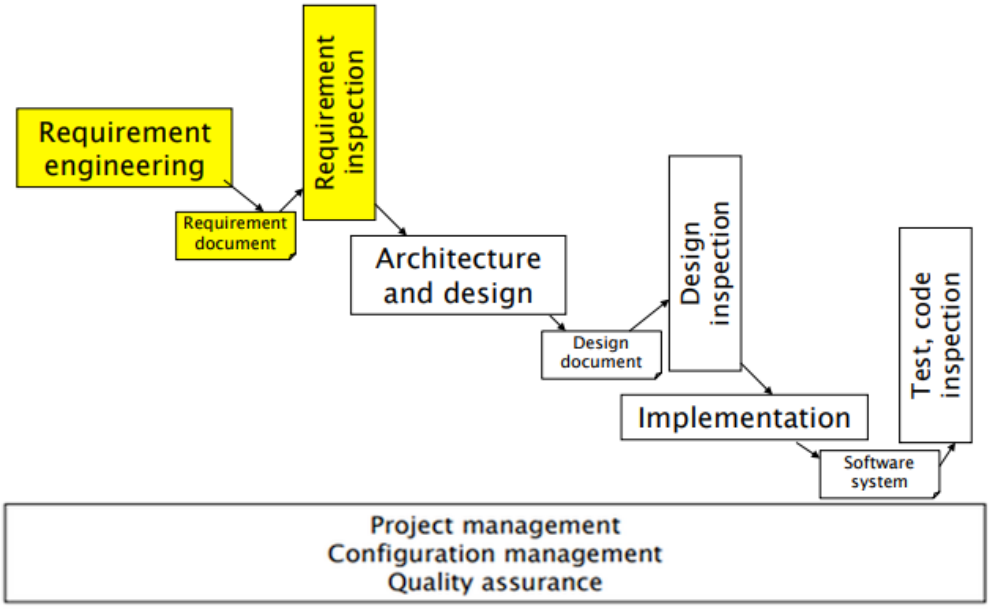

Figure 20 - Requirements phase in the software process

3.1.1 Domain

A **Domain** is a collectin of related functionality, and/or a collection of all applications with the same functionalities. It defines a set of common requirements, terminology and functionality or all the software programs that belong to it. Higher level domains can be specialized by subdomains, which pose additional requirements for the applications belonging to them.

> *Example: Banking domain, that includes subdomains like account management, portfolio management, etc.*
>
> *Example: Telecommunication domain, that includes subdomains like switching, protocols, telephony.*

An **Application** is a software system supporting a specific set of functions, and that belongs to one or more domains.

3.1.2 Stakeholders

A stakeholder is a role or a person with an interest (stake) in the system to be built, that can range from the final user of the system, to the developers that are coding it. Each profile has a different interest in the system, and that must be considered when the software is in the development phase. The following are the typical staeholders for a software project:

- User: uses the final system. Different user profiles can be defined for the same system, each with different expectations from it.
- Buyer: pays for the system.
- Administrator: manages the final system.
- Analyst: expert in requirements engineering, and/or in the domain the system belongs to.
- Developer: writes the code by which the system is built.

Often, the analyst and developers are the only ones involved in the software creation process, and the others only need or use it. In recent Agile techniques, more cooperation is required between developers/analysts and other stakeholders of the same system.

In the phase of requirement collection, it is always mandatory to identify all the involved stakeholders, and understand their desires and needs.

User	Cashier at Point of Sale (profile 1); supervisor, inspector (profile 2); Customer at Point of Sale (indirectly, through cashier).
Administrator	POS application administrator (profile 3); IT administrator, managing all the applications in the supermaret (profile 4); Security manager, responsible for all security issues (profile 5); DB administrator, managing DBMSs on which applications are based (profile 6).
Buyer	CEO and/or CTO and/or CIO of supermaret.

Table 8 - stakeholders for a Point of Sale (POS) in a supermarket

3.1.3 Requirements

A requirement is a description of a system or a service and its constraints. Requirements describe what are the goals and objectives of the software, how it will work, what are its properties. A requirement can have different levels of abstraction and formality. It can be either functional or non functional.

For instance, taking into acount the Point of Sale example given before, an informal (set of) requirement can be the following:

A POS (Point-of-Sale) system is a computer system typically used to manage the sales in retail stores. It includes hardware components such as a computer, a bar code scanner, a printer and also software to manage the operation of the store. The most basic function of a POS system is to handle sales.

Most of the requirements engineering phase has to do with the stakeholders –other than developers and analysts- that are involved in the software process: requirements are tailored based on the needs of the client. Because of that, lots of communication between stakeholders is involved in the phase of Requirements Engineering. Initially, stakeholders give the requirements informally, and this is often source of mistakes and misunderstandings, because ambiguous requirements may be interpreted in different ways by developers and users (e.g., *manage sales* and *handle sales* are the same thing or not? What is a sale?).

Requirements should try to comply to two main characteristics, completeness and consistency.

- Requirements are **complete** when they include the descriptions of all the facilities required from the system. It should be ensured that there is no information left aside from the specification of the system.
- Requirements are **consistent** when there are no internal contradictions or conflicts in the descriptions of the system facilities. If ambiguities are present, they should be eliminated and properly signaled in the relative documents

Ideally, requirements should be both complete and consistent; however, in practice this is nearly impossible to achieve; hence, a verification and validation procedure is essential for requirements.

Most common errors in requirements specification are:

- Omissions: when some functionalities that the system should provide are not covered by requirements, or not well specified;
- Incorrect facts: when errors or wrong assumptions are present in the specifications of the system;
- Inconsistencies/contradictions: different behaviours are described for the same component;
- Ambiguities: the functionalities are specified, but in an unclear way;
- Extraneous information: information that are not fundamental for the requirements of the project;
- Redundancies: the same information is specified in more than one part of the requirements document.

3.1.4 The Requirements Document

To overcome the issues presented before, a standard **Requirements Document** should be used, using formalized techniques for writing the requirements. The requirements document provides a formalized structure; within such structure, and in its parts, different techniques can be used. The following sections should be provided in a Requirements Document.

- Overall Description: description of the need for the system, and brief description of the system functions, and how the system fits among other similar systems and the business of the organization requiring it.
- Stakeholders: description of all the stakeholders involved with the system.
- Context Diagram and Interfaces: definition of the boundaries between the system and its environment, and how the system will communicate with other actors.
- Requirements: description of the functional and non functional requirements of the system, in detail.
- Use case diagram: descriptions of individual functions of the system, as perceived and used by actors.
- Scenarios: description of possible complete interactions of actors with the system, starting from a goal and achieving a result.
- Glossary: definition of all the terms used in the requirement.
- System Design: description of the architecture and design of the system.

It is not mandatory to provide all the sections of the requirements document, and in a precise order: what is fundamental is that the individual parts are described with sufficient precision.

Other structures are available for the Requirement Document; for instance, a simpler structure is given by IEEE std 830 1994, which contemplates: Introduction, General Description, specific requirements, Appendices, Index.

Overall Description	Text
Stakeholders	Text
Context Diagram	UML UCD
Interfaces	Text, PDL, XML, screenshots
Requirements	Type, Numbering
Use Cases	UML UCD
Scenarios	Tables, Text
Glossary	Text, UML CD
System Design	UML CD

Table 9 - Requirements Document and Techniques

3.1.5 V&V of Requirements

Requirements verification and validation –useful for ensuring, for instance, completeness, consistency and correctness- is a careful review process of all the requirements, that should be done as early as possible in the development cycle, to avoid more costly expenses for fixing them later.

Verification and validation can be conducted in different ways, according to how the requirements have been expressed. If the requirements are in natural language, or in UML, an inspection (proof reading) by the end user or developer may be a first form of validation. Some tools exist to perform syntactic checks on requirements expressed in UML form. If requirements are expressed using formal languages, model checking tools are available for verification and validation.

3.2 Context Diagrams and Interfaces

The Context Diagram defines what is inside the system to be developed, and what is outside. The outside entities are called *actors* and can either be other systems, external applications, or human users (the stakeholders). Actors are not matter of development: they are just considered as black boxes, that take some input and give some output (or vice-versa).

The Context Diagram also defines the **interfaces** between the system and the actors. Interfaces have two levels:

- **Physical Interface:** addresses the specific means through which the interaction between the system and the actor(s) has place.
- **Logical Interface:** defines the information flow of the human-computer interaction, without specifing the actual means of the dialogue (i.e., protocols are specified).

It is also usually specified what are the data formats of the interface (e.g., format of the data sent for a credit card, error messages).

Interfaces can also be classified in three types, each one having its formal notations that gives an effective technique for interface specification:

- **Graphical User Interfaces (GUIs):** (partial) depiction of the GUI that will be eventually shown to users, obtained with mockups and sketches (e.g., Balsamiq).

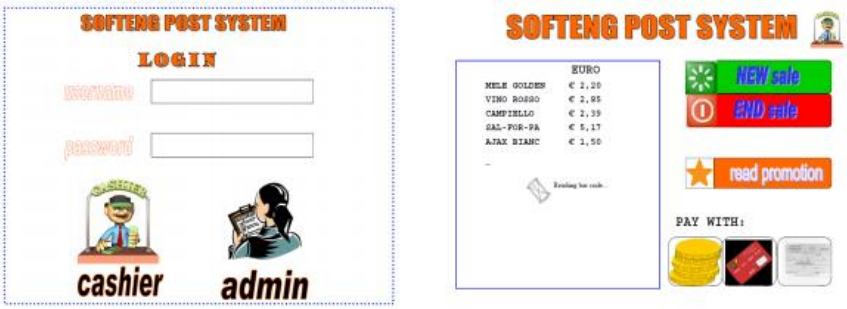

Figure 21 - GUI sketch

- **Procedural Interfaces:** documented and ordered set of methods and procedures of the system, described with PDL (Program Design Language).

```
interface PrintServer {

// defines an abstract printer server
// requires:          interface Printer, interface PrintDoc
// provides: initialize, print, displayPrintQueue, cancelPrintJob, switchPrinter

            void initialize ( Printer p ) ;
            void print ( Printer p, PrintDoc d ) ;
            void displayPrintQueue ( Printer p ) ;
            void cancelPrintJob (Printer p, PrintDoc d) ;
            void switchPrinter (Printer p1, Printer p2, PrintDoc d) ;
} //PrintServer
```

Figure 22 - PDL Interface Description

- **Data Exchanged:** the interaction is described only by the data that is given as input or output to the system and actors. XML and Json can be used to describe the exchanged data in a standardized way.

```
<food>
<name>Belgian Waffles</name>
<price>$5.95</price>
<description>
two of our famous Belgian Waffles with plenty of real maple syrup
</description>
<calories>650</calories>
</food>

<food>
<name>Strawberry Belgian Waffles</name>
<price>$7.95</price>
<description>
light Belgian waffles covered with strawberries and whipped cream
</description>
<calories>900</calories>
</food>
```

Figure 23 - Description of Data Exchanged, with XML

Context diagrams are typically represented with the UML Use Case Diagram (UCD) technique. In the context diagram, the central entity represents the system, which is connected to a set of actors interacting with it.

The context diagram must always be coupled with the definition of the interfaces, as in the following example.

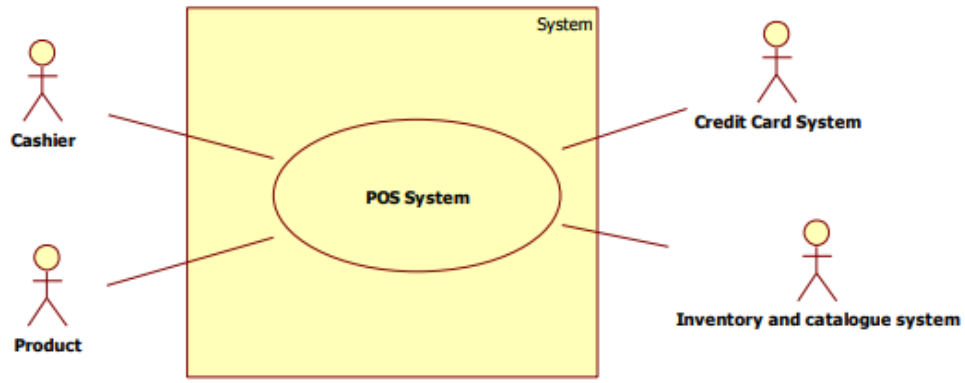

Figure 24 - UML UCD for a POS System

	Physical Interface	Logical Interface
Cashier	Screen, keyboard	GUI (to be described)
Product	Laser beam (bar code reader)	Bar code
Credit Card system	Internet connection	Web services (functions to be described, data exchanged, soap + XML)
Inventory and catalogue system	Internet connection	Web services (functions to be described, data exchanged)

Table 10 - Interfaces for the POS System

3.3 Requirement Types

Requirements can be subdivided between Functional and Non functional requirements, with the addition of Domain requirements that are taken from the related domain, and not proper of the system under development.

3.3.1 Functional Requirements

Functional requirements are the ones that specify the behaviour that the system should have. Functional requirements can be related to hardware, software or both, to calculations, technical details, how data will be processed. They also include the description of services, from a high level point of view.

Functional requirements can be related to the application (something that the software should do) or to the domain (requirements that are related to the domain of the application, e.g. for law or regulamentations related to a specific category of systems). To make the requirements clearer, it is common to use a glossary, with the words that are used to describe the functions.

3.3.2 Domain Requirements

Domain requirements are derived from the application domain, and describe system characteristics and features that reflect the domain. As said, these can be functional requirements, but also constraints on

existing requirements or definition of specific computations. If some domain requirements are not satisfied, the system may be not workable.

For instance, the following is a Domain Requirement for an healthcare system, that must be compliant to the regulamentation that exists in its domain:

> *The system safety shall be assured according to standard IEC 60601-1: Medical Electrical Equipment – Part 1: General Requirements for Basic Safety and Essential Performance.*

Another example of a Domain Requirement may be a description of how to carry out some computations, using domain-specific terminology. The following requirement is about the computation of the deceleration in a train control system:

> *The deceleration of the train shall be computed as: D_train = D_control + D_gradient Where D_gradient is 9.81ms2 * compensated gradient/alpha and the values of 9.81ms2/alpha are known for different types of train.*

Domain requirements have often problems of understandability and implicitness, since they are typically expressed in the language of the application domain and may not be understood by software engineeres: the use of a glossary may help solving this issue. Also, domain specialists understand the area so well that they may not think about making the domain requirements explicit.

3.3.3 Non Functional Requirements

Non Functional (NF) requirements are those that specify constraints on the system/service. Non Functional requirements are very general (they may pertain many different aspects of the system) but they are of mandatory importance.

According to ISO 9126 / ISO 25010, five categories of Non Functional requirements can be identified (the Functional requirements are considered under a sixth category of requirements, called *Functionality* requirements). In general, it is possible to identify a higher number of categories of Non Functional requirements, with a prior distinction between Product, Organisational and External requirements.

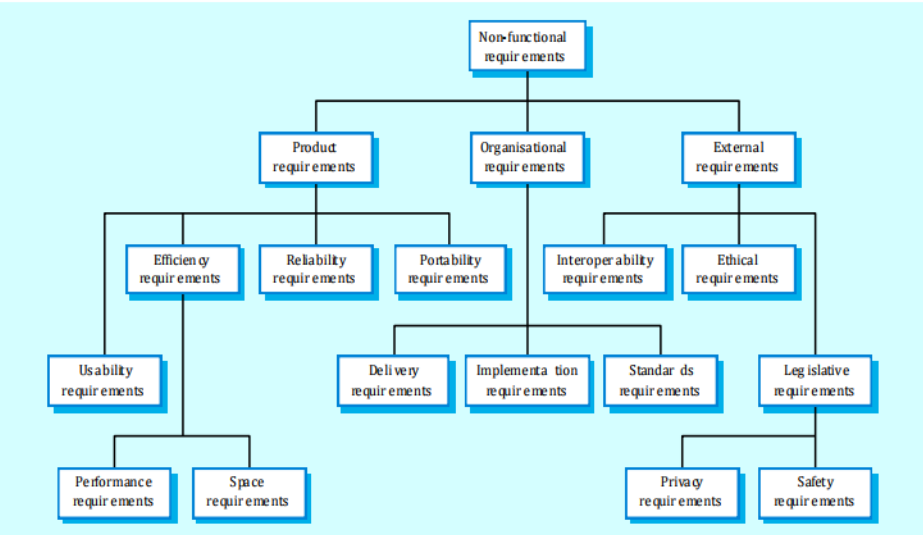

Figure 25 - Non Functional Requirements Taxonomy

- **Product requirements**: requirements that specify that the delivered product must behave in a particular way:
    - **Usability**: satisfaction obtained by the user while interacting with a system, and training time needed for a user to learn to use the software.
      Example: non trained user must be able to use the functionalities of the application in half an hour.
    - **Efficiency**: performance of the software, space and time needed for performing its functionalities.
      Example: waiting times of the application must be < 1 second (according to a theory of perception of waiting times, while delays shorter than 0.1 seconds are not perceived, users are annoyed by response time that are longer than 1 second).
    - **Reliability**: security of the software, availability (capacity of being working in almost all situations), accuracy of the offered behaviour.
      Example: system should work 99.9% of the time.
    - **Maintainability**: how much is easy and cheap to maintain the system, changing hardware and software components when it is needed (e.g. for new requests from clients, or for bug fixes).
    - **Portability**: how much is easy to make the software work on different platforms, domains or environments.
- **Organisational Requirements:** Requirements that are a consequence of organisational policies and procedures of the company.
    - **Delivery:** Constraints on the delivery of the final system, and characteristics that it should have.
    - **Implementation:** Constraints on the components, languages and practices that must be used in the development of the system.
    - **Standards:** Policies and standards that must be adopted mandatorily in any project of the company.
- **External Requirements:** Requirements that arise from factors that are external to the system and its development process.
    - **Interoperability:** Capability of the system to work in conjunction with other specified system.
    - **Ethical:** Possibility of the system to raise ethical issues.
    - **Legislative:** requirements about the possibility that the system or software may create harm to people or things, and regulamentations about the privacy of the users.
      Example: user name and password of the user must not be disclosed outside the software.

Non Functional Requirements can be formulated in a way that is measurable or not:

- **Not Measurable:** a general rule, for which no specific parameters are specified.
  The system should be easy to use by experienced controllers, and should be organised in such a way that errors are minimized.
- **Measurable:** it is specified a metric and a way to measure it on the system, with parameters (numbers) that tell whether the system satisfies the requirements or not.
  Experienced controllers shall be able to use all the system functions after a total of two hours training. After this training, the average number of errors made by experienced users shall not exceed two per day.

Since Non Functional requirements must be evaluated and verified on the working system, they should *always* be measurable.

Property	Measure
Speed	Processed Transactions / second User / Event response time Screen Refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Table 11 - Measures for NF Requirements

Often the Non Functional requirements are more critical than the functional ones; if they are not met, the system may be useless. For instance, a real time system must be very fast to report changes, and it is useless if its latency is higher than 1 second.

In complex systems, it is also frequent the presence of conflicts between different Non Functional requirements. These conflicting situationsmust be handled, and in general a prioritization of the requirements must be adopted, in order to identify the most important ones that should *always* be satisfied. As an example, consider a spacecraft system:

- To minimize weight, the number of separate chips in the system should be minimized;
- To minimize power consumption, lower power chips should be used;
- Using low power chips may mean that more chips have to be used: which is the most critical requirement?

3.3.4 Writing Requirements

Requirements are written in the Requirements section of the Requirements Document. Usually, all requirements are assigned an ID, useful for a fast identification. The ID indicates the type of the requirement (F, NF, Domain) and a progressive, unambiguous number.

Also, requirements can be nested: this means that there may be a main function (e.g., *F1 – manage sensors)* and, under it, a set of sub-functions composing the main one (e.g., *F1.1 – setup sensor, F1.2 – insert sensr in sensor list, F1.3 – get sensor information*).

Requirement ID	Description
F1	Handle sale transaction
F2	Start sale transaction
F3	End sale transaction
F4	Log in

F5	Log out
F6	Read bar code
NF1 (efficiency)	Function F1.1 less than 1msec
NF2 (efficiency)	Each function lessthan 0.25sec
Domain1	Currency is Euro – VAT computed as…

Table 12 - Functional Requirements for the POS System

3.4 Glossary

The Glossary gives, in a readable way, an explanation for all the terms used in the rest of the requirements document. It is useful to help developers and software engineers in understanding the domain language.

> *Sale = Commercial transaction between customer and retailer, where customer buys a number of products from the retailer.*

The glossary can be refined and accompanied by an **UML Class Diagram**. Class Diagrams can also be used to describe the application model and the system design.

In the UML Class Diagram, elements of the system to be described in the glossary are represented as *Classes*. Classes are described by a sets of attributes, with the relative description, rather than by their behaviour. As in object-oriented programming, classes are used to generate instances of objects, all with varying attributes but with the same structure.

The Glossary well represents only static relationships and associations between objects (classes). To represent dynamic associations between parts of the system, it is better to use scenarios (or use case diagrams) or sequence diagrams). Also, the Glossary is not made to design classes of the software (this is a design task) but to clarify the elements of the system and their relationships.

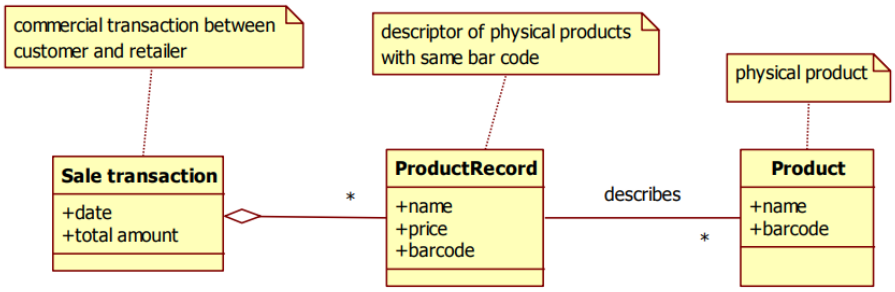

Figure 26 - Glossary (Class Diagram) for the POS System

Classes may represent different types of elements of the system:

- Physical objects: a physical element / person that is part of the system or interacts with it (e.g., Pilot, Airplane, Airline, Airport);
- Legal / Organizational entities: an organization which has a role interacting with the considered system (e.g. Company, Airline, University, Department);

- Descriptors: a standardized attribute for another class, that can assume one of a given set of values (e.g., Aircraft type, Pilot Qualification);
- Time (Events): date and timestamps of occurrence of events of importance in the system (e.g., Departure of an aircraft);
- Time (Intervals): classes representing periods of time of situations of importane for the system, that last a certain amount of time; they usually have a start date and an end date (e.g., Flight Internship);
- Commercial Transactions: record of the execution of a money exchange between other elements of the system, themselves represented by classes (e.g., insurance payment).

3.5 Scenarios and Use Cases

3.5.1 Scenarios

A scenario is a sequence of steps that describe a typical interaction with the system. It is made to show how things change over time, not to give a static depiction of the system: hence, it is complementary to the sequence diagrams and use case diagrams.

A scenario is often represented as a table having two columns: step number and description. Each row in the scenario represent a step of the interaction with the system. In a requirements document, there may be many scenarios, even hundreds. Scenarios do not require a high level of detail in their definition, since they should abstract the function of the system that must then understended and developed by the developers. Low level details in scenarios would create unuseful noise.

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

Table 13 - scenario for a transaction

3.5.2 Use Cases

A Use Case is a set of scenarios with common user goal (e.g.: All types of subscription with different payment system that can be successful or not) and is useful to give a picture of what the system offers. It is a summary of a high level functionality that the system offers. The use case wraps up nominal scenarios (the basic ones, which clarify the goals of the Use Case) and exceptions scenarios.

Use Case: Handle sales

scenario 1: sell n products;

scenario 2: sell n products, abort sale because customer has no money;

scenario 3: sell n products, customer changes one of the products.

Often a Use Case describes a set of similar scenarios relative to the interaction of an actor with the system. The three key pints of a Use Case are:

- The **Actor**: the individual entity (it can be a machine or a stakeholder) that interacts with the system. An actor can be primary, if it starts the interaction with the system, or secondary if the interaction is started by the system (hence, the actor is passive). It is essential that the actor is external to the system.
- The **system**: the system under specification, that in this case is viewed as a black box;
- The **Functional Goal**: what the actor achieves by using the system.

Use case can have three scopes:

- **Enterprise**: business processes use case, set of activities where people and maybe also other systems are involved. It is the highest level category of use cases;
- **Software System**: describes an actor and his interaction with the system, to achieve a goal;
- **Software Component**: interaction with a single software component; not to be considered in the requirements specification phase but in the design phase, being at a lower level.

Use Cases are usually described with a UML Use Case Diagram (UCD). Each use case is reported in the diagram and represents a value that the system offers to some actors. It is important to write down for each use case its goal and its description. This information cannot be embedded in the program. Moreover it could be useful to write down what actors are involved, for each use case.

It is important to underline in the Use Case diagram the complex peration to be performed in a certain scenario, to let the developer have a detailed view of what should be done, and what are the key points of the system.

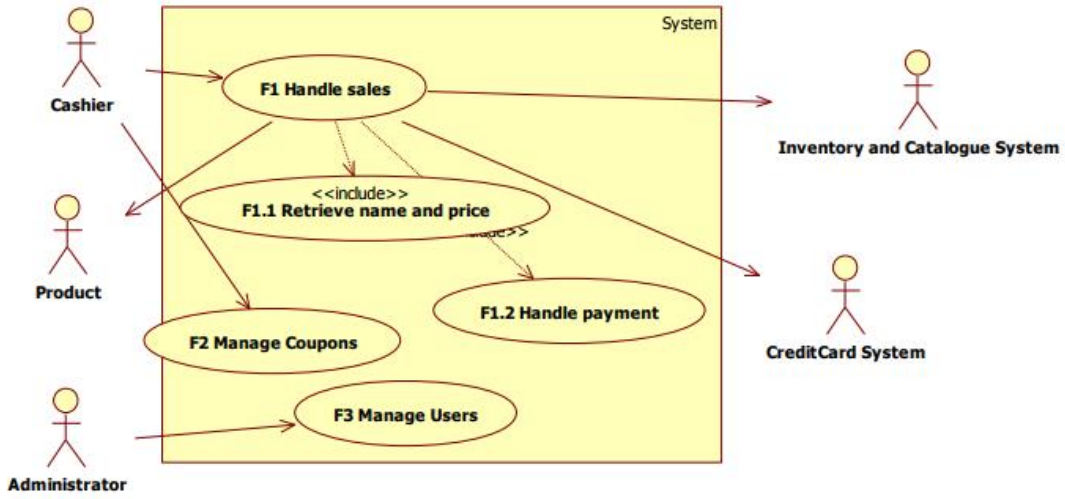

Figure 27 - Use Case Diagram for the POS system

In the example given in Figure 27, three different use cases can be seen. The first one is *Handle sales* (identified also by the functional requirement F1), that includes two complex operations, *Retrieve Name and Price* and *Handle Payment*. The *Include* relationship is used to highlight, in the Use Case Diagram, the fact that an interaction makes sense only if it is part of a higher level use case. Other two use cases are defined:

Manage Coupons, and *Manage Users*. The arrows point from the actor to the use case if the actor is active, from the use case to the actor if the actor is passive.

It is worth highlighting that the Use Case Diagram is very similar (but more detailed) to the Context Diagram.

3.6 System Design

The system design is the definition of the elements of a system, like its architecture, components, and all subsystems –software and not software- that compose the system.

It is an useful tool to have an overall vision of the system, to understand how the system is composed and what are its key elements. System design can be described using an UML Class Diagram, in which –for each element- it is important to show what are the main attributes, and the functions provided.

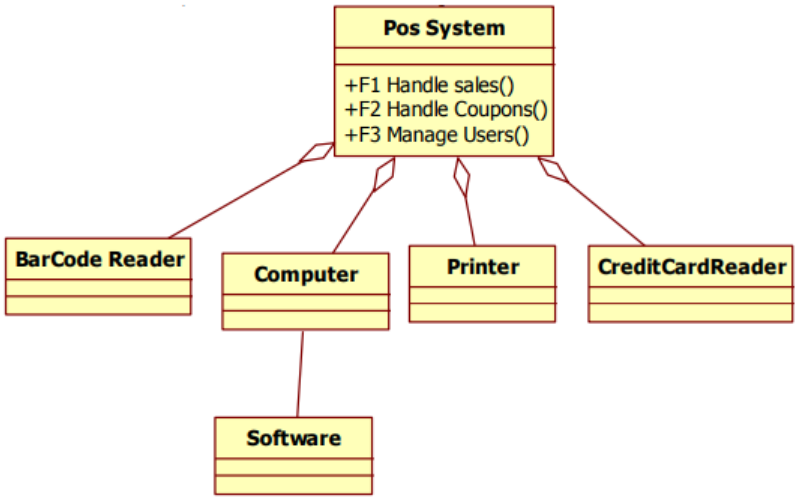

Figure 28 - System design for the POS System

IV OBJECT ORIENTED APPROACH AND UML

The object oriented approach has been the more influential, both in research and practice, in software systems development in almost the last 20 years.

Unified Modeling Language (UML) is the dominant notation based on the object-oriented approach. It is standardized by OMG (Object Management Group). There are many UML diagram templates, each one useful in doing different tasks. For instance, the UML Class Diagram (UML CD) models structures, entities and concepts: it is useful to design software but also to represent the system design and glossary requirements. The UML Use Case Diagram (UML UCD), instead, is useful to represent functions, i.e. what the system can do: in fact, it is used to represent the Use Cases and the Context of the system. Time, dynamics and temporal constraints of a system can be modeled using UML sequence (collaboration) diagrams, statechart diagrams, Activity diagrams.

4.1 Approaches to Modularity

Several principles exist in software engineering, about the creation of a modular structure of a system:

- **P1:** Low coupling and high cohesion, i.e. high intra-module correlation, low inter-module correlation;
- **P2:** Information hiding, i.e. interfaces representing the boundaries of each module;
- **P3:** Conway's Law, i.e. the design of the system should match the structure of the company.

Those principles encourage a *Divide and Conquer* approach to software engineering. Given them, there may be two approaches to implement the software: a *Procedural* approach, and an *Object-Oriented* approach.

4.1.1 Procedural Approach

In the procedural approach, the basic concept is the *procedure call* (or routine call, or function call), which is a set of computational steps that must be performed.

Using a procedural approach, the techniques of structured analysis and structured design come in handy for the performance of analysis and design of the product. Several procedural languages exist for coding: C, Pascal, Fortran, and so on.

Procedural approach defines two different kinds of modules: the procedures, and the data on which the procedures must work. Two different relations are defined: the *call* relation between procedures (with or without parameter passing back and forth) and the *read / write* between procedure and data. While the call relation creates low coupling between the involved procedures, read and write relations present higher coupling (highest for the write relation).

The use of global declarations, in procedural approach, makes possible for read and write relations to happen between data and any other function, without explicit declaration (in the form of parameter passing). This is permitted and likely happens, especially in the first phases of coding and during maintenance and evolution of the project. The presence of variables in global scopes also increases the coupling of code.

In general, the root problem of the procedural approach is the lack of an explicit link between the structured data, and the procedures working on it.

In figure 29 and 30, respectively, an excerpt of code working on a vector and its representation in the form of modulesand relations are given.

```
void init (int [] v, int size) {
     for (i=0; i<size; i++) {  v[i ]=0; }};
void sort(int [] v, int size) { // sort };
int main(){
    int vector[20];
    init(vector, 20);
    sort(vector, 20);
}
```

Figure 29 - Procedural approach - Code example

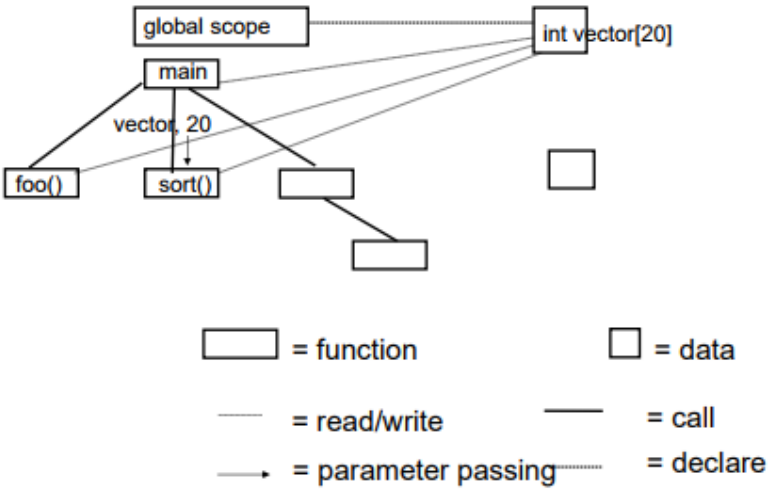

Figure 30 - Modules in the procedural approach

In the example, even though in the code excerpt the reads and writes are confined only in the functions that receive the vector as parameter (init, and sort), they can actually happen anywhere.

The functions init(), sort() and the data element vector[20] are not linked. However, they should, as they work in symbiosis: the parameter passing should be avoided, and the read and write relationship should be confined only within the functions. These two principles lead to the concept of object, that is the basis of the Object-Oriented Approach.

4.1.2 Object-Oriented Approach

The Object oriented approach identifies, as its basic module, the class. It is given support in analysis and design by UML, and several programming languages follow the object-oriented approach: C++, Java, smalltalk, C#, and so on.

In the OO Approach, classes describe structured (in terms of attributes) data, and procedures that can read and write them. Objects are instances of a class, in the sense that they are described by the set of attributes of the class, and the operations described by the procedures defined in the class can be operated on them. No read and write operations are performed outside the class.

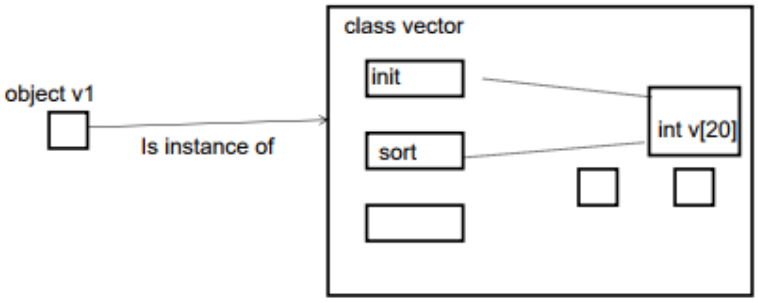

Figure 31 - Classes and Objects

The OO approach introduces the message passing relation between objects, which is similar to a procedure call with parameter passing, and hence creates low coupling. The message passing, with respect to a procedure call, hides read and write relationships, creating lower coupling between objects. The lower amount of relationships among objects, and the high level of abstraction of classes, allow to build more complex systems than the ones that can be built with a procedural approach.

Information hiding is guaranteed in the OO approach by *Interfaces*, i.e. the only set of messages that an object can answer to, and that are exposed to other objects.

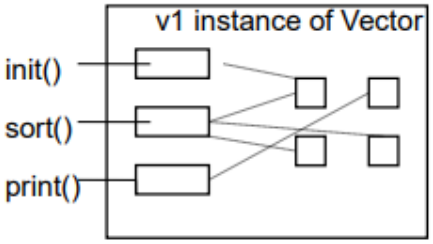

Figure 32 - Interface exposed by an object

The translation of the vector example introduced before to the OO approach is shown in Figure 33. A class is defined, with the declaration of data owned by each vector object, and operations that can be performed on it: the vector() constructor, and the sort() function.

```
class vector{
   private:
      int v[20];
   public:
        vector(){  // same as init }
        sort(){  // same as sort }

}
```

```
int main() {
   vector v1, v2;
   v1.sort();

}
```

Figure 33 - OO Approach - Code example

In conclusion, objects and classes are a better modularization elements, and by construction message passing has much lower coupling than procedure call and read/writes. The designer has the duty of deciding which are the “right” classes to implement the system in the correct way, and to guarantee information hiding.

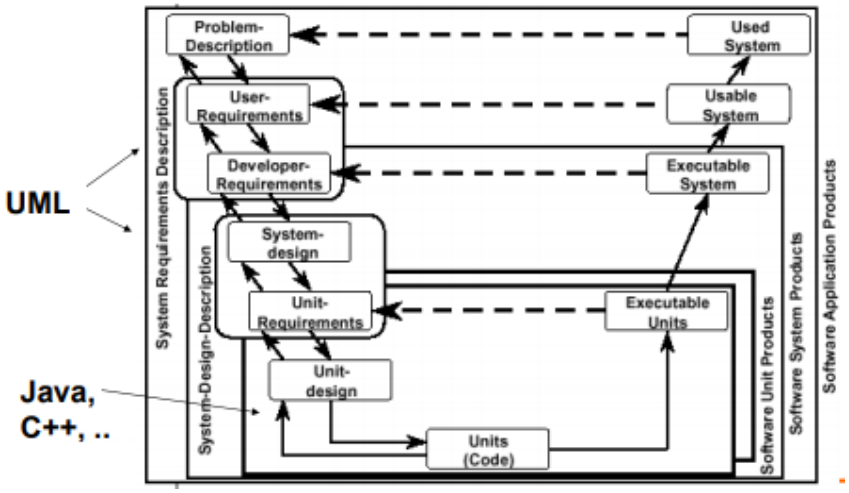

Figure 34 - OO Approach and the software process

4.2 UML Class Diagrams

Class Diagrams are a notation that can be used in different documents, with different goals: User Requirements, Developer Requirements, Model of concepts (glossary), Model of hardware + software system (system design), Model of software classes (software) design, and so on. They allow to create a conceptual model of the system, as a set of few domain level classes with relevant attributes and relationships. With the support of Object-Oriented programming languages, source code can be reverse-engineered from Class Diagrams.

4.2.1 Objects

The key concept to underline for the UML Class Diagrams is the Object: it is a model of an entity, and it is characterized by its identity, its attributes, its behavior and the messages that can receive.

In the UML CD, an object is represented by a rectangle divided in three sections, from top to bottom:

- **Identity:** the name of the object;
- **Attributes:** properties and parameters that characterize the object;
- **Behaviors:** methods/functions that the object can perform.

An object diagram models the objects of interest in a specific case.

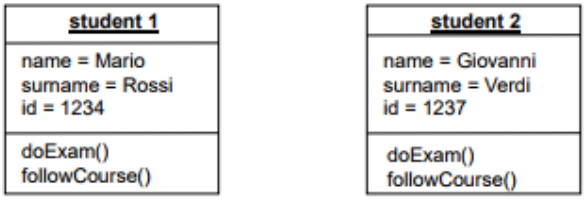

Figure 35 - Objects in the UML CD

4.2.2 Classes

A class is a descriptor of objects with similar properties. Classes can represent physical objects (e.g., Cars), organizational entities (e.g., Companies), descriptors of objects (e.g., Product Type), time intervals (e.g., Flight Duration), Time Events (e.g., Departure Time), geographical entities (e.g., Region), reports (e.g., Report of a sensor), transactions (e.g., Product sold).

The properties that are defined in a class can be subdivided in two types:

- **Attributes:** the name and type of an attribute is the same for all objects and can be described in the class; the value of an attribute may be different on each object, and cannot be described in the class (e.g., a student object has its proper name, surname and ID, which differ from other students' ones).
- **Operations:** operations are the same for all objects and can be described in the class; the operation will be applied to different objects, possibly with different results that are based on the object's own attribute values (e.g., students can take an exam, and follow a course; the function is the same for all students, but has different results based on who is the one performing it).

Objects are instances of a class. A Class Diagram models the classes of interest in a specific case.

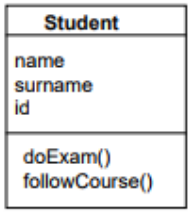

Figure 36 - A class in the UML CD

4.2.3 Relationships

In an Object Diagram, the objects that are part of a specific case of interest may be connected by **links**. A link models the association between objects.

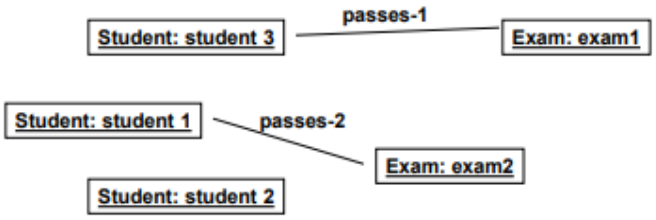

Figure 37 - Links in an Object Diagram

Classes may be interrelated to others, in different ways. **Relationships** provide a mean of describing logical connections between classes in a Class Diagram.

As objects are instances of classes, that provide for them a high level description, Relationships are descriptors of links that apply to classes. On the other hand, links are individual instances of relationships between classes.

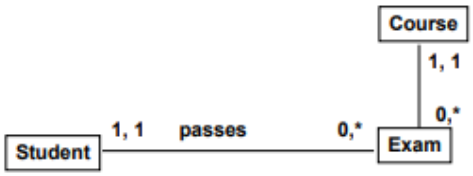

Figure 38 - Relationships in a class diagram

The existence of a relationship between two classes does not mean that all objects belonging to them have to be linked together.

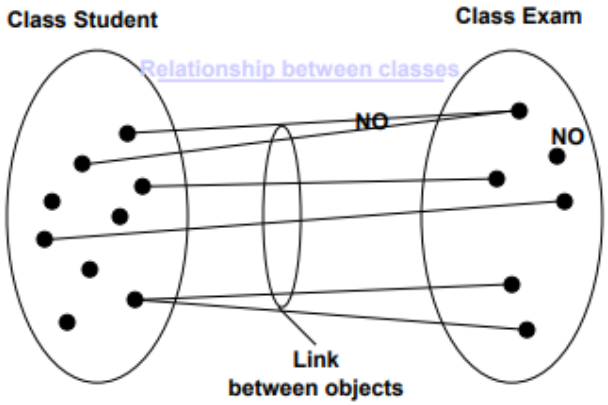

Figure 39 - Relationship and Links

Each relationship can be assigned a **name**, which specifies what is the kind of logical connection between classes. The two classes involved in the relationship can be assigned a **role**, which specifies what are the duties of objects of such class that form links described by the relationship.

Several types of relationships are contemplated by UML Class Diagrams, and are detailed in the following paragraphs.

4.2.4 Multiplicity

A multiplicity is a logical association, that depicts the cardinality of a class in the relationship with another one. Each relationship connecting two classes must specify the multiplicity of each end, that acts as a constraints on the maximum number of links that can exit or enter an object of that class, for that specific relationship. Different multiplicities that can be expressed in Class Diagrams are shown in figure 41. In general, when no multiplicity is expressed on one end of a relationship, the standard multiplicity (0…*) is assumed.

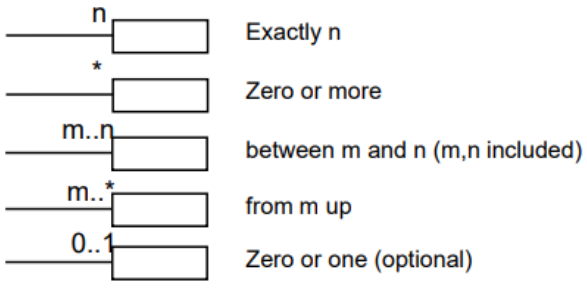

Figure 40 - Multiplicities in Class Diagrams

In the example in figure 38, we can see that the student class has a relationship with Exam, with 0 to * multiplicity: that means that each student can take from 0 to infinite exams. Instead, Exam is connected to student with a 1,1 multiplicity: it means that each Exam, in the system, must be taken by exactly one student. Also, each Exam belongs to one and only one course (1,1 multiplicity), while for any Course there may be 0 to infinite (0…* multiplicity) Exams.

4.2.5 Dependency

A dependency is the least formal among all the types of relationships. It represents that one class has a form of shallow dependency or connection on the one pointed by the arrowhead. Dependencies are represented with dashed lines, even though there is no inclusion or direct cooperation between the classes.

Dependencies can be used when changes in one of the classes would require changes in others (that are, hence, dependant classes).

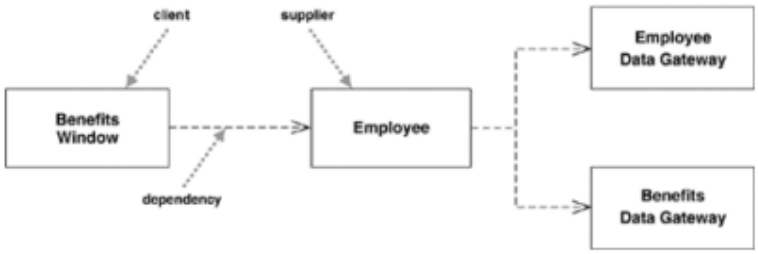

Figure 41 - Examples of Dependencies

4.2.6 Association

Associations are the most common relationships in UML class diagrams. They define dependency, but in a stronger way than a plain dependency relationship.

An association can be **directed**, if a clear active role can be identified for one of the classes at its ends, with the other having a passive role in the logical association. Typically, a directed association is represented with an arrow. If no clear active and passive roles are identified in the association, it is defined as **bidirectional**, and arrowtips on both ends can be (optionally) used to represent it. The name of an association, and the roles of the connected classes, reflect the fact that an association is directed or bidirectional: in general, directed associations have a verb phrase as a name.

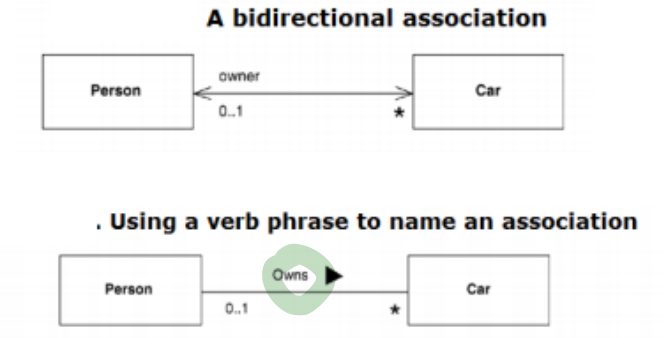

Figure 42 - Directed and bidirectional relationships

Associations can also aid in uncoupling the properties of objects from the object themselves: this may be useful especially when, for one or more of its attributes, an object may assume a value that belongs to a predefined set, or when one attribute would have, as a value, an array of a certain type of elements.

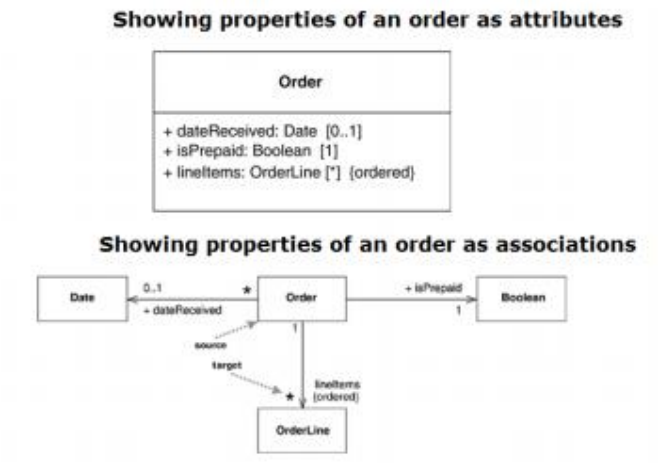

Figure 43 - Properties as attributes or relationships

4.2.7 Aggregation

An aggregation defines a class as a result of a collection of objects of another class (e.g., a Library can be made up of one or more Books). The aggregated classes are not so strongly dependent on the container, i.e. they can be instanced and have a lifecycle of their own even if a container does not exist. In the Class Diagram, an aggregation relationship identifies the parent (container) class with a diamond shape.

If B is part of A, it means that the objects described by class B can be attributes of objects described by A.

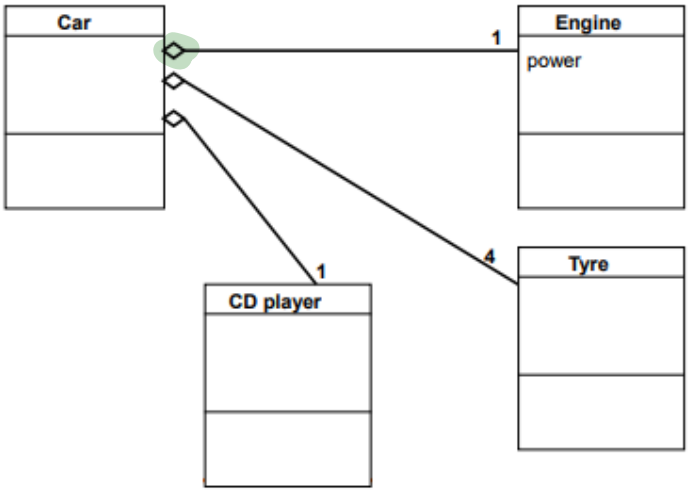

Figure 44 – Aggregation Example

4.2.8 Generalization

Also called specialization, or "is-a". One of the class of the relationship is a specific type of the parent class. A specializes B means that the **inheritance** mechanism exists between A and B: objects described by A have at least the same properties (attributes, operations) of objects described by B, and those properties do not

need to be repeated in the definition of A; objects described by A can have additional properties. It can be said that the properties in common between A and B are *inherited* by A.

The specialized class can be called *subclass*; the parent class can be called *superclass*. For instance, a Human is a subclass of Animal, that in turn is a subclass of Living Being; Human inherits the moving ability from Animal and the living ability from Living Being, but also has some class specific abilities (e.g., thinking).

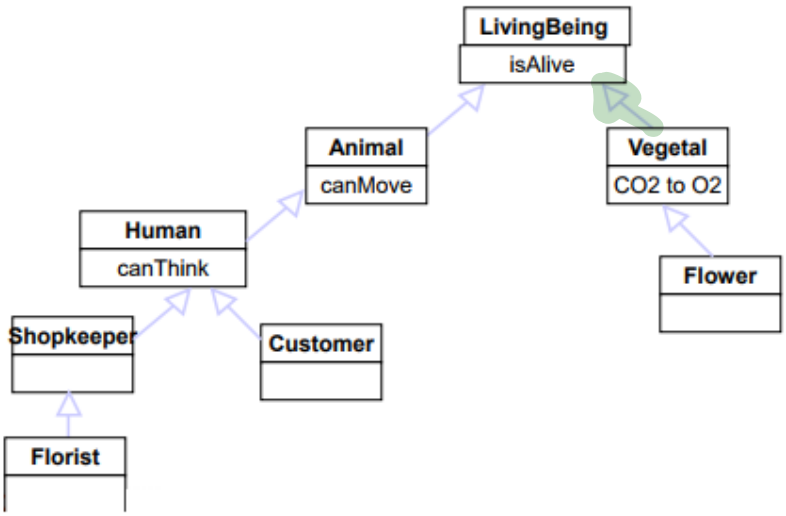

Figure 45 - Generalization Example

4.2.9 Notes

Notes can be attached to any element of the class diagram, either classes or any kind of relationship between classes. They can be used to give additional details about the attributes or the relationships in which the objects of a given class are involved, that are not easy to desume from the names or roles of the objects.

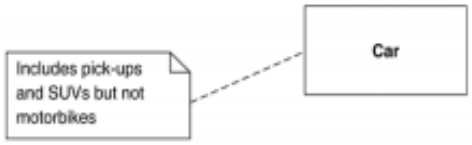

Figure 46 - Notes in Class Diagrams

4.2.10 DONTs for Class Diagrams

The following practices should be avoided in the definition of Class Diagrams, and of the attributes and relationships described inside them:

- Use plurals for class names (e.g., Classrooms instead of Classroom);
- Depict transient (dynamic) relationships: they are modeled in scenarios and sequence diagrams;
- Create loops (cycles);
- Overlook multiplicities;
- Forget to define a proper goal for the diagram, confounding system/software design or glossary;
- Use classes as attributes;
- Use attributes that represent many objects (arrays);
- Repeat a relationship starting from a class as one of its attributes.

4.3 Use Case Diagrams

Use Cases are a semi-formal notation that are used in the study of the application domain of a given system. They aid in identifying the boundaries between the system and the environment in which it is deployed, and in highlighting the interactions between the system and actors of the external world (that can be human stakeholders, or other systems).

Use Cases are useful to:

- Oblige the analysts to state well-defined boundaries between the system and the external world;
- Organize the functions of the system into individual elements, namely the use cases, on which the attention is focused;
- Supply a first basis for the specification of the system structure, from the user perspective.

The UML Use Case Diagram provides readability to all stakeholders involved in a given system, even customers / users and not only analysts and developes. Usually, Use Case Diagrams are defined before the Class Diagrams, to provide a first functional view of a software system.

UML Use Case Diagrams are composed by three main elements: actors, use cases and relationships.

4.3.1 Actors

An actor is any person, organization, or external system, that plays a role in one or more interaction with the system. An actor can either:

- Exchange information with the system;
- Supply input to the system;
- Receive output from the system.

Actors are typically represented as stick figures in UML Use Case Diagrams. They can be classified as primary or secondary actors:

- **Primary Actors** are actors using the system to achieve a goal.
- **Secondary Actors** are actors that the system needs assistance from to achieve the primary actor(s).

In the Class Diagram of the system, an Actor typically becomes a class.

Figure 47 – An Actor in UML UCD

4.3.2 Use Cases

A Use Case is based on the scenarios (i.e., sequences of steps describing interactions between a user and a system) of the system, being a set of scenarios tied by a common goal. In practice, a use case is a functional unit (functionality) of the system, in the form of a set of steps representing an interaction between the system and the actors.

Use cases are loosely mapped to functional requirements of the system. Each functional requirement can be linked to a complete use case, or to an individual scenario described by a use case, or to a step in a scenario (the mapping is not 1:1). In general, there is a subtle differences in the aims of requirements and use cases: the purpose of requirements is to support traceability of the system, and are finer grained than use cases; use cases are more high level, and their primary purpose is to understand how the system works.

In UML use cases are depicted as horizontal ellipses; by naming conventions, the use case names should always begin with a strong verb, and use domain technology. Primary use cases (or use cases that come first according to timing considerations) should be placed in the top-left corner of the complete Use Case Diagrams.

In the Class Diagram of the system, any use case must become one operation on a class, but can also generate several operations on multiple classes.

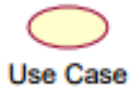

Figure 48 - An Use Case in UML UCD

4.3.3 Relationships

Several types of relationships are available in a Use Case Diagram, between one actor and one use case, two actors, or two use cases.

- **Associations** between an actor and a use case define the participation of the former in the latter. They can be represented with a simple straight line, but some adornments are allowed: multiplicities can define how many actors can participate in a given use cases, or how many instances of the same use case may be participated by the same actor; arrowheads indicate if the use case is triggered by the actor (i.e., the actor is *primary*) or vice-versa (i.e., the actor is *secondary*).

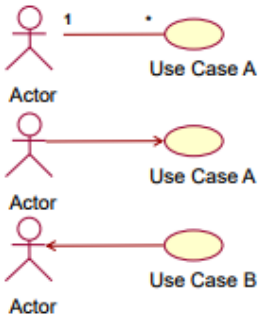

Figure 49 - Associations in UML UCD

- **Include** relationships, between two Use Cases, model that the functionality A is used in the context of the functionality B, being one of its phases. Typically, include relationships are used to extract fragments of use cases that are duplicated in multiple use cases. If a use case includes multiple others, they are both required in the including use case. The include relationship is represented as an oriented arrow, directed towards the included use case, with the <<include>> keyword. It can be also indicated as *uses* (e.g., in the following example, both password check and fingerprint scan are phases of the identity verification).

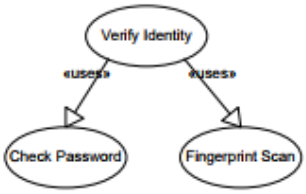

Figure 50 - Include relationship in UML UCD

- **Generalization** relationships, between two Use cases, model that the functionality B is a specialization of functionality A (for instance, a special case). Typically, generalization relationships are used to model a special case of an existing use case. The generalization relationship is represented with a directed arrow towards the generalized use case, without the addition of any keyword. Use cases that generalize others can stand on their own (e.g., in the following example, either password check or fingerprint scan can be performed, on their own, as a for of identity verification).

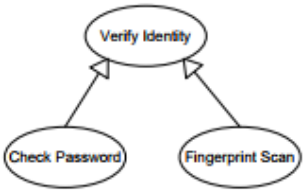

Figure 51 - Generalization between Use Cases in UML UCD

- **Generalization** relationships, between two Actors, model that an instance of the actor B can communicate with the same kinds of use-cases of an existing actor A, plus some use cases that are not available to B (e.g., in the case of an ATM system, a system administrator may be considered an extension of a normal user, since he is able to perform the normal withdrawal operations plus some additional specialized use cases).

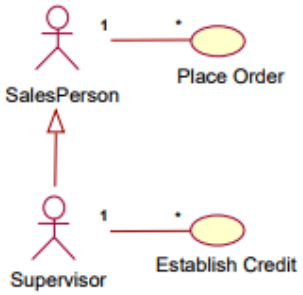

Figure 52 - Generalization between actors in UML UCD

- **Extension** relationships, between two Use Cases, model that an instance of the use case B may be augmented by the behaviour specialized by the use case A, in some particular conditions (e.g. errors). The behavior described by B is optional, and the location where the behavior must be executed is defined by an extension point referenced by the extend relationship.

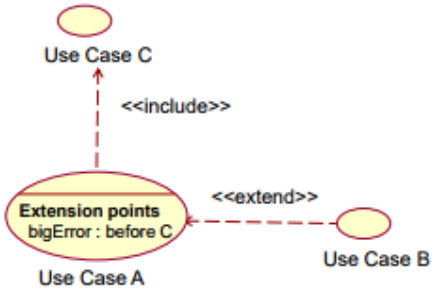

Figure 53 - Extend relationship in UML UCD

4.3.4 Example: student management

A student management system should provide the following functions:

- Students select courses;
- Professors update the list of available courses;
- Professors plan exams for each courses;
- Professors can access the list of students enrolled in a course;
- Professors perform exams, then record the issue of the exam for each student (pass/not pass, grade);
- All users should be authenticated.

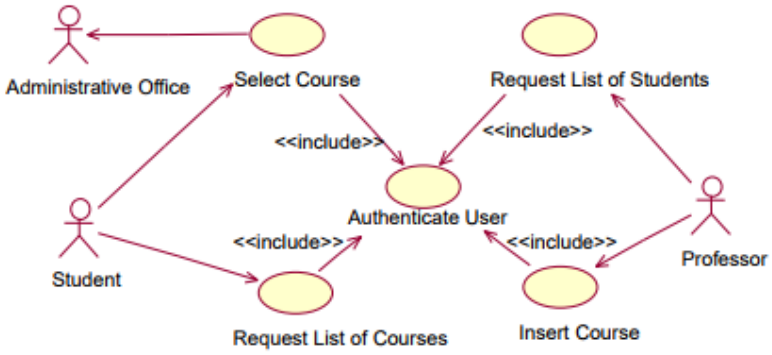

Figure 54 - UML UCD example: student management

In the solution shown in figure 54, three actors are considered as interacting with the system: students, professors, and the Administrative Office. The Administrative Office acts as a passive actor, since it does not initiate any use cases, and it is only involved in the selection of course initiated by the student. All the use cases (the ones initiated by students, and those initiated by professors) include a procedure of authentication, that is therefore described by a dedicated use case included by all others.

V ARCHITECTURE AND DESIGN

5.1 The Design Process

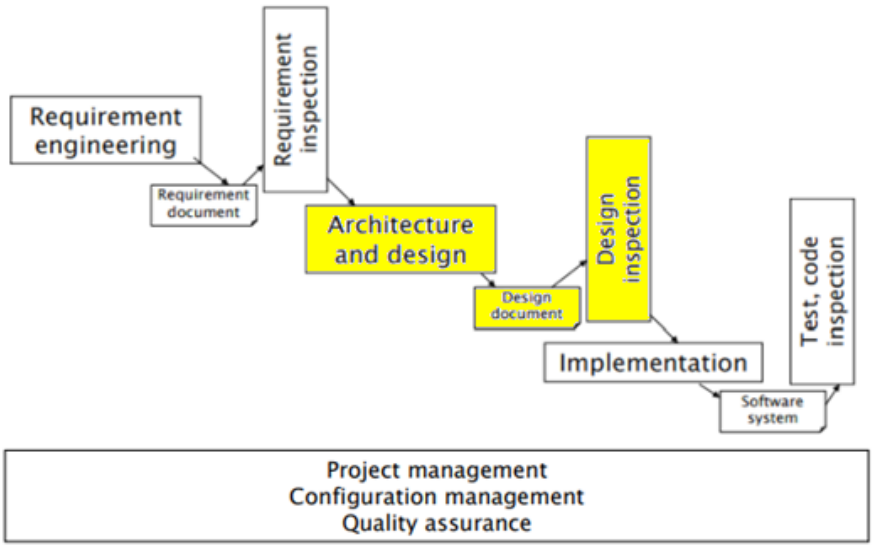

Figure 55 - Architecture and Design in the software process

project activity

While requirements specify what the system should do, Architecture and Design is the part of the software process involved in specifying how the system will be implemented. This phase is very important, because it aims at reducing errors in the coding part, and to avoid most logical bugs that may hamper the system: those errors must be found *early* in the development process. It is essential to define, analyze and evaluate design choices, making them explicit through the use of documents, in order to avoid errors that otherwise would be discovered only at coding time, when making design changes becomes difficult.

If the design is not defined, but the code is developed immediately based on the requirements, design choices are made *implicitly*, leading to a *late evaluation* of the design. The use of design docuents allows to make *explicit* design choices, that can be evaluated before coding.

The fundamental difference between Architecture and Design is that, while at the very end they have the same objectives, Architecture is higher level: defining the Architecture of a system means deciding which are its main components, how they will be controlled and which is the communication framework defined between them. Design, on the other hand, is lower level, and has the objecting of defining the internals of each component of the system. The design part is often performed using UML Class Diagrams, where each class is represented by a Class Entity in the diagrams.

Given one set of requirements, different designs (*design choices*) are available:

> *Example: for a mid sized car in price range between 10 to 20 thousand Euros, there may be hundreds of different design choices (i.e., models in the market); not all designs are equal.*

The translation of requirements into design is a creative process that cannot be completely formalized and that is strongly driven by individual skill and experience. Semiformal guidelines, based on previous experience, can be given about both architectural and design patterns.

5.1.1 Phases of the Requirements Process

A first characterization of the design activities can be made between the system design and software design side:

- **System Design:** decisions about the structure of the system (in terms of hardware and computing nodes, and their direct or indirect interconnections). According to the type of system being developed (e.g., for embedded systems) the system design must include decisions about component and connections in other technologies (electrical, electronic, mechanical).
- **Software Design:** upon a given system design, the software design activity defines all the software components that will ultimately be developed, their goals, and the logical interconnections and interfaces they provide each other.

The design process is basically made of three different parts: analysis, formalization, verification. This process can be abstracted, as a black box that takes in input the requirements document (with all functional and non functional requirements) and produces as output the design document (all the components and connections between them capable of satisfying the requirements extracted from the requirements document).

Architecture	Architectural Patterns
High Level Design	Design Patterns
Low Level Design	Classes implementation details
Formalization	Text, diagrams (UML)
Verification	Inspections

Table 14 - Phases of the Design Process

In more detail, the **Analysis stage** can be subdivided in three different phases:

- **Architecture** (about the whole system): definition of high level components and their interactions; selection of communication and coordination model about eleents; selection of architectural styles and patterns.
- **High Level Design** (about many classes): it is about the overall architecture of the entire system, the definition of all the modules that will be developed, the understanding of the flow of the systems. The outcome of the high level design is the definition of the class diagram for design, mapping classes with attributes and methods, defining bonds between classes as simple relationships (without defining how the relationship is implemented).
  Classes can be derived from the glossary, considering a class for each key entity, and from the context diagram (considering a class for each actor, either a physical device or subsystem, and defining the way each actor can interact with the system).
  In this phase of the design, design patterns are selected.
- **Low Level Design** (about single classes): the system overall view is broken down into modules, and the internal logic design is defined for each element, according to requirement specifications. A Low Level Design document allows the developers to code the software easily. As opposed to the High Level Design, the Low Level Design is usually used to specify how to implement the relationships between classes, going deeper in the design of each class.
  For each class that must be developed, the low level design also defines: type and privacy of each attribute; return type, parameters and privacy of each method; needed setters and getters.
  For each object, persistency (e.g., serialization to files or network, or insertion in databases) is defined.

In the **Formalization stage**, each part of the design is translated in a document, using either formal (UML) or semi-formal methodologies and languages.

In the **Verification stage**, techniques for the validation of the architecture and design are executed, to understand how the proposed design fits with the functional and non functional requirements taken from the requirements document.

	Internship Management	**Heating Control system**
Technical Domain	Web application	Embedded system
Architectural Choices	Client server Layered (database, application logic, presentation) Repository	Single computer Layered (sensor, application logic, presentation)
High Level Design (packages, classes, relationships) choices	Manyreused from glossary, added some (app logic level and presentation layer)	Many rused from glossary, added some (app logic layer and presentation layer)
Low Level Design (attributes, methods, relationships) choices	Common choices for implementing relationships	Common choices for implementing relationships

Table 15 - Examples of architecture and design choices

5.1.2 Design and Non Functional Properties

A can be characterized according to its functional and non functional properties:

- **Functional properties:** does the design support the functional requirements, as defined in the requirement documents?
- **Non Functional Properties:** does the design support the non functional requirements, as defined in the requirement documents?

There are some guidelines that can be followed in the design phase to support the non-functional requirements elicited in the requirements engineering phase:

- **Performance:** to enhance the performance of the system, the design phase should minimise the communications between different modules, and localize critical operations, using large rather than fine-grained components;
- **Security:** to enhance the security of the system, adopting a layered architecture allows to place the critical elements of the system in the inner layers;
- **Safety:** to enhance the safety of the system, safety-critical features should be localized in a small number of subsystems. As for security, a layered architecture helps in enhancing the safety of a system;
- **Availability:** to enhance the availability of the system, redundant components (i.e., different components with same functions, that can work one in place of another that stops functioning) and fault tolerance mechanism should be adopted by design;
- **Maintainability:** to enhance the maintainability of the system, components should be kept as small (fine-grained) as possible, and should provide highly replaceability.

It is evident that design choices cannot provide support to all non functional requirements at once: using large-grained components improves performance, but removes maintainability; introducing redundant components improve availability, but may make security assurance more difficult; at the same time, security-

related features require high amounts of communication between components, so they lead to a degradation of performance.

It is part of the design process to decide trade offs about which non functional properties have to be enforced. Possibly, trade offs can be decided at requirement time, and it is duty of the design phase to select the structure of the system accordingly.

Some non-functional properties are more specific to the design phase, like the following examples:

- **Testability:** support to testing provided by the system. Testability is strictly linked to the availability and complexity of the designed system. It contains the properties of Observability and Controllability.
- **Monitorability:** possibility of setting up activities capable of monitoring the system, its performance and its users. It is close to the Controllability property.
- **Interoperability:** a property of the interfaces provided by the system. Is it easy to change a component? Is it easy to let the system talk with other systems?
- **Scalability:** capability of the system to handle a growing amount of work (e.g., in terms of users to be served), and degree of incrementation of performance when resources are added to the system.
- **Deployability:** how easily and reliably the software can be deployed, from the development into the production environment (e.g., installation and upgrades of desktop software).

5.2 Notations for Formalization of Architecture

Different ways are available for formalizing the architecture of a software system, from informal ones, to semiformal (e.g., UML diagrams, providing both structural and dynamic views), to formal Architecture Description Languages (ADLs).

5.2.1 Box and Line Diagrams.

A box-and-line diagram depicts the components of the architectures as boxes, connected with lines. The notation is completely informal: there are no rules governing the diagrams. Each box corresponds to a physical component or a set of functions (subsystem). By nature, box-and-line diagrams are also very abstract, because the nature of the relationships between components (and the nature of the components themselves) is not clearly identified. Neither interfaces or externally-exposed properties of the sub-systems are shown.

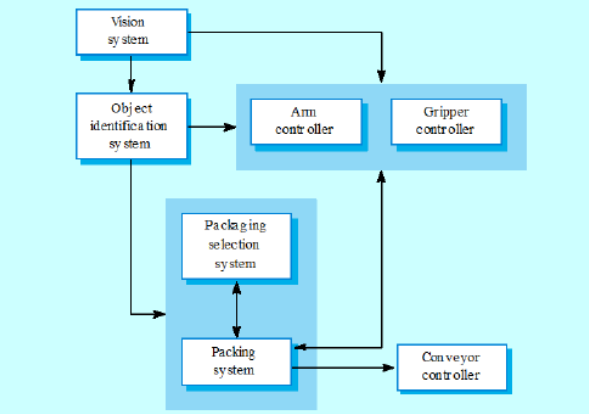

Figure 56 - Box-and-line diagram for a packing robot system

However, box-and-line diagrams may be a mean of communication with the stakeholders of a project, and may prove useful during project planning.

5.2.2 UML structural diagrams

UML provides means for presenting the structure of a system in a hierarchical, organized way: starting from the organization of packages in the system, then of classes in the packages, finally of attributes and methods in individual classes.

The **UML Package Diagram** provides a high level, structural view of the systems, depicting all the dependencies between the different packages in which the system is organized. A package is a collection of UML elements that are logically related; they are depicted as file folders, while dotted arrows represent dependencies between them. One package depends on another if changes in the other could cause changes in it.

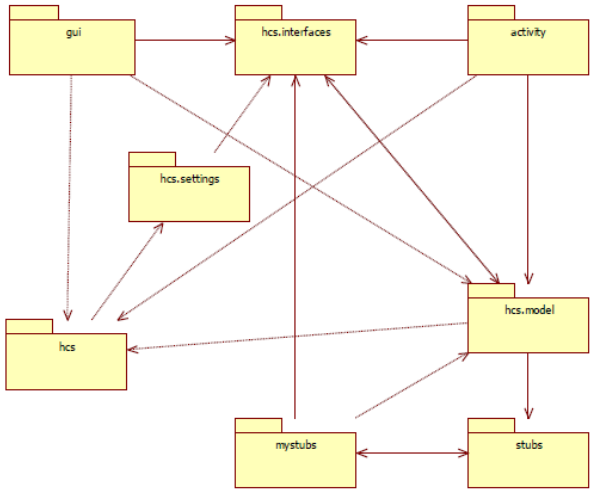

Figure 57 - Example of a UML Package Diagram

Then, in the context of the description of the architecture of the system, **Class Diagrams** can be used to describe the internal structure of each package, and a **Class Description** (i.e., the specification of each attribute and method of the class) can be provided for each class.

5.3 Architectural Patterns

Patterns are defined as reusable solutions, to recurring problems, in a defined context. The concept of pattern was originally proposed by *Christophe Alexander*, in the context of architecture (of buldings):

> *The pattern is, in short, at the sime time a thing, which happens in the world, and the rule which tells us how to create that thing and when we create it. It is both a process and a thing.*

In a nutshell, *patterns* are known, working ways of solving a problem. A distinction, in software engineering, may be made between Architectural Patterns (or styles), which are ways of building system-wide structures, and Design Patters, which leverage higher level mechanisms in describing the system.

Usually, a real system is not based on a single architectural pattern/style, but is influenced by many of them.

5.3.1 Layered model (Abstract Machine)

The idea of this pattern is to separate the software into layers or abstract machines, each one providing specific services. The constraint posed on layers is that only functions of adjacent ones can be used. An example of a layered architecture is the ISO/OSI model for network architecture, organized in seven different layers, from Physical (pertaining wires and cables) to Application (web processes and applications).

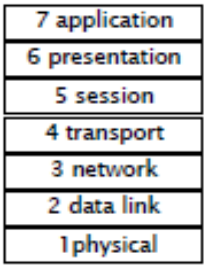

Figure 58 - The ISO/OSI model

From the architectural point of view, it forces the separation of concerns, since each layer is dedicated to solve one problem; moreover, the layered structure guarantees easy modifiability, since when a layer interface changes, only the adjacent layers are affected.

New implementations can be added, after having designed the system, by changing one layer only (e.g., a Bluetooth Module can be added after the design, without changing other layers).

Most of the times, software is organized in three layers: presentation (GUI or other ways of presenting the application to the users); application logic (logic that handles the business of an application); data (where the data of the application is collected, and the way to retrieve it, both for sensors, drivers and databases).

Figure 59 - 3-tier Architecture

5.3.2 Pipes and Filters

The situation in which the Pipes and Filters architecture is used, is when data streams have to be processed, in several different steps (commands) to perform in order to do a specific action. The output of the first step must be passed to the second, that takes it as its input, and so on: this is the only possible interaction between commands in the system.

In this scenario, it must be possible to recombine steps (i.e., change the order in which they are executed), and the information is confined to the individual steps (i.e., adjacent steps do not share info). The user storing data after each step of the pipes and filter architecture may result into errors and garbage.

In the Pipes and Filters architecture, four main elements can be identified:

- Data sources, in charge of providing inputs to other functions;
- Filters, in charge of executing manipulations over data;

- Pipes, in charge of connecting other elements: a data source to a filter, or two filters together (*in* filter, and *out* filter);
- Data sinks, which consume the output created by the system of pipes and filters.

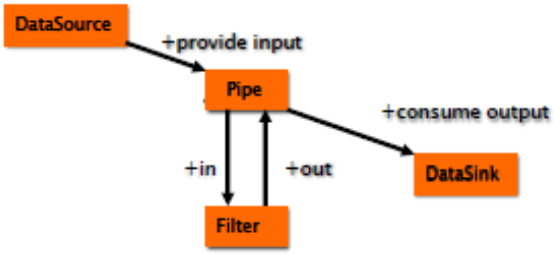

Figure 60 - Elements of the Pipes and Filters pattern

An example of the Pipes and Filters architecture is a Compiler, in which after the Input is given, a series of steps is performed: scanning, parsing, semantic analysis, and finally code generation.

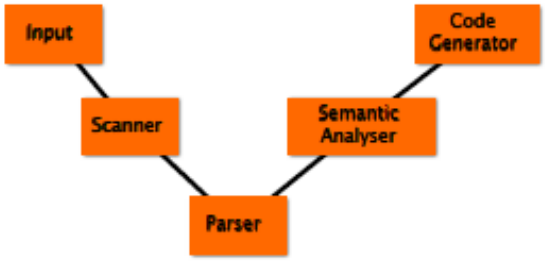

Figure 61 - Architecture of a compiler

Another example of the Pipes and Filters pattern is the UNIX shell, where each command (e.g., grep) can be considered as a filter, input and output files serve as data sources and sinks, and connecting characters (e.g., > or <) serve as pipes.

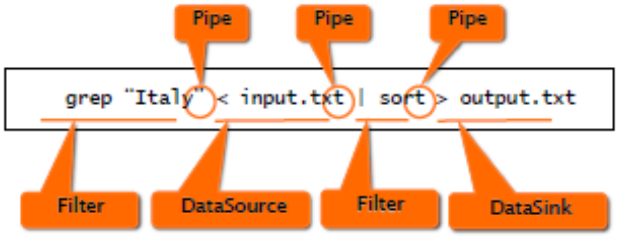

Figure 62 - Unix shell commands as Pipes and Filters example

5.3.3 Repository

Subsystems in an architecture must be able to exchange data, which can be done in two possible ways: the creation of a shared central database that can be accessed to all the sub-systems, or the creation of private databases for each sub-system, with explicit exchange of data between different sub-systems. Typically, when the amount of data to exchange is large, the repository model is used.

In the repository architectural pattern, the core is developed beforehand, and then many plugins can be attached to it. The core takes charge in offering an horizontal service to all the plugins, such that they do not need to be concerned about how data is produced. The sharing model is published as the repository schema.

The possibility of developing and adding many plugins in separate moments, independently from the core development process, is one of the main advantages of the repository architectural pattern: it ensures high modifiability and maintainability for the system. The centralized management of information ensures high level of safety, security, and backup management. High efficiency is guaranteed for sharing large amounts of data.

The main drawback of the repository pattern is that plug-ins can communicate only through the core software, and not directly between each other. The data model, also, must be shared among all plug-ins, hence some comrpomises cannot be avoided. The data evolution is then difficult and expensive.

An example for the repository architectural pattern is Eclpse, that is a container for many tools that can be installed or not.

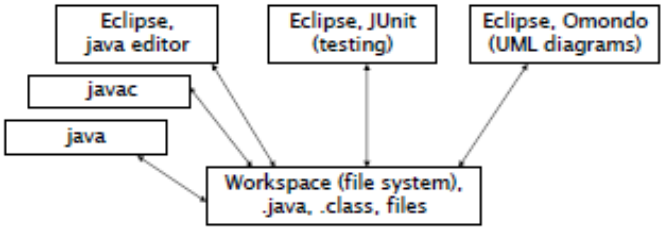

Figure 63 - Eclipse and Plugins

5.3.4 Client-server

In the client-server model, there are many stand-alone servers connected through a network, that are independent and that can be connected to many clients, to provide specific services. Both data and processing are disributed across such range of components.

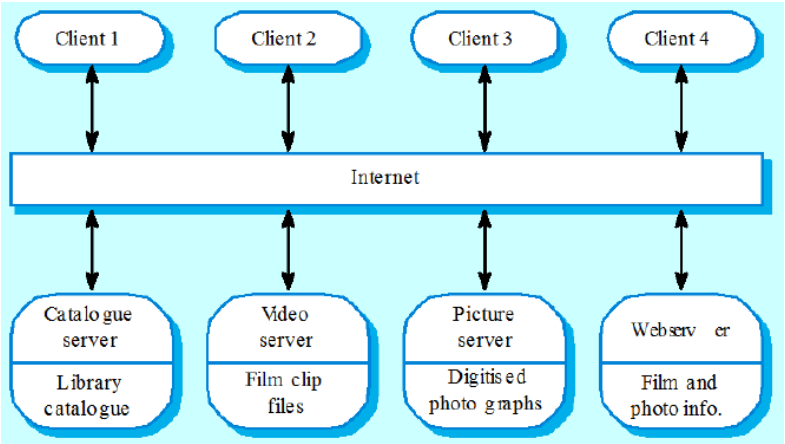

Figure 64 - Film and Picture Library

Server must agree onlyon the internet protocol, but the data model used by them can be different. Having different data models can be a drawback, since the data interchange between servers may become inefficient if data is organized in different ways. In addition to that, management is done separately in each server, thus introducing redundancy in the system. In a client-server architecture without central register of names and services it can be hard to find out what servers and services are available.

The main points of strength of a client-server architecture are a straightforward distribution of data, an effective use of networked system that allows to use cheap hardware, and the easiness on adding new servers or upgrading existing ones (scalability).

5.3.5 Broker

The Broker pattern is typically used in an environment with distributed and possibly heterogeneous components. The requirements suggesting the use of a broker pattern are: components should be able to access others remotely and independently from the location, and users should not see too many details about the implementation, and differences in their experience due to the components used; finally components can be changed at run-time.

The Broker solution considers a set of components that talk with a broker, that offers a common API (examples can be Trivago, Booking.com, etc.). A unique interface connects many heterogeneous serves and clients: this introduces a new layer in the middle, but guarantees the system enhanced flexibility (e.g., changing a server or a client does not lead to the necessity of modifying the whole system).

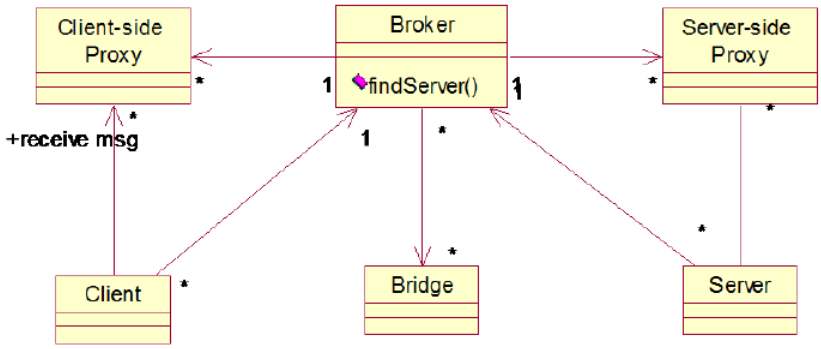

Figure 65 - Broker Pattern

5.3.6 MVC (Model-View-Controller)

The problem to address is to show data to user, and at the same time manage the changes to such data. A first approach to solve the problem should be using a single class; Model View Controller (MVC) pattern provide a finer way to approach the problem, in which the problem of representation is split in two parts (namely, the model and the view).

The Model View Controller is used mainly for interactive imperactions, with flexible interfaces provided to the human users, and with the same information that should be presented in different ways, or in different windows. All the windows should present consistent data, and the data may change at runtime. The Model View Controller pattern, in such situations, must ensure maintainability and portability for the system.

The pattern is based on the following elements:

- The **model**, which has the responsibility of managing the state (interfaces with the Database, or the file system); it is a description of data, which always remain the same.
- The **view**, which is responsible to render the data on the UI; the same data can be represented in different ways (e.g., multiple ways of representing the same data in a graph).
- The **controller**, which is responsible of handling events that are triggered from the UI, after interactions with the user. In fact, if a model changes, the change must be reflected by the view,

hence introducing a synchronization problem: such problem is solved by introducing a Controller, which is a handler for the modifications, and helps in synchronizing the Model and the View.

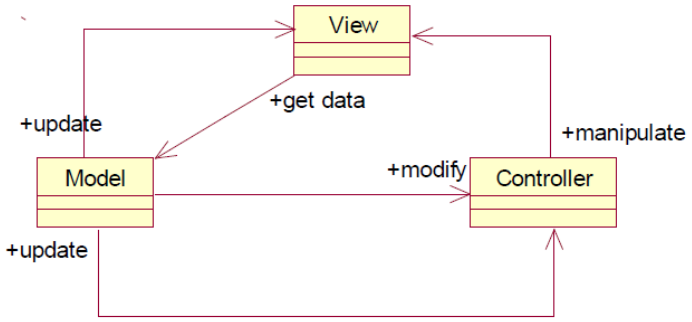

Figure 66 - Components of MVC

In systems realized using the MVC pattern, there is typically no predefined order of execution: the operations are performed in response to external events, triggered by the users e.g., mouse clicks on the interface). Event handling is serialized, with recognized events that are handled one at a time by the controller: to execute parallel operations, threads must be used.

The main advantage of using a MVC pattern is the separation of responsibilities among subsystems: manydifferent views are possible, and model and views can evolve independently, thus ensuring maintainability to the system. The main drawbacks of the MVC pattern is a high complexity, and hence a minor performance with respect to interfaces not realized in this way. However, today almost all User Interfaces are organized according to this pattern: given the high level idea of the MVC, an implementation is provided for almost any environment (Java, C#, Android, Ios, etc.).

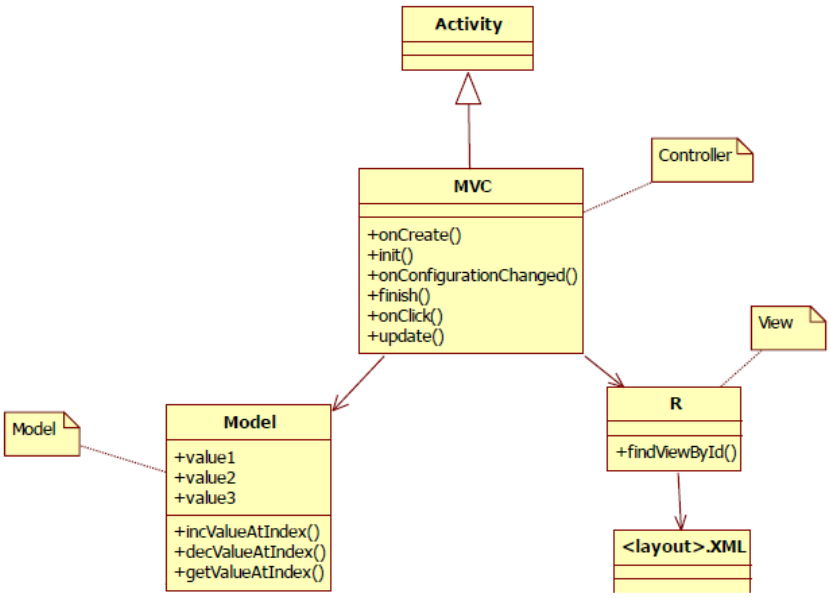

Figure 67 - MVC in Android

5.4.7 Microkernel

The microkernel pattern is used in a context in which several APIs insist on a common core, and hardware and software evolve continuously and independently.

Microkernel pattern separates a minimal functional core, from modules that provide extended functionalities, and user-specific parts. Only part of the core functionalities are included in the micro-kernel, with the rest moved to separate internal servers. External servers provide more complex functionalities, and are built on top of the functionalities provided by the microkernel.

The microkernel ensures good portability (only the microkernel must be adapted to the new environment) and high extensibility and flexibility; on the other hand, a significant amount of inter-process communication is required (less efficiency), and the design is far more complex than the one of a monolithic system.

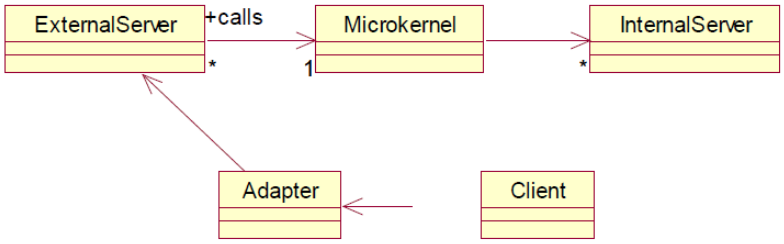

Figure 68 - Microkernel structure

5.4 Design Patterns

To design the system, design patterns can be considered. Design patterns are the most widespread category of pattern, and the first category of patterns proposed for software development. As opposed to architectural patterns, whichi impact the whole system and all the classes, design patterns describe the structure of components (that may be made by a single class, or a group of classes).

A design pattern can be defined as the description of communicating objects and classes, that are customized to solve a general design problem in a particular context. A design pattern names, abstracts and identifies the key aspects of a common design structure, that make it useful for creating a reusable object-oriented design.

Three different categories of design patterns exist, according to their purpose: creational patterns, structural patterns and behavioral patterns. Design patterns can also be characterized by their scope, if they are related to the description of classes or individual objects of the system.

5.4.1 Creational Patterns

Creational patterns describe techniques to manage the instantiation of new objects, with the aim of avoiding errors and creating too many low-level functions. Main creational patterns are detailed in the following.

- **Singleton pattern:** the context is a class representing a concept that requires a single instance. Without the adoption of the singleton pattern, the client could use that class in inappropriate ways, instancing many objects of that class. Using the singleton pattern, the client can use only one object of that given class. In the singleton pattern, the class itself is responsible for the control of its instantiation (that happens only once); an hidden private constructor ensures that the class can never be instantiated from the outside.

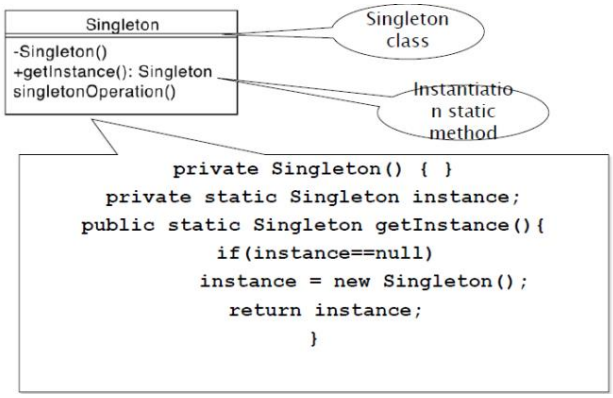

Figure 69 - singleton class

- **Abstract Factory pattern:** it is useful when an application must be developed for different operating systems. In the abstract factory pattern, a class is created dependently on which operating system the software is run, but the client does not see it: it only sees abstract classes, and does not know anything about which variant he is using or creating.

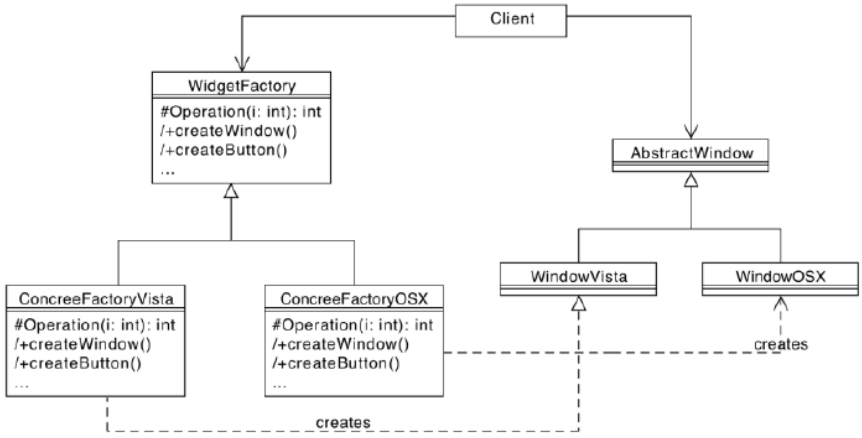

Figure 70 - Abstract Factory example

5.4.2 Structural Patterns

Structural patterns describe techniques on how to organize classes and objects in order to form larger structures. They are divided in structural *class* patterns, which use inheritance to compose interfaces, and structural *object* patterns, which describe how to compose objects to realize new functionalities. In the following, the main structural patterns are described.

- **Adapter pattern:** the adapter pattern is used when a class providing the required functionalities exists, but its interface is not compatible with the system. It converts the interface of the class, to match the expecting one, in order to integrate the class without modifying it: it proves useful especially when the source code of the class is not available, or the class isalready used somewhere else as it is. Adapter pattern is often used with device drivers, a case in which a single interface for all the device drivers can be desirable, but individual original interfaces of the drivers may be not compatible with the required one.
  In the Adapter pattern, the inheritance mechanism plays a fundamental role. It is the only example of structural class pattern.

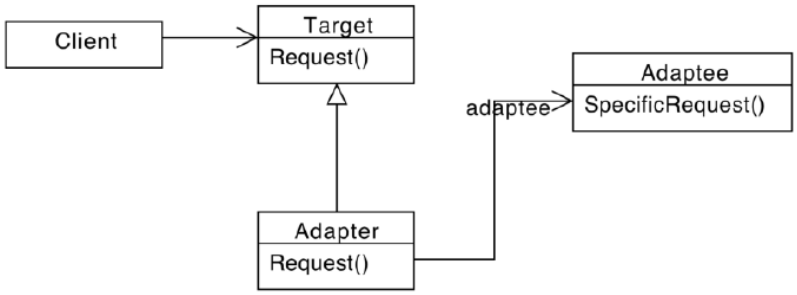

Figure 71 - Adapter pattern

An example of use of the Adapter pattern is the Java Listener Adapter: in Java, GUI events are handled by Listeners, and Listener classes need to implement Listener interfaces, with several methods that should all be implemented.

- **Composite Pattern:** it is used in order to allow the user to logically group objects, by creating trees and represent part-whole hierarchies. Two types of objects are described: Composites and Leaves. They both share the same parent class. Composite objects, moreover, can group other objects and, occasionally, perform operations on them.

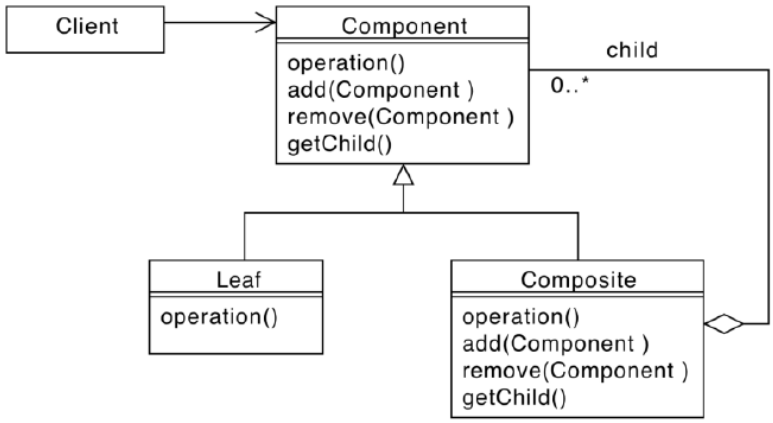

Figure 72 - Composite Pattern

An example of usage of the Composite pattern can be a computer system, with a parent class, *Equipment*, deining the method watt(), used to compute the power consumption. Leaves objects can be *Hard Drive*, *Motherboard, CPU,* and so on; a Composite object, including leaves, can be *Server*. By applying the composite pattern, all the leaves can be added to the composite one, and by calling server.watt() the total power, for all the components, can be computed. Moreover, *serverFarm* can be defined, to which several servers can be added.
The composite pattern can be used also for the representation of Graphical User Interfaces, which expose to the user components that contain other components inside.

- **Façade Pattern:** this pattern is used when a functionality is provided by a complex group of classes (with interfaces and associations) and it is desirable to use the classes without being exposed to their details. With the facade pattern, the client only sees a facade class, with the entire system being hidden behind it. Since the user sees only a simplified version of the system, the whole system results cleaner and easier to use. Facade pattern allows the user not to worry about the hidden complexity managed by the facade class. Only users needing more customization of the system to their needs

must access the lower levels. Moreover, using a facade class enables the lower level components to be changed without affecting the user which is accessing the facade only (enhanced modifiability of the system).

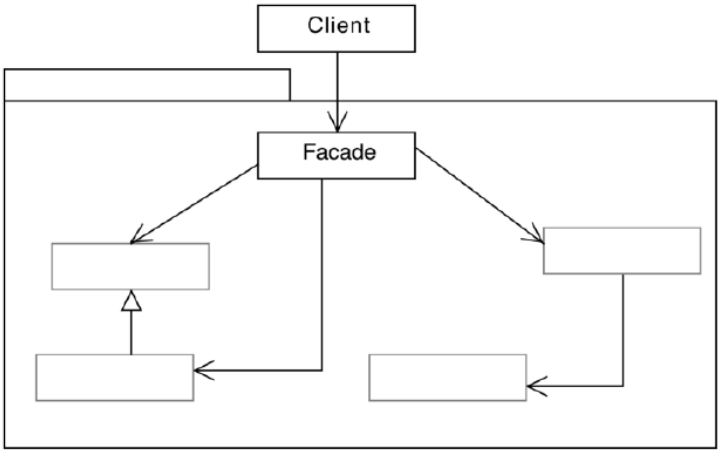

Figure 73 - Facade pattern

5.4.3 Behavioral Patterns

Behavioral patterns describe the behavior of objects, the algorithms governing them and the responsibilities between them. They are more patterns of communication between objects, shifting away from complex control flows description to concentrate on the way objects are interconnected. Main behavioral patterns are described in the following.

- **Observer pattern:** it is used in a context in which changes in one object may influence one or more other objects, i.e. there is high coupling between objects. The number of type of objects influenced by changes in others may not be nown in advance. The observer pattern creates an abstract coupling between subject and observer classes, that may be then instantiated by a variable number of concrete subjects and observers.

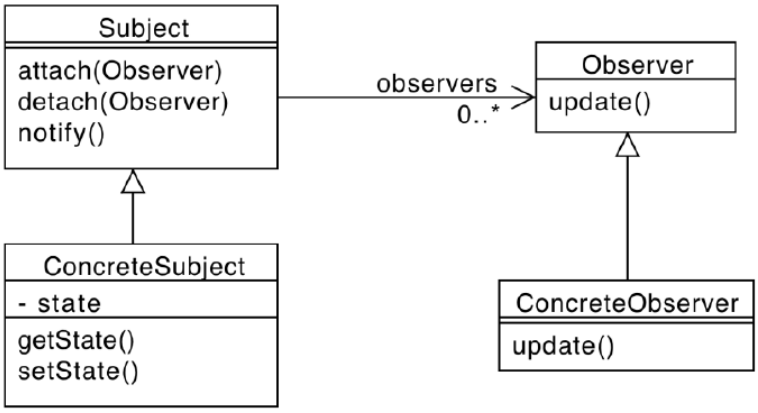

Figure 74 - Observer pattern

- **Template Method Pattern:** it is used when an algorithm or behavior has a stable core, and several variations at given points, and avoids the maintenance or implementation of several almost identical pieces of code. In the template method pattern, an Abstract Class contains the core algorithm, with

abstract primitive operations; Concrete Classes implementing the Abstract Class define variants of the algorithm, without having to implement the primitive operations.

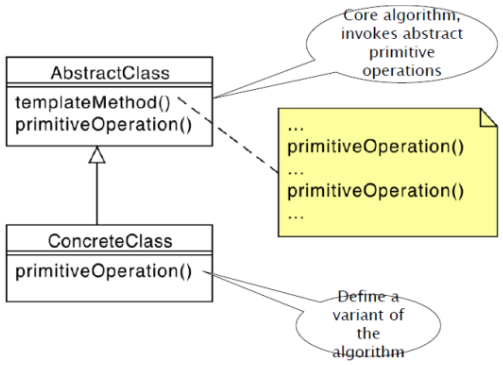

Figure 75 - Template Method Pattern

- **Strategy pattern:** it is used when there are many algorithms that have a stable core and several behavioral variations, and it is desired to use one or another without changing the way to call it (since multiple conditional constructs tangle the code). In these cases, an interface for the algorithm is defined, and then many classes can implement it: this way, one or another class can be used without changing the caller (e.g., the Comparator in Java). There is a fundamental difference between Abstract Factory pattern and Strategy pattern: Abstract Factory describes the way objects are created, which involves the class itself with its own set of attributes and methods; Strategy pattern describes the way objects behave, while calling the same method.
  Using the strategy pattern allows to avoid conditional statements and to organize algorithms in families, providing run-time binding of the proper algorithm to caller objects. The main drawbacks of the pattern are a communication overhead and an increased number of objects defined.

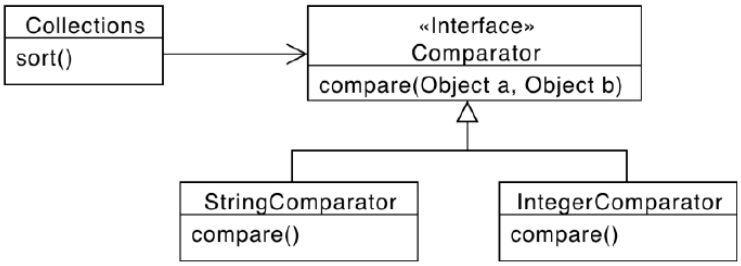

Figure 76 - Strategy pattern

5.5 Verification

After the architecture and design are decided for a given system, they must be verified with respect to the requirements that are specified in the requirements document, that serves as input to the design phase. Different practices are used to verify the conformity of the design to functional and non-functional requirements.

5.5.1 Traceability Matrix

The traceability matrix is a matrix having a row for any functional requirement (as they are defined in the requirementsdocument) and a column for each function in the classes of the software design. The matrix allows to keep track of which requirements are implemented by each function of the design.

If the system is designed properly, each functional requirement must be supported by at least one function in one class in the design. The more complex the requirements, the more member functions will be needed by the system.

	AwayManagementStrategy	Boiler	CRoom	DefaultHouseSettings	Env	Environment	HouseController	InvalidTimeException	PhysBoiler	PresenceManagementStrategy	Room	RoomManagementStrategy	RoomSettings	SetRoomParametersActivity	SetRoomParametersDialog	XMLSettings
Temp-UR-F1											X		X	X	X	X
Temp-UR-F2											X		X	X	X	X
Temp-UR-F3											X		X	X	X	X
Temp-UR-F4											X		X	X	X	X
Temp-UR-F5											X		X	X	X	X
Temp-UR-F6		X	X		X	X	X		X	X	X	X				
Temp-UR-F7	X	X	X		X	X	X		X		X	X				
Temp-UR-F8		X	X		X	X	X		X	X	X	X				
Temp-UR-F9		X	X		X	X	X		X	X	X	X				
Temp-UR-F10	X	X	X		X	X	X		X		X	X				
Temp-UR-F11								X							X	
Temp-UR-F12				X									X	X		
Temp-UR-F13	X	X	X		X	X	X		X		X	X				
Temp-UR-F14	X		X		X	X	X			X	X	X				
Temp-UR-F15			X		X	X	X			X	X	X				
Temp-UR-F16	X									X						
Temp-UR-F17			X	X			X				X					X
Temp-UR-F18		X					X		X							
UR-Inv 1	X	X	X		X	X	X		X		X	X				
UR-Inv 2		X	X		X	X	X		X	X	X	X				

Figure 77 - Traceability Matrix

5.4.2 Scenarios

Each scenario, defined in the requirements document, must be feasible with the architecture and design chosen for the system.

It must be possible to define a sequence of calls to member functions of the classesin the software design, that match each scenario.

5.6 Design of Graphical User Interfaces

User interface design is a part of the software process that is more suited to the context of enterprise and commercial software, quite never for the embedded market. Systems may provide an interface to their users, to obtain input from them, and giving them proper outputs.

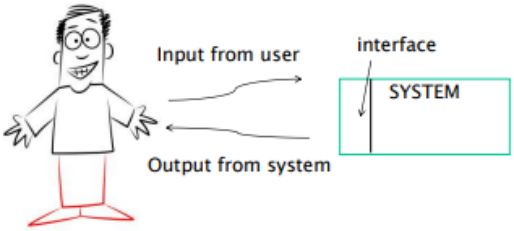

Figure 78 - Interface with the user

The user interface design defines the interactions with the user: sight, hearing, touch, voice, eye tracking, position and gestures, and whatever can be useful to understand the user needs, or to produce feedback to him (examples are given in table 16).

User	System
Sight	Screen, printer, glasses
Hearing	Noise, music, voice synthesis
Touch	Glove
Hands	Keyboard, mouse, touchscreen
Voice	Voice recognition
Eyes	Eye tracking
Position, Gesture	Gesture Recognition

Table 16 - Interaction means

There are three dimensions with which the user interface design can be characterized: a good user interface should have low complexity for usage, low complexity of functions, and low complexity of feature, while still providing access to a wide range of features. Often, the simplicity of the user interface is limited by its efficiency.

Nowadays, there are three possible UI platforms: web interfaces, desktop interfaces, and mobile interfaces. The current trend on desktop computing is to move on browsers, because it does not require users to install anything. Instead, on mobile the trend is the opposite, with installation of apps and rare use of browsers.

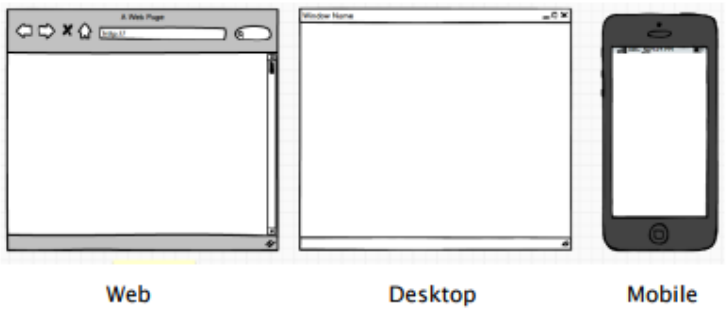

Figure 79 - Possible User Interfaces

The basic principles in designing user interfaces are:

- **Ergonomy:** comfort and safety in using the user interface; adaptability to the user's needs; usability, i.e. the ease of use of the GUI;
- **Emotional design:** the interaction with the GUI should cause positive emotions in the users;
- **User Experience (UX):** considers all other properties of a GUI, i.e. usability, emotions and values of the presented eleents.
- **Transparency:** no emphasis on the used technology, but instead on the functionalities provided to the user;
- **Feedback:** the design of the user interface is user-centered, i.e. decisions about the user interfaces are not based on personal opinions of the developers, but on feedback from real users.

5.6.1 Techniques of User Centered Design

Activity #	Activity	Techniques
1	Identify the users	Personas
2	Define requirements	Focus groups, Questionnaires, Interviews, Ethnographics
3	Define system and interaction	Prototypes
4	In lab tests	Ethnographics, Interviews
5	In field tests	A/B testing, Measurements

Table 17 - Activities of User Centered Design

User Centered Design (UCD) process is composed by several activities, and it starts from the identification of users (often they coincide with the stakeholders).

- ***Personas*** must be identified, to cluster the types of customers, taking into account their lifestyles and possible life scenarios that identify an interaction with the system.

  Example: persona 1, male, middle age, professional, high income, married with children; persona2, female, young, student, low income, not married.

The second phase of UCD is the definition of requirements (in a semi-formal way) by applying one of many sociological techniques (focus groups, questionnaires, interviews, ethnographics):

- A **focus group** works by taking some personas and putting them in a room, to discuss the system with a moderator. The group of people is homogeneous, and the discussion can be open or guided by a script. It is a long practice, that requires group interaction.
- A **Questionnaire** is a set of written questions, with open or closed answer, with the aim of gathering information through a statistical analysis of the answers.
- An **Interview** is a deep discussion, between a possible user of the system (that can be identified in a persona) and an interviewer, who asks a set of scripted questions and reports a log of the answers.
- **Ethnographics** are observations of facts, behaviors habits and tastes of a group of people or a population; in this case the researcher is "hidden" inside the environment of the possible users, to monitor their characteristics. Collecting ethnographics is an expensive and long practice, and may be perceived as invasive by people being studied.

In the third UCD phase, the system is prototyped, in order to be testable by users and obtain their feedback. Some alternatives exists for the creation of prototypes:

- **Low fidelity prototypes:** design of the user interface with paper and pencils, sketches, storyboards and post its. Feedback about the low fi prototype is given by cognitive and ergonomy experts, which can apply checklists and their experience to identify possible issues in the GUI design.
- **High fidelity prototypes:** creation of computer executive mock-ups, with general-purpose instruments (e.g., PowerPoint), or specialized software (e.g., Balsamiq). Feedback about the high fidelity prototype is obtained in lab, making people try the prototype and gathering their response through focus groups, interviews or ethnographics.

Finally, when the feedback about the prototypes has been obtained, an evaluation can be obtained about the final system. Measures about the usage of the system are collected (e.g., which parts the user uses the most, and which are ignored; how much time is spent on each function; how and when errors are made and so on). A common technique that can be used in this phase of feedbac gathering is the **A/B test:** it consists in giving 50% of users the variant A, and 50% users the variant B of the UI, and then observe a measure on the result, e.g. how many people used and appreciated the variant.

Figure 80 - A/B tests

VI VERIFICATION AND VALIDATION

6.1 Definitions and Concepts

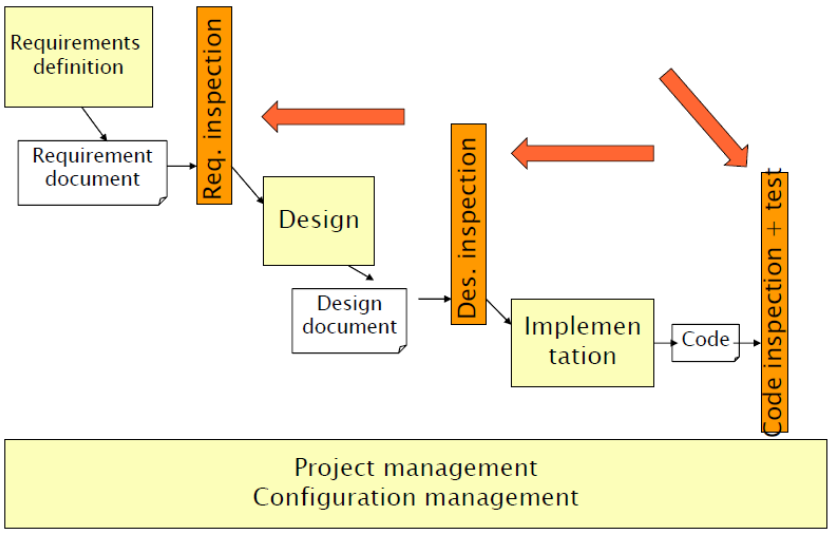

Figure 81 - V&V Activities

Verification and Validation involve two main activities:

- **Validation**: it answers the question: is the right software syste? Validation is about checking the effectiveness and reliability of the system. It ust be performed by the end users and stakeholders, and cannot be done autoatically.
- **Verification**: it answer the question: is the sotware system right? It checks the efficiency and the internal correctness of the system. It must be done by the software company, to check whether the system respects its constraints.

While passing the verification phase made by the software company, a software system may still not pass validation: a software system may be inherently correct and working, but still not compliant to the real needs of the involved stakeholders.

Verification must be done after each phase of the software development. Requirements, design and implementation must be checked, to verify that what has been done so far is correct. If the system to be developed has to be placed in a safety critical environment, validation becomes significantly more important.

In general, four main ways for verifying a software system are possible:

- **Testing:** checks if functional units or the whole system work correctly. It cannot be appliable to the requirements document;
- **Inspections:** manual checks and readings of documents, to check if they are correct;
- **Prototyping:** creation of a fast version of the software, that satisfies the most important requirements, in order to check whether it is the required system (e.g., MyBalsamiq, with which a GUI prototype can be built);
- **Static Analysis**: analysis of the source code, to detect errors in it early.

On the other hand, Validation (with stakeholders) is applied to the requirement document (to check that the elicited requirements correspond to the real needs of who is commissioning the system) and to the final system.

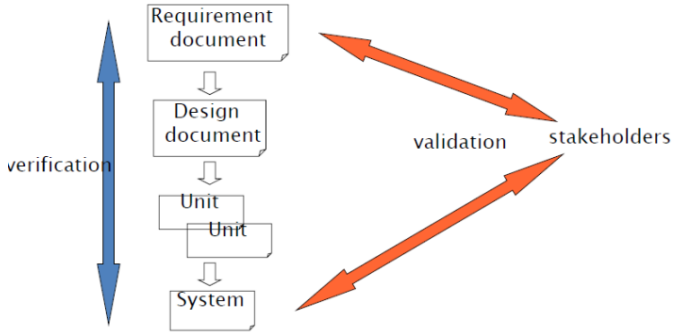

Figure 82 - Where V&V is performed

For a big project, the main problem is the validation and verification of requirements. Often, the requirements do not contemplate the real user needs, and thus they may be the source of the most expensive issues of the software project. To overcome this problem, the use of a glossary is very important, but it is not enough.

The cost of fixing defects is lower in the early stages of development, and it becomes very high when the system is already deployed. Therefore, it is important to detect and fix defects in early stages, to avoid expensive operations later.

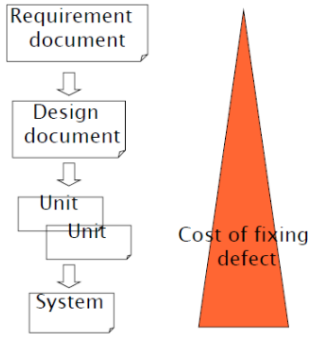

Figure 83 - Cost of fixing defects

6.1.1 Failures, Faults and defects

A **Failure** is an execution event where the software behaves in an unexpected way (there are some errors during the execution of the software). Each failure can be caused by one or many faults.

A **Fault** is a feature of the system (software) that causes a failure; it may be due to an error in software or to incomplete/incorrect requirements. A fault can cause 0 to many failures: therefore, a fault can be hidden to the user if it is never executed and never generates failures (e.g., faults in a functionality of a smartphone that is never used, like the radio).

A **Defect** can either transform in a failure or a fault. A defect is human-based, and must be discovered to be removed. The discovery can occur at every stage of the software life, from early stages to final ones. In some

cases, the defect is never discovered, meaning usually that it does not cause any fault or failure. A defect is characterized by an insertion activity (or phase), a possible discovery and then a removal activity (or phase).

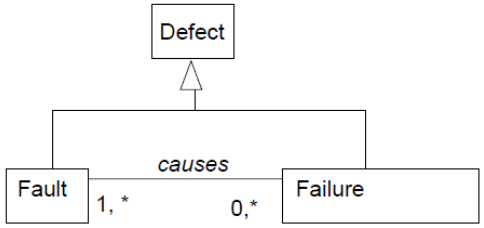

Figure 84 - Defects, faults and failures

The best software is one that is correct, having no faults. A reliable software, on the other hand, is one with no failures. The basic goal of Verification and Validation activities is to minimize the number of defectsinserted in the software. This number cannot be zero, due to the inherent complexity of software. Hence, the actual goal of V&V activities is to mximize the number of defects discovered and removered, and to minimize the time span between the insertion of a defect, and its discover and removal.

The number of inserted defects, in each phase of the development process, can be greater than the number of removed ones, or vice-versa. The goal of Verification and Validation is to equalize the amount of defect injection and defect removal, as early as possible.

The longer is the insert-remove, delay, the higher is the cost of removing the defect: this is known as the **Rework Problem**. A huge quantity of developing time is used to fix defects: such time is called rework time. According to Boehm (1987) and Boehm and Basili (2001) avoidable rework accounts for 40-50% of development.

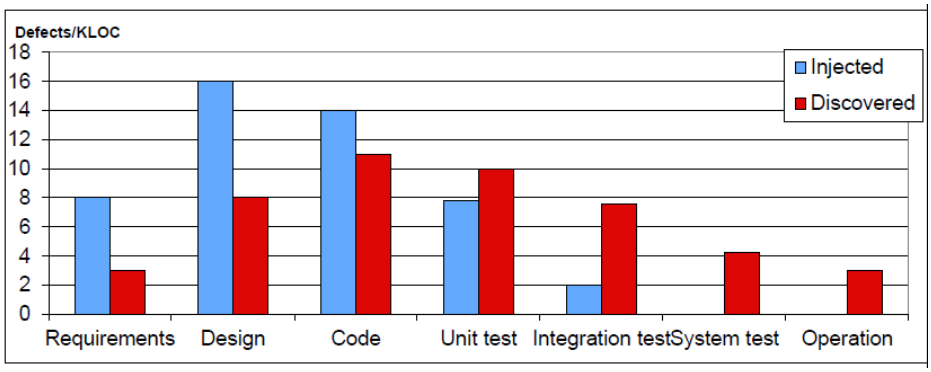

Figure 85 - Injection and removal of defects by phase: typical scenario

6.2 Inspections

An inspection is a static V&V technique, used to verify the requirements document or the developed code. Performing an inspection means reading a document (or code) alone, and then repeating the reading with a group of people (typically more than three) that include the author of the document. The goal of an inspection is to find defects in the document. Inspection is a formal techniques, that has several codified steps to follow; different variants of inspections exist (e.g., different reading techniques, walkthroughs, reviews).

Inspections can be applied to any kind of document (requirements document, design documents, test cases, and so on) and also to code, without being in the execution environment. The main advantage of inspections is their effectivenes: they reuse experience and knowledge of people about the domain and technologies, and have a global view on the software (as opposed to test, which can find one defect at a time). However, inspections are suitable mainly for functional aspects, and they require many effort and calendar time to be performed. Inspections and testing are compleentary techniques: they should both be used in V&V.

There are many types of defects that can be intercepted by inspections, and that can be searched type by type. Past experience can be leveraged, creating a checklist based on past defects information; the checklist consists in a set of evidences of defect, that are divided by defect class.

Some prerequisites are needed for a successful inspection: first of all, there must be commitment from management, and the inspection technique must be seen as a valuable effort. The approach to inspections must be constructive, and the results must not be used to evaluate the people who produced the inspected documents: inspections are not a "kill the author" game. The inspection process has to be focused and productive, aimed at producing the best possible document: meetings must not be seen as simple, relaxed chats.

6.2.1 Roles in an Inspection

Four main roles can be identified for an inspection:

- **Moderator:** leads the inspection process, and principally the inspection meeting. He is in charge of selecting the participants to the inspection, and preparating the material. Usually, he is not part of the process that has produced the documents to be inspected (to guarantee impartiality);
- **Readers:** they are in charge of reading the documents to be inspected. One of the main issues of the inspections is that every person who reads the document may find the same defects. To make up for this issue, a perspective-based reading can be assigned, with each reader starting from a different perspective (e.g., user, designer, tester). Each reader in an inspection should be asked to produce a document at the end, in order to encourage a more careful and accurate reading.
- **Author:** the creator of the document under inspection. He participates to the inspection process to answer questions that may arise from readers or from the moderators.
- **Scribe**: in charge of writing the inspection log, documenting the inspection procedure and the defects that have been found.

6.2.2 Fagan Inspection Process

1	Planning	Select team, arrange materials, schedule dates
2	Overview	Present process and product
3	Preparation	Become familiar with the product
4	Inspection Meeting	Team analysis to find defects
5	Rework	Correct defects
6	Follow-Up	Verify fixes, collect data

Table 18 - Fagan Inspection Process phases

The Fagan Inspection process has been invented in the early 70s by ichael Fagan, based on the production engineering inspection. It is a rigorous and formal inspection technique, applicable to all documents and with a very high defect detection effectiveness.

In the following, the phases of a Fagan Inspection are detailed:

- **Planning:** it is the zero stage of the inspection. The moderator chooses the participants, schedules the events of the inspection, and the budget dedicated to it.
- **Overview:** first phase of the meet-up, in which the moderator presents the inspection goal to the participants and the documents to be inspected. The benefits guaranteed by the inspection process are remarked.
- **Preparation:** each participant should read the documents individually, obtaining familiarity with the product, and apply the inspection technique.
- **Meeting:** the group reeds together the document, and discusses the issues found in it. The group finds agreement on problems. The scribe logs the problems, while the modreator keeps the focus and pace of the meeting, and breaks long discussions between the participants.
- **Rework:** the author, which is part of the team performing the inspection, fixes the found defects/problems.
- **Follow-up:** repeat the inspection to find further problems, or close and pass to the next phase of verification and validation.

6.2.3 Inspection Techniques

The Fagan Inspection Process formalizes the various steps that must be followed during the inspection of a given document. According to the type of document inspected, different techniques can be used:

- **Defect Taxonomies:** they are applicable to code, requirements, and design documents, and list categories of common defects that can be found. Several defect taxonomies are available in literature.

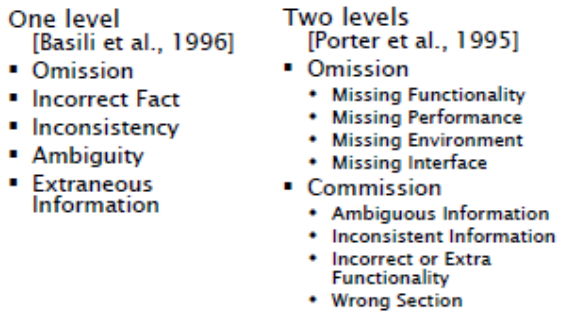

Figure 86 - Defect Taxonomies for Requirements

- **Checklists:** they are applicable to code, requirements and design, and list questions and control that can be formulated about the document. Checklists are typically based on past defect information and previous results of inspections, and refine existing defect taxonomies. Checklists for code also depend on the programming language used.
- **Scenario-based reading:** inspectors create an appropriate abstraction that helps understanding the product. Inspectors answer a series of questions tailored to different scenarios, each focusing on specific issues.
- **Defect-based reading:** a form of scenario-based reading, focused in the detection of defects in requirements that are expressed in formal notations. Each scenario is based on a specific class of defects (e.g., data type inconsistencies, incorrect functionalities, ambiguity or missing functionalities).
- **Perspective-based reading:** a form of scenario-based reading, focused in the detection of defects in requirements that are expressed in natural language (and later extended for design and source code).

Each scenario focuses on reviewing the document from the point of view of a specific stakeholder: users¸designers, testers. Each reviewer, at the end of the inspection process, produces a report that is in line with the perspective he is interpreting.

6.3 Testing

Testing is a dynamic V&V practice, that verifies the correctness of the parts of the software system (unit testing) or of the system as a whole (system testing). Testing can be formally defined as *the process of operating a system or component under specified conditions, observing or recording the results to detect the differences between actual and required behaviour.*

The goal of testing is not to ensure that the system is free of failures but, on the contrary, to find defects in software products: a testing process is successful if it reveals defect. Testing is a disruptive process.

Testing is different from debugging. The debugging practice has the objective of finding why defects are generated, to understand which faults causes a given failure and try to remove it. Testing aims to find the failure itself. Often, testers and debuggers belong to distinct groups of the company, and the testing and debugging activities are performed at different times. The output of testing can be the input of a debugging activity, with the aim of trying to fix the failures that have been found; after fixes are performed in the debugging phase, testing has to be performed again, to find if the failure has been fixed or if other failures have been added to the software.

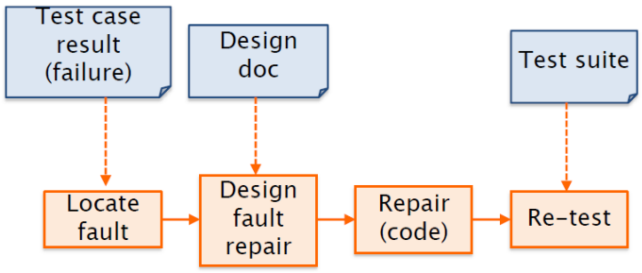

Figure 87 - Testing and debugging activities

6.3.1 Testing Concepts and Definitions

A **Test Case** is a defined stimulus that can be applied to the executable (system or unit), described by a name, composed by an input (or a sequence of inputs), an expected output, and a set of constraints that define the context in which the test case has to be executed (e.g., type of operating system, database or GUI used, and so on). A **Test suite** is a collection of related test cases (e.g., pertaining the same unit of the system).

The **Test Log** documents the execution of a test case, with time and date of application, actual output and outcome (pass, if the output is the expected, not pass otherwise). Test cases should be traceable, in the sense that it must be simple to identify them and understand to which requirements they refer, and which code fragments they are testing. Traceability simplifies significantly error fixing (and debugging) work, because it helps finding where the errors are, and what are the reasons behind them.

The **Test Activity**, hence, is made by three principal steps: (1) write the test cases, based on design and requirements document; (2) run the test suite; (3) record the results of the test execution in a test log. The most expensive part, among the three, is test writing, because it is based on human effort from developers and testers.

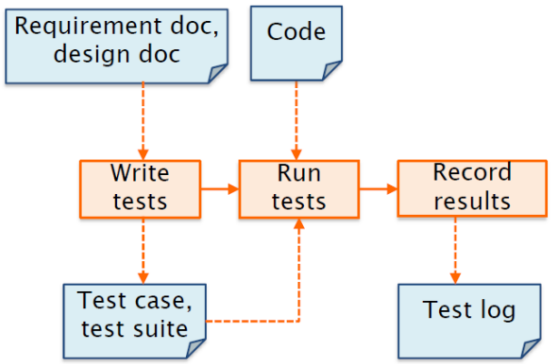

Figure 88 - Test Activities

There are three possible scenarios for what concerns the testing activities:

- Scenario 1: Developers and Testers form two different teams. Test cases are written informally by the development team, and then are developed and run by testers, that finally record the results of the execution of tests. Then, the test log is passed to developers, who perform debugging on code.
- Scenario 2: The testing process is operated in its entirety by the development team.
- Scenario 3: After the debugging process is operated by developers, a 3rd party tester team, that may be external to the development company, writes, run and documents a new test suite.

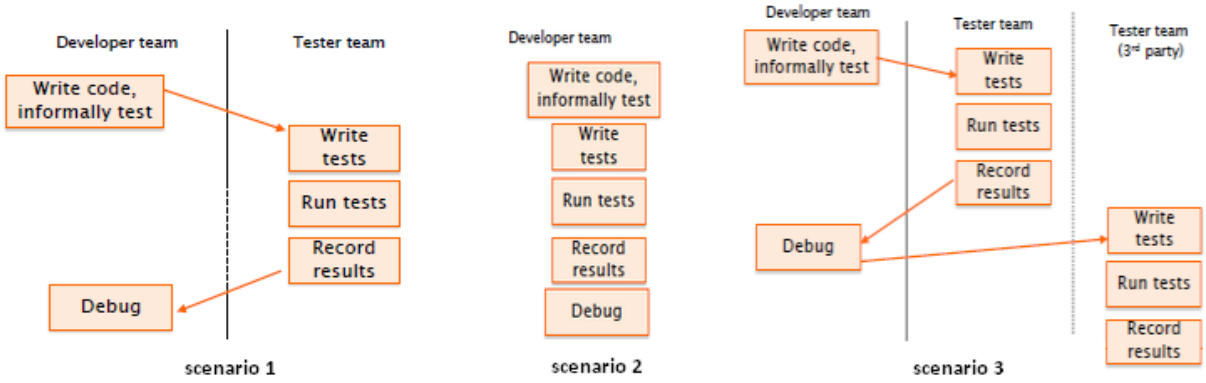

Figure 89 - scenarios for testing activities

An **Oracle** is what the output of a test case is compared to. Oracles can be human or automatic:

- A **human oracle** is based on individual judgement, and on understanding of the requirements specification;
- An **automatic oracle** is generated from formal requirements specification, or is based from similar software developed from other parties, or from previous versions of the program (regression).

The ideal condition would be to have an automatic oracle generation, and an automatic comparator between the actual output of a test case and the oracle. In reality, it is very difficult to have automatic oracles. They are most of the times human-generated, and hence they are also subject to errors: even test cases can be wrong. In some cases, e.g. UI comparisons, the comparators cannot be automatic.

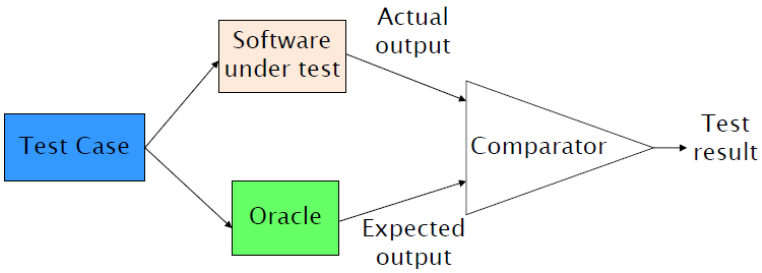

Figure 90 - Oracles and Comparators

Correctness is a characteristic of a software that provides correct output for all possible inputs. To demonstrate that no errors are present, each part, each behavior and each input of the program should be tested. However, exhaustive testing, for digital systems that are not continuous, should lead to a possibly infinite number of test cases, which is unfeasible. For instance, the Pentium processor (1994) had an error in the division function, that would have required 9 billion test causes to be tested exhaustively. To test the program whose flow chart is shown in figure 91, that performs 20 iterations with 5 possible path per each iterations, 10^{14} test cases would be required to perform exhaustive testing. Given that one millisecond is required to run a test case, exhaustive testing of such function would require 3170 years.

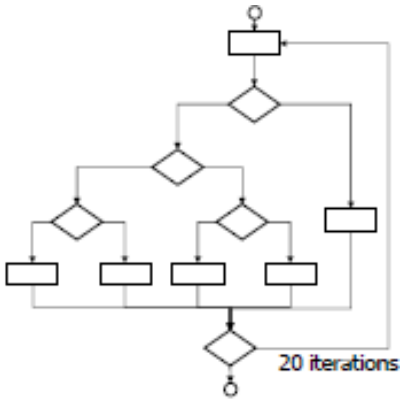

Figure 91 – A sample flow chart

Being exhaustive testing not possible, the goal of testing becomes to find defects, not to demonstrate that the system is defect free. According to **E. W. Dijkstra's thesis**, "*Testing can only reveal the presence of errors, never their absence".*

The goal of V&V in general becomes to assure a *good enough* level of confidence with the system. A number of test cases has to be selected, in order to guarantee sufficient **Coverage** of the system. Coverage is defined as the ratio between the number of test cases defined, and the total number of possible test cases. Correctness would be guaranteed by a test suites providing 100% coverages and triggering no failures. Other measures exist, in addition to coverage, to evaluate test suites.

Other theorems have been postulated for testing theory. The **Weinberg's law** says that *"a developer is unsuitable to test his/her code"*. The theorem suggests that testing should be performed always by separate Quality Assurance teams and peers, and not by the developers themselves: this is due to the fact that if a developer misunderstands a problem, it is not likely that he will find his own errors; in addition to that,

developers may not wind to highlight the presence of many errors in their own code. However, for non-formal testing, developers should test anyway the code they are writing.

The **Pareto-Zipf law** tells that *"approximately 80% of defects come from 20% of modules"*: in the testing proces it is therefore convenient to concentrate on the modules that have already proven faulty, instead of moving in inspecting others.

6.3.2 Test Classification

Testing techniques can be classified according to phase/granularity level at which they are executed, and to the approach they follow.

Types of testing per phase/granularity level are:

- **Unit testing**: the goal of unit tests is to find errors inside a single unit of code (for instance, a package or a class). Units are tested in isolation from other components of the system, and their size can depend on the programming language used, on the type of software application, and on the objectives of testing. Unit test is typically performed right after development.
- **Integration testing**: tests the interaction between separate modules of the system, to identify defects in the interfaces between them.
- **System testing**: the total, integrated and usable system is tested as a whole. The focus of the procedure is to evaluate the compliance of the system to the requirements behind it.

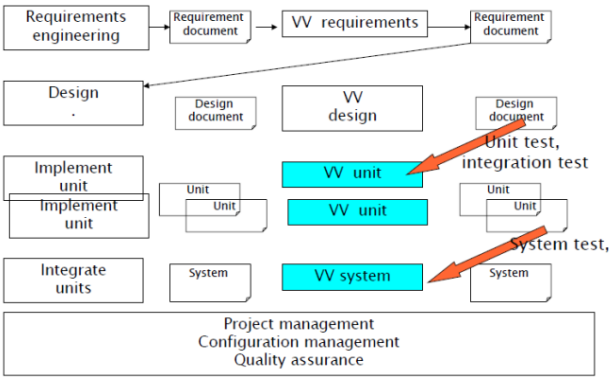

Figure 92 - Unit, integration and system testing in the software process

- **Regression testing**: it is a testing procedure applied to a software system that is evolving over time, to identify problems that result from changes. Non-failing test cases on version x are applied on version x+1 of the system: if new failures are detected by test execution, it is said that a *regression* is identified in the system.

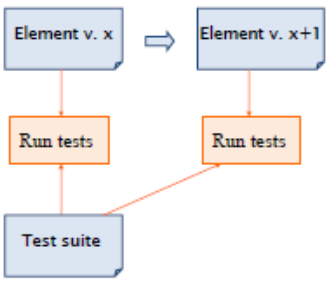

Figure 93 - Regression testing

Types of testing per approach are:

- **Functional / Black Box**: this approach tests the functionality of the system, without looking at its inner structure (i.e., code). Black box testing checks if the requirements are satisfied, starting from the requirementsdocument. To testers, the interfaces and prototypes of functions are known, but not their implementations. The focus of black box testing is on the input provided to the system and on the relative output, that is compared to the expected one to assess the validity of test cases. Black box testing can be applied at unit, integration and system level.
- **Structural / White Box**: it aims to check the internal structure, framework and mechanisms of a software application. It can be applied only at unit level, since at other levels the amount of code to test would be too much.
- **Reliability**: it focuses on the customer's needs, and exercises the most common scenarios of use of the application. Since it tests the functionalities of the finished software, it can be applied only at system level.
- **Risk**: it focuses on the verification of safety and security of the software, priorizing those functions having a higher likelihood or impact of failure. It is applied only at system level.

	Unit	Integration	System
Functional / Black Box	X	X	X
Structural / White Box	X		
Reliability			X
Risk			X

Table 19 - Testing Classification

6.3.3 Unit test - Black Box

Black Box testing is a form of functional testing. It does not test the structure of the code, but instead how the software works. In black box testing, input is provided to the unit (or to a set of units, or to the whole system) and the resulting output is compared to an oracle: if the output and the oracle are not the same, a failure is found. To ensure the effectiveness of black box testing, many different inputs should be provided, trying to cover all possible kinds of inputs that can be given to the application. Many techniques exist to choose the possible inputs: the main ones are random pick, equivalence classes and boundary conditions.

In **Random Pick** selection of inputs, the input of all test cases is chosen randomly, without any underlying logic. For any input defined, Oracles are necessary to compute the expected output, to which the actual output will be compared. The technique is very fast in defining test suites, and it is completely independent from the requirements document. The main disadvantage of the technique is the necessity of defining many test cases to cover all possible conditions in the tested software.

Figure 94 - Black Box Testing - Random Pick

Equivalence Classes Partitioning divides the input space in partitions. Each partition consist of inputs that, according to an understanding of the requirements, have a similar behaviour from the point of view of the

considered unit. For all the inputs taken from the same partition, the assumption is that the tests should yield the same results (i.e., fail or not fail): if a test in a class has not success, the other tests in the same class may have the same behavior. Hence, one or two test cases can be taken from each partition.

Boundary Condition Testing is based on selecting, as inputs, values that are at the boundaries between the defined conditions, or immediately close to them. The concept behind boundary condition testing is that extreme cases of inputs should always be tested, in order to test that the system is able to correctly discriminate between different classes of input values.

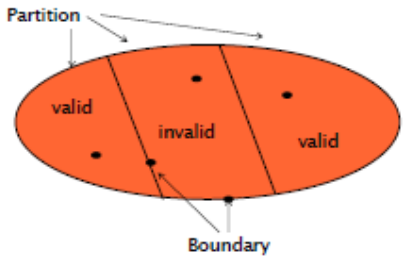

Figure 95 - Black Box Testing - Equivalence Classes and Boundaries

Equivalence classes are identified in the input space according to conditions on the inputs. The conditions that are applied depend on the type of inputs considered (e.g., true and false values for a boolean input, inside or outside a given interval for a numeric input, all the individual alternatives for an input that can tae a discrete set of values).

Conditions	Classes	Example
Single value	Valid value invalid values < value invalid values > value	Age = 33 Age < 33 Age > 33
Interval	Inside interval Outside one side Outside, other side	Age > 0 and age < 200 Age < 0 Age > 200
Boolean	True False	Married = true Married = false
Discrete set	Each value in set Value not in set	Status = single Status = married Status = divorced Status = jksfhj

Table 20 - Conditions and equivalence classes

The conditions defined on inputs split the input set in subsets, and the input space is fragmented in many classes (one for each combination of the conditions). Every equivalence class must be covered by a test case at least: a test case for each invalid input class, and a test case for each valid input class, that must cover as many remaining valid classes as possible. The selection of test cases, thus, becomes a *combinatorial problem*.

In general, the process of black box testing with equivalence classes and boundary conditions can be synthesized in five different steps:

1 Identify criteria (from requirements) for definition of classes of inputs;
2 Define conditions for each criterion;

3 Define an order for combining the conditions: in general, the most restrictive conditions are tested first, to restrict the number of test cases to define and run;

4 Combine conditions, obtaining partitions: the more partitions are made, the more defects are likely to be found in the black box testing process;

5 Define a test case for each partition.

In the case of testing **Object Oriented classes**, the modules of the software to test can have states. This must be considered while defining the partitions, because there may be many functions in objects leading from one state to another, and test cases pertaining to a function may require sequences of calls to other functions.

Example 1: Function double squareRoot(double x);

Partitions:	*Positive numbers*	*T1(1; sqrt(1))*
	Negative numbers (invalid)	*T2(-1; error)*
Boundary values:	*Zero and close values*	*T3(0; sqrt(0)), T4(0.01; sqrt(0.01))*
		T5(-0.01; error)
	"infinite" and close values	*T6(maxdouble; sqrt(maxdouble))*
		T7(maxdouble-0.01; sqrt(maxdouble-0.01))
		T8(mindouble; error)
		T9(mindouble+0.01; error)

A function that computes the square root function has two possible input partitions: positive numbers plus the zero value, and negative numbers. All inputs from the former equivalence class yield valid result, while all the ones from the latter yield invalid results (since the square root of a negative number is mathematically unfeasible).

The boundary values for the function are 0, that is the value between the two equivalence classes defined, and "infinite" (represented by the maximum and minimum double values of the programming language used). Of the two boundary values close to zero considered, the negative one must yield invalid result.

Example 2: Function int convert(string s);
The function converts a sequence of chars (max 6) to an integer number; negative numbers start with -.

Partitions:	*string represents a well formed integer (boolean)*
	Sign of number (positive or negative)
	Number of characters (less than six, more than six)

Equivalence classes - Combinatorial			
Well formed integer	*Sign*	*N char*	*Test case*
Yes	*Positive*	*<= 6*	*T1("123"; 123)*
		> 6	*T2("1234567"; error)*
	Negative	*<= 6*	*T3("-123"; -123)*
		> 6	*T4("-123456"; error)*
No	*Positive*	*<= 6*	*T5("1d3"; error)*

		> 6	*T6("1d34567"; error)*
	Negative	*<= 6*	*T7("-1d3"; error)*
		> 6	*T8("-1d3456"; error)*

Boundary – Combinatorial			
Well formed integer	*Sign*	*N char*	*Boundary to test*
Yes	*Positive*	*<= 6*	*"0" "999999"*
		> 6	*"9999999"*
	Negative	*<= 6*	*"-0" "-99999"*
		> 6	*"-999999"*
No	*Positive*	*<= 6*	*""*
		> 6	*" " (7 blanks)*
	Negative	*<= 6*	*"-"*
		> 6	*"- " (6 blanks)*

6.3.4 Unit Test – White Box

White box testing is a form of structural testing that is based on the code, and has the objective of validating the internals of a software applications. White box test cases must be defined with the aim of maximizing the coverage of source code. Several different types of coverage are defined for software, and a white box test suite should maximize each of them. Coverage is computed for each test case, and for a whole test suite as the sum of the coverage guaranteed by all test cases that are part of it.

Statement coverage is defined as the ratio between the statements covered by the test suite, and the total number of statements in the tested software unit. Statement coverage is also called *line coverage*, since statements in software identify with lines of code.

To aid white box testing, the tested program has to be represented by a control flow graph. The control flow graph can be obtained by examining the lines of code of the software, or from the original flow chart. Each node, in the control flow graph, represents an atomic instruction or decision. Edges in the control flow graph represent the transfer of control from a node to another. Decisions may be if clauses or conditions of permanence in *while* or *for* loops. Nodes can be collapsed into basic blocks: a basic block is a sequence of instructions that has a single entry point at the initial instruction, and a single exit point at the final instruction.

Control flow graphs also help in identifying **Dead Code**, i.e. parts of code that cannot be executed: only live code has to be tested, whereas dead code can be ignored.

Node Coverage starts from the Control Flow Graph, and is defined as the ratio between the number of nodes covered by test cases and the total number of nodes of the software. Since Nodes are composed by statements, node coverage can be cosidered as coincident to statement coverage.

Decision Coverage measures the amount of decisions (if/then/else) in the software code that are covered by the test suite. It is defined as the ratio between the total number of decisions covered by the test suite and the total number of possible decision in the code. Decision coverage equals to the **Edge Coverage**, i.e. the ratio between the number of covered edges in a control flow graph and the total number of edges. For the computation of decision coverage, each decision is considered independently (in the sense that the possible

influence of a decision on successive one in software code is not taken into account). Edge Coverage implies Node Coverage, but not vice-versa.

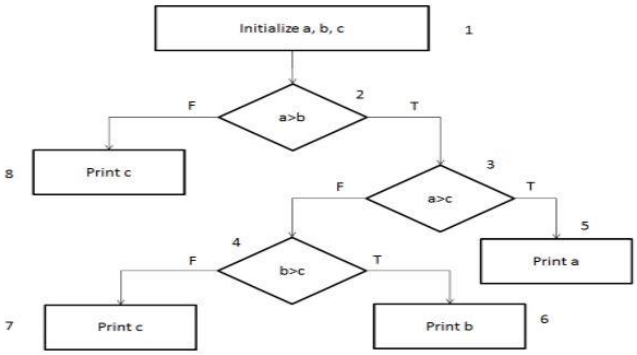

Figure 96 - Control flow graph for finding the greatest of three numbers

Condition Coverage aims to teach all the possible outcomes from conditional clauses in source code, e.g. if clauses or for/while loops. Simple Condition Coverage aims at testing conditions indipendently by others; Multiple condition coverage aims at testing each possible combinations of connected multiple conditions. For instance, for an if statement controlling a boolean expression made of two atomic conditions, four possible combinations can be possible (namely, true-true, true-false, false-true, false-false). In this case, simple coverage is obtainable with two tests (it is sufficient a test case causing both atomic expressions to be true, and another causing both to be false), while multiple condition coverage requires all the four possible combinations of test cases. Simple condition coverage does not imply decision coverage, where as multiple condition coverage does.

Path Coverage aims at covering all possible execution paths (all combinations of control flow decisions) from the starting point to the exits of the control flow graph. It is defined as the ratio of all the paths covered by test cases, and the total number of paths in the control flow graph. If no cycles are present in the code, the number of paths is finite; otherwise, the number of paths can encounter combinatorial explosion, and obtaining 100% path coverage may be unfeasible with a reasonable number of test cases.

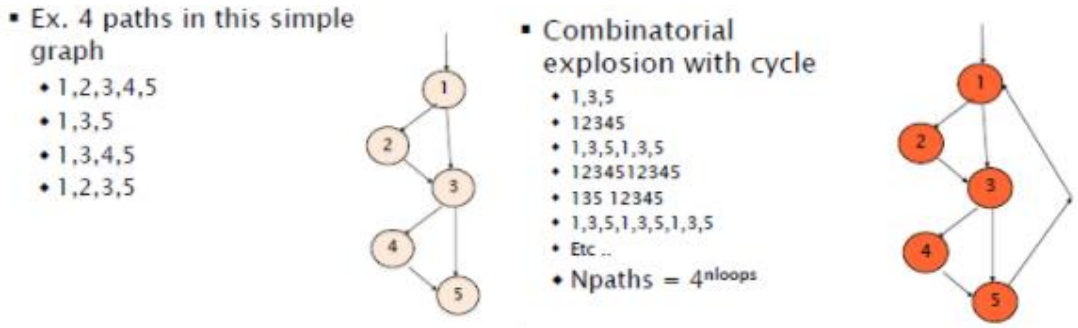

Figure 97 - Combinatorial explosion for path coverage

Cyclomatic Complexity is a measure that can come in handy to evaluate how much the code is complexity: it is defined as $E - N + 2$, where E is the number of Edges in the Control Flow Graph, and N the number of nodes. Cyclomatic Complexity is by definition the number of linearly independent cycles in code (i.e., cycles that do not contain other cycles). The measure is often used in the automotive or other critical domains: since codes with small Cyclomatic Complexity are more testable, an upper bound on the property is imposed by requirements.

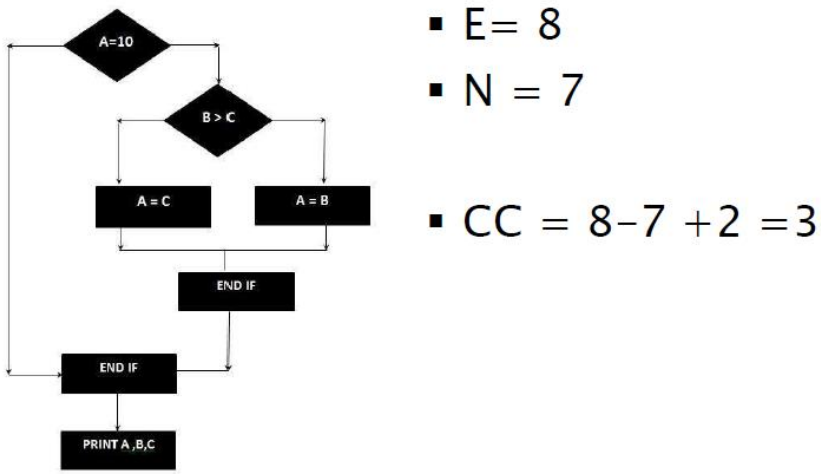

Figure 98 - Cyclomatic Complexity Example

Loop Coverage is an approximation of path coverage. It aims at testing the loops that are present in test code, selecting test cases such as every loop boundary and interior is tested. To test the loop boundaries, test cases that execute the loop 0 times are selected. To test the interiors of the loop, test cases that execute the loop one and multiple times are selected. The selections of test cases for loop coverage is done to cover the loops that are not covered well with decivion coverage. If many loop are present in the code, each one must be considered independent by the others.

Two additional important concepts for white-box testing are **Observability** and **Controllability**. Controllability is the possibility of actually perform interventions on the execution of the code (i.e., no hard Icoded decisions are present). If, for example, a loop is not controllable, the only testable loop condition is the hard-coded one. Also, it is not always possible to know how many time the loop code is executed: if this happens, there is no controllability over it. In general, changing a program to achieve observability/controllability is not possible: another version of the program is tested, not the original one.

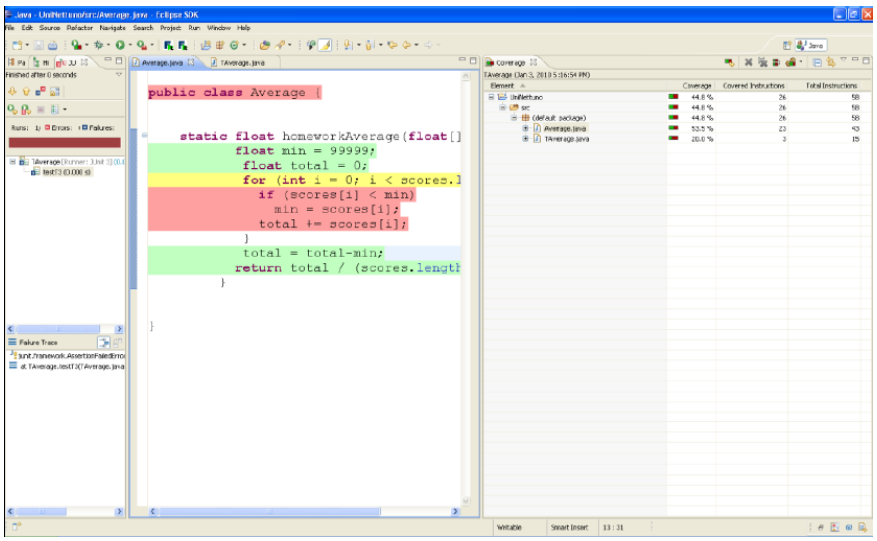

Figure 99 - Cobertura Tool for Coverage

Following the Howden theorem, it is impossible to find an algorithm that can generate a finite ideal test suite, but the tester can be helped in that construction using tools. Program languages exist for creating automated

and repeatable test cases. Junit offers tools that enable faster development of tests and a form of automatic code development starting from an excel definition of test data. Graphical tools (e.g., Eclemma for Ecliplse IDE) underline executed statements and give statistics about coverage.

6.3.5 Mutation Testing

Mutation testing (introduced in early 70s and also known as Mutation Analysis or Program Mutation, and used both in software and hardware) is a procedure aimed at checking if the test cases are written and able to find errors. The idea behind mutation testing is, after having written the test cases, to inject errors (in the form of single small changes) in the program, and then verify whether the test cases are able to catch the error injection. Common mutations that are introduced in programs are deletions or swaps of statements, changes in boolean relations or arithmetic operations, replacement of variables or constant values. As an example, for a sum function, the algorithm can be modified in order to multiply the given input. When the tests are run again, a failure is expected.

According to Mutation Testing terminology, a **Mutant** is a mutated version of a program, i.e. the program with an even small change. A **Killable (Non Equivalent) Mutant** is a program that, after the injected change, is not functionally equivalent to the original: a correctly written test case should be able to detect the difference with respect to the original, which is called *killing the mutant.* An **Equivalent Mutant** is, on the other hand, a program that after the change is functionally equivalent to the original program: no test case –even properly written- can kill such mutant.

The **Mutation score** is a property of the test suite examined with mutation testing, and is defined as the ratio between all the killed non equivalent mutants, and the total number of non equivalent mutants.

6.3.6 Integration Testing

When unit testing is performed, it is assumed that each external call to other units works properly. Integration testing is about testing the interaction between different units of the system that depend on each other, i.e. functional calls from one unit to the other exist. Integration problems can be genrated by any kind of interaction between units (e.g., calls to a function that does not work properly, passage of wrong parameters or wrong parameter order, bad timing).

Integration testing is of crucial importance for complex software/hardware architectures, and should always be performed in mission or safety critical systems. An example of the disasters that may be caused by integration problem is the Mars Polar lander incident (in 2000): the Polar lander was a probe expected to land on Mars, for scientific exploration. A measure of length had to be exchanged between two components, developed by two different teas. The two teams used two different unit of measures. The difference was very small, and went unnoticed until the probe crashed on Mars.

The operation of integration testing can be based on the **Dependency Graph**, which depicts the dependency between each unit of the program.

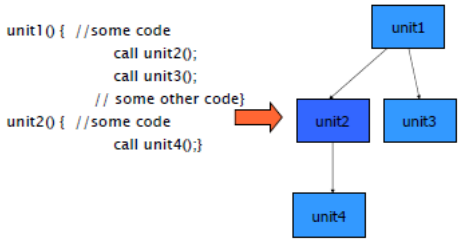

Figure 100 - Dependency Graph

The tools that can be used for integration testing are **stubs** and **drivers**. A driver is an unit (a function or class) that is developed specifically to pilot another unit, in order to test it. A stub is a unit that is developed to substitute another unit. A stub must be simpler than the substituted one, and it works as a trade off for testing (between simplicity of testing and amount of covered functionalities); it can be a fake unit, or a trusted one, for instance taken from a library. For instance, if an unit calls another unit that returns an element for a catalogue of product, but there is no access to the implementation of the latter, a simple stub catalogue can be created, returning randomly one of a limited set of elements.

Three approaches exist for integration testing:

- In **Big Bang Integration**, all units are developed and aggregated, and then the test cases are applied to the overall system, as if it was a single unit. The main problem of this approach is that, when a defect is found, it cannot be located, because it could have been generated by any unit in the integrated system, and by any interaction between units. Moreover, the approach does not scale well to large software systems.

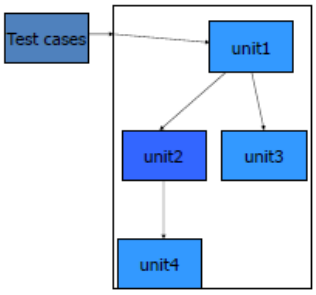

Figure 101 - Big Bang Integration

- **Test in Isolation** tests each unit by itself, representing the dependencies to and from the others with stubs. The test must be repeated for each non independent unit in the system. The main drawbacks of the practice are the difficulty and cost of stub creation.

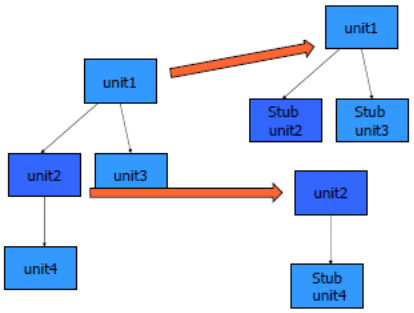

Figure 102 - Test in isolation

- The **Incremental Integration** approach adds a unit at a time, testing partial aggregates (and not directly the whole system as in the big bang approach). The main advantage is that, when defects are found, they are typically due to the last unit integrated, and hence they are easilly localized. The main drawback of the technique is that it requires more tests, stubs and drivers to write.
  The **bottom-up** incremental integration starts from testing units with no outgoing dependencies: they are tested in isolation, with unit testing techniques. Once they are tested they become "trusted" units; other units that have outgoing dependencies to them are integrated to them, and the partial aggregates are tested: if defects are found, they should come only from the newly added units, or from the interactions with the trusted ones. In the bottom-up incremental integrations, drivers may be needed to provide inputs to the isolated classes. Bottom-up incremental integration allows to start testing early in the development process, and to observe directly the lower levels of the software. The main disadvantage is the need of defining drivers for all lower level units.

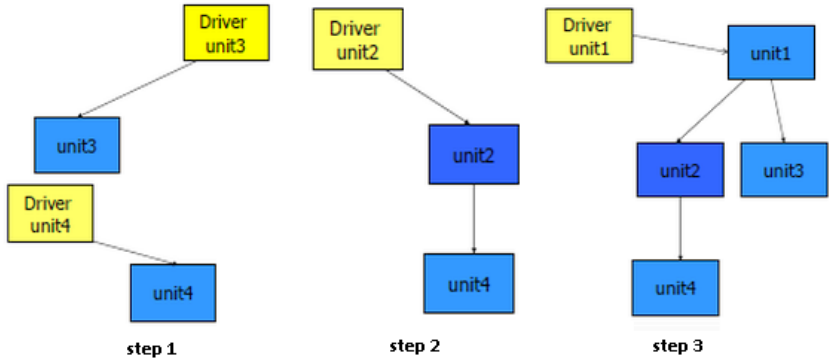

Figure 103 - Bottom Up Incremental Integration

The **top-down** incremental integration strategy, instead of starting from the units with no dependencies, starts from the highest-level units. To test them, stubs are used, then units in lower levels –the ones that used to be mocked by stubs- are integrated progressively. This testing method fits well if the software is developed in a top down approach, and can detect early architectural flaws. It allows also to have a limited working system available early. However, the top-down approach does not provide observability for lower level classes, and requires the definition of stubs for all low-level units.

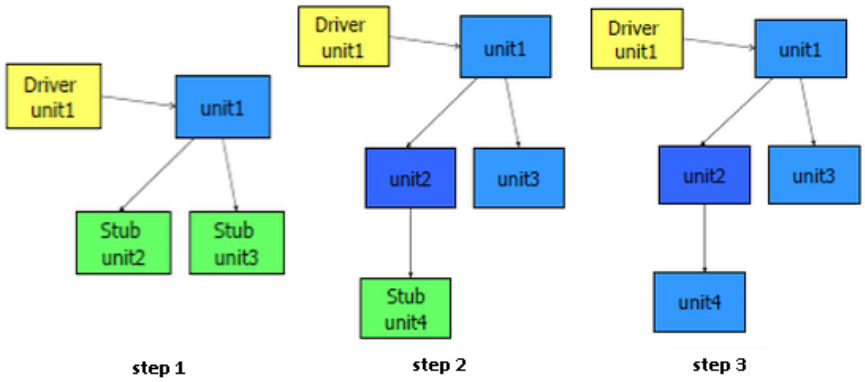

Figure 104 - Top Down Incremental Integration

A problem with integration testing may arise when units have dependencies loops. If unit A depends on B and vice-versa, they can be considered as a single unit and unit tested. If the units are too large and/or

complex and it is difficult to consider them as a single entity, the alternative is to change the design, and split the loop between the two units using a third intermediary.

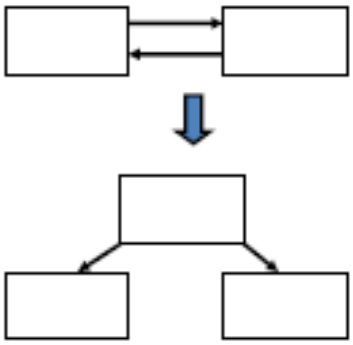

Figure 105 - Integration testing: problem of dependency loops

When integrating software with hardware (e.g., for embedded software), a typical solution is to test software units (software-in-the-loop/model-in-the-loop) in the developement environment, with stubs that replace hardware. Then, software and hardware are integrated (hardware-in-the-loop) in the target environment (e.g., Android on ARM Device). Often, replacing hardware pieces with stubs requires significant effort, because it is necessary to model/simulate the environment in which the hardware will be deployed.

6.3.7 System Testing

System testing is applied to the software system as a whole. The whole system is now considered as a black box, and it is checked whether the functionalities offered reflect the requirements on which the system is basd. System testing is often performed by a test team, typically different from the development team.

System testing should check that all requirements are available, and that all scenarios are correct. Hence, the starting point of system testing are scenarios and use cases, as listed in the requirement document. At least one test case is created per functional requirement. The coverage measure for system testing can therefore be defined as the amount of functional requirements/scenario covered, and the total amount of functional requirements/scenario defined in the requirements document.

System testing should also consider the usage profile (i.e., most common ways of using the system), and should be done in conditions as close as possible to the working conditions. The latter situation can be obtained developing the system and installing it in a real environment (very expensive) or in a testing environment (may leverage the use of simulators and hence be more affordable).

The system is tested on a specific **Platform**. The platform is the environment where the application runs, defined by many components (operating system, database, network, memory, CPU, libraries, other applications installed, other users, and so on). The platform on which a system is tested can belong to one of two typologies:

- The **Target Platform** is where the system will eventually run, for daily use. Most of the times it cannot be used for production, because production operations may create risks of data corruption and may hamper its availability.
- The **Production Platform** is where the system is produced. In most cases it differs from the target platform. For instance, for an information system (e.g., bank account management, student enrollment management for a university) the production platform is a PC or a workstation where the

database is replicated in a safe and simplified form; for an embedded system (e.g., heating control systems, mobile phones) the production platform is typically a desktop PC, with emulated external devices.

In addition to the platform used, the other variable in system testing is who is performing the test: it can be performed by developers, either on production or target platform; on the other hand, it can be performed by users, either on the production or target platform. Testing performed by selected groups of users on target platform is called **beta testing**. System testing performed by the customers, with its own data and test cases and on the target platform, is called **acceptance testing**.

Non-functional proprieties are also object of system testing: they are typically properties that emerge when the system is already available (e.g., efficiency, or reliability). Some non functional properties are of particular interest in the context of system testing:

- **Configuration:** availability and effectiveness of mechanisms to perform changes on the system;
- **Recovery:** capability of the system to react to catastrophic events (e.g., serious hardware malfunctionings or huge data loss);
- **Stress:** reliability of the system under limit conditions;
- **Security:** resilience of the system to non-authorized accesses.

6.3.8 Reliability and Risk Based Testing

The aim of **Reliability Testing** is to provide an estimate of the non-functional property of reliability, which is defined as the probability of having a failure over a given period of time. Other alternative measures can be used for reliability, like the defect rate (ratio of defects and time) and MTBF (mean time between defects). A constraint for reliability tests is that test cases –that may be many- must be independent from each other, so that a defect fix does not introduce other defects.

The typical trend of reliability testing for software products is to find a huge number of defects at the initial stages, that then decreases over time. This trend is very different from the one typical of mechanical produts, which age during time, with decreasing reliability.

Software Reliability testing is typically divided in three parts:

- **Modeling:** specific models are applied to the software under test, to capture their characteristics and simplify the problem of identifying reliability issues;
- **Measurement:** factors are selected and measured to estimate it and compare it with other products (e.g., MTBF);
- **Improvement:** corrective actions on the system to enhance the reliability metrics selected.

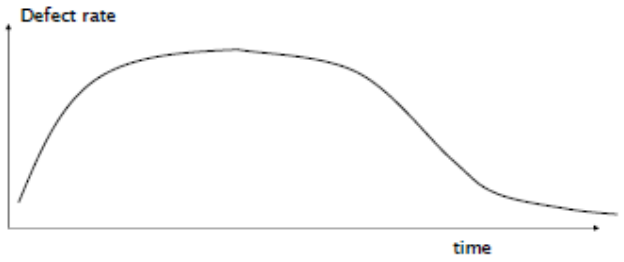

Figure 106 - sw reliability with time

Risk Based Testing is a form of testing performed on projects based on the risks identified for it. Risks are prioritized, in terms of probability and effects, and test cases are emphasized according to the risks they are linked to.

6.3.9 Test Documentation

Test documentation is almost always mandatory for software project, unless in cases in which very informal testing or exploratory testing is performed. Test cases can be represented and documented informally with natural language, e.g. with Word documents. Several means, on the other hand, exist for representing test cases formally, for instance Excel documents with translators to programming languages.

According to the Test Process standard by IEEE (latest revision std. 829-2008), a set of deliverables must be produced by a testing process: some relative to planning and specification, others to the enactment of testing.

Five deliverables are specific to test planning:

- The **Master Test Plan (MTP)** is the main deliverable of the testing documentation: it guides the management of testing, establishing the plan and schedule of testing activitiies. General pass and fail criteria and resources needed are defined. The Master Test Plan can contain the traceability matrix, in which test cases are mapped to requirements (as in the requirements document).
- The **Level Test Plan (LTP)** defines in detail each test activity, in terms of scope, approach used, and scheule. It lists the individual testing tasks to be performed, with the personnel responsible for each task and the associated risk.
- The **Test Design specification (LTD)** defines, for one or more features to be tested, the details of the approach that will be used: the testing techniques to use, the analysis of the results obtained, the list of test cases and motivation, and attributes that are proper of such feature.
- The **Test Case specification (LTC)** specifies an individual test case, in terms of goal, input data, expected output (oracle), test conditions (hardware and software required), inter-test dependencies and procedural requirements.
- The **Test Procedure specification (LTPr)** specifies the steps needed to execute one or more test cases: the operations of preparation of test cases, the measurements that have to be collected, the actions to perform in case of exceptional events, the way to resume a suspended test.

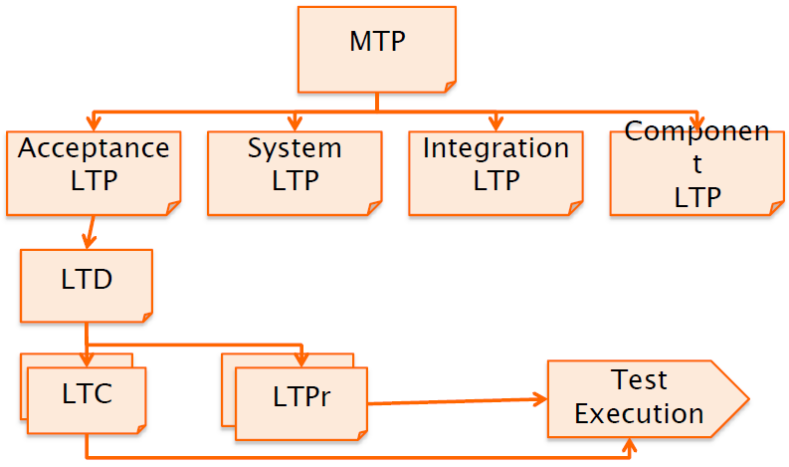

Figure 107 - Test Planning Documents and relationships

Four deliverables are related to test execution:

- **Level Test Log (LTL)** provides a chronological record of relevant details about test execution. All (or at least part of) this information can be captured by automated tools. Relevant elements of the test log is the description of the execution of test cases, the results of the procedure (success, or failure), the environment in which tests are executed, and the anomalies found.
- An **Anomaly Report (AR)** documents any event that occurs during the test process and that requires further investigation. The description of the event must include: time and context, input, expected output, actual output, anomalies and impact.
- A **Level Test Report (LTR)** summarizes the results of the designated testing activities, with an overview of the results and more detailed results (open and resolved anomalies, test executed and metrics collected, test assessment with coverage metrics). The LTR can contain recommendations, in the form of an evaluation of the item under test, and comments on the suitability for production use.
- The **Master Test Report (MTR)** summarizes the result of all testing activities: the test tasks results, the anomalies and resolutions obtained, the quality assessment of the release, the summary of all metrics collected.

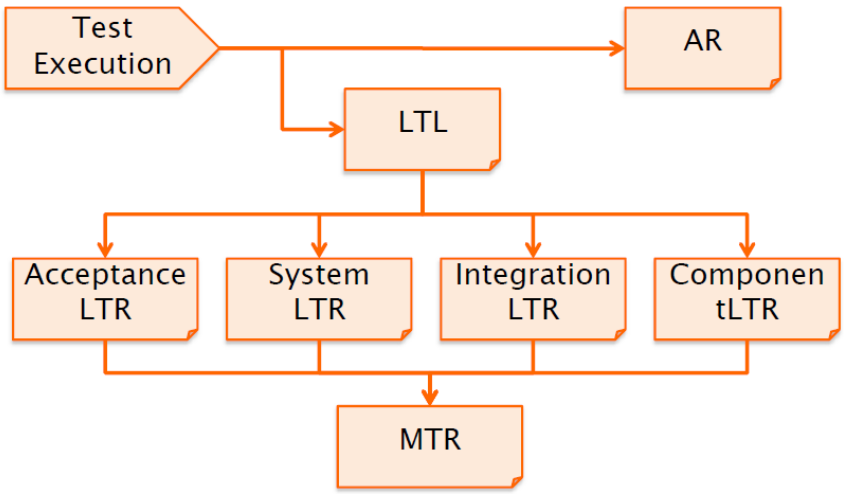

Figure 108 - execution deliverables and relationships

6.3.10 Test Automation

Test cases should not only be documented, so that they are not easily lost and that they can be reapplied, but also automated, so that the execution of test cases is faster and free from human-induced errors. Several tools are available for automated testing:

- **Table-Based Testing:** test cases are written as tables, and linked to the application to be tested. The main advantages of table-based testing are the easiness for end users (or customers) to write the acceptance tests, that are often black-box, and the indipendence from the GUI of the system. Once test cases are defined in tables, they can be automated using specific frameworks: with FIT, for instance, the user specifies the tests in HTML tables, and developers define fixtures to parse tables and automate tests; FIT then compares the expected results in the tables and the actual outcome of test cases. FITnesse is a standalone wiki hooked to FIT, that allows groups to easily edit test files without worrying about ensuring that the correct versions of test cases propagate out to all locations.

- **Capture and Replay (C&R):** capture tools allow to capture the end user test, as they are performed on the software itself, recording each interaction performed with the Graphic User Interface (e.g., mouse movements and clicks, keystrokes from keyboards) and the corresponding outputs given by the software. Aside, replay tools allow to replicate, on a given software, pre-recorded sequences. Capture and Replay tools ensure very high easiness in writing test cases for end users, that can define them seamlessly as they are normally using the application, and allow easy automation for test suites. On the other hand, C&R test suites cannot be applied to functionalities of the applications that cannot be easily triggered from the user interface, and depend strongly on the GUI structure and stability: if the GUI changes, the captured test cases cannot be applied again.
- **Coverage Tools:** tools that show graphically and numerically the coverage obtained by test suites on source code. They are typically plug-ins for popular IDEs (e.g., Eclemma, Cobertura for Eclipse).
- **Profilers:** tools that trace the time spent in each function of a software, typically used for performance testing (e.g., Jmeter).

6.4 Static Analysis

Static analysis is the activity of checing the source code without running it. Several kinds of static analysis exist:

- **Compilation Analysis** is the one performed by compilers, that check the code for syntax, type and semantic correctness. The errors detected in this form of static analysis strongly depend on the language and its type (loosely typed vs. strongly typed ones, static vs. dynamic visibility).
- **Control Flow Analysis** is based on the Control Flow Graph of the program, and checs the control flow constructs (basic blocks, loops, method calls, and exceptions handling).
- **Data Flow Analysis** gathers information about the data used by the program, and about the possible sets of values computed at different points of the program. Values of variables are analyzed during execution, to find out anomalies. The three basic operations that are monitored for variables are the definition, the use (read or write the variable), and the nullification: forbidden sequences of the three (like definition-definition and nullification-use) are identified and signaled, without executing the code.
- **Symbolic Execution** is an analysis of the program to determine which kinds of inputs cause specific parts of it to be executed and specific outputs to be given. It uses symbolic values as inputs. The output variables are expressed as symbolic formulas of the input variables. The technique can be extremely complex even for very simple programs.

VII CONFIGURATION MANAGEMENT

7.1 Motivation

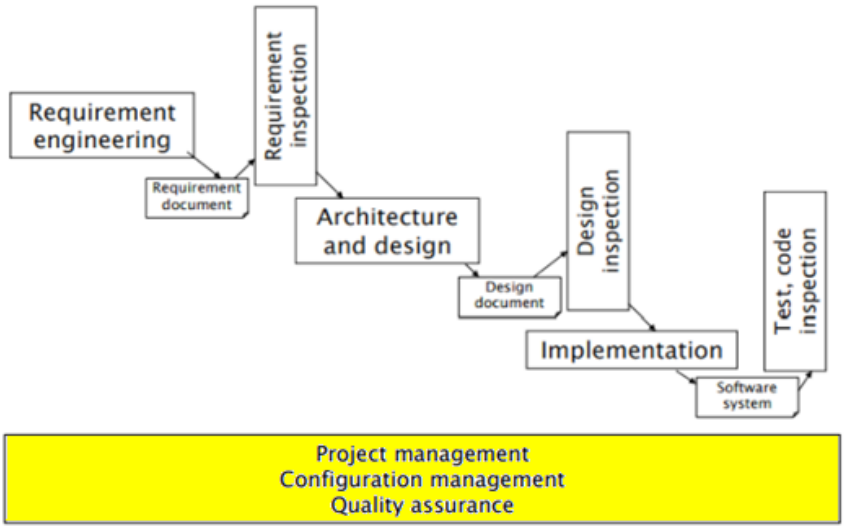

Figure 109 - Configuration Management in the software process

Configuration Management is one of the Management Activities of the software process. It is one of the most important operations that have to be performed to deploy software. The main goal of configuration management is to keep track of the evolution of a software, release by release. CM must identify and manage all the parts of software and their evolution, control the changes performed to any part and the permissions to access and modify them, and guarantee the possibility of rebuilding previous versions of software.

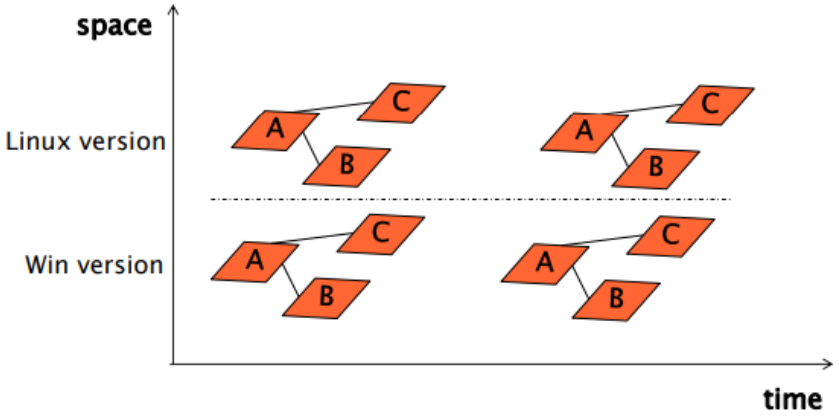

Figure 110 - Growth of a software system in space and time

According to Bersoff et al. (1980), *"No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle".* Software may evolve and grow in two different dimensions:

- **Time**: over time, the parts of the software system are subject to change. Different releases of the same software are published (e.g. for adding new functionalities, or for fixing defects);

- **Space:** different versions of the same software can be deployed in parallel (e.g., a Windows and Linux version of the same software) in order to adapt it to different environments and situations. In addition to that, a software typically grows in space by itself, since it is made by many parts (e.g., test logs, code, documents, and so on: a large software system may generate thousands of separate documents).

Configuration Management copes with the existence of different parts of the software system, and with the evolution of each of them during time. Four main concepts are under the practice of configuration management:

- **Versioning:** what is the history of a given document/source file?
- **Configuration:** what is the correct set of documents for a specific need?
- **Change Control:** who can access and change what?
- **Build:** how the whole system is obtained?

7.2 Versioning

Software Versioning is the process of keeping track of different releases of a program. It allows developers and analysts to understand what changes have been made and when on software code and related documents. A basic, naive way of performing versioning is a simple progressive naming of the files and packages of a software. For refined Versioning management, tools are capable of keeping tracks of versions, providing the users the possibility of deciding when new versions of the a file (with the same name or with a different version name) must be created, through a *commit*. Versioning tools make always possible to recover a past version of either a single file or the whole software.

7.2.1 Configuration Item

A Configuration Item (CI) is an element put under configuration control. It is the basic unit of the Configuration Management system, and corresponds to a piece of code or any kind of work product (e.g., requirement or design documents) related to the program. Based on the type of element, a Configuration Item may corresponde to one or more document/files (for instance, a Configuration Item for a C++ class can be made by two files, the .hpp header and the .cpp class file). Every CI is identified by a name and a version number, and all versions of the file (its history) are kept.

Not all the documents in a project must be treated as Configuration Items: the choiche of which documents to consider as CIs is a choiche specific to each project. Considering all documents as CIs, in fact, may cause unnecessary overhead for the Configuration Management activity; on the contrary, if no document is treated as a CI, no history and no configuration information is available for the system. For instance, it is generally a good choice to exclude auto-generated files from the Configuration Items, since they are easily obtainable for each version without keeping track of all modifications performed on them.

7.2.2 Version

A Version is an instance of a Configuration Item, at a certain point in time (e.g, Req document at 2015/07/11, and same Req document at 2015/07/12). Each instance of a CI has a unique number, that identifies the CI during time.

Versions record the changes that are performed during time on Configuration Items. The **Derivation History** is the record of all the changes applied to a document or code component, along with the rationale, the performer and the date and time of every change.

The derivation history can be included, for instance, as a header in every new version of a document/source file: a standard prologue style can be used for this purposes, so that ad-hoc tools (e.g., svn) may process it automatically.

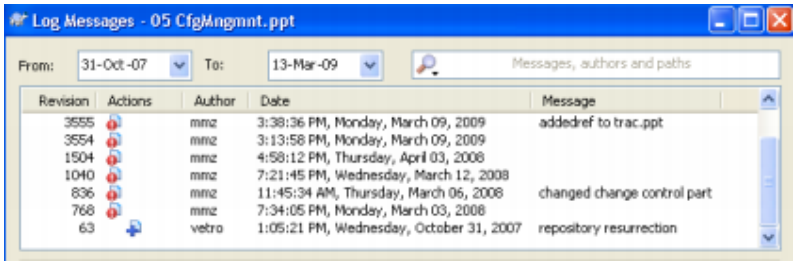

Figure 111 - Derivation history – svn

7.2.3 Configuration

A Configuration is a set of different Configuration Items, that have dependencies between them, in a specific version. In general, Configurations do not include only code, but also other types of documents that are part of the project.

Dependencies among Configuration Items may be syntactically defined (e.g. include instructions in C#, or import statements in Java, that link different source code files); however, the majority of them are not, e.g. between test cases, requirements and tested code fragments (issue of *traceability*).

Also Configurations have their version. Inside the Configuration, each CI is in a certain version. The same version of a CI may appear in different versions of a Coniguration (cfr figure 112, where two classes, each having two different versions, may be part of three different versions of the same Configuration).

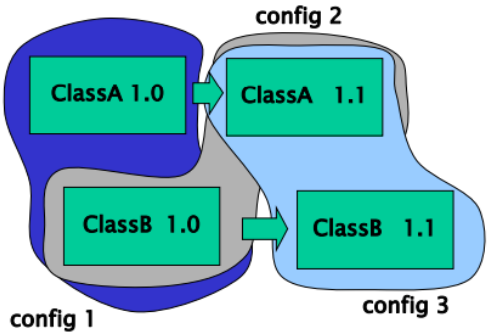

Figure 112 - Configurations and Configuration Items

There are two main approaches to identify Configurations:

- Keep an ID for the configuration, and list the ID of the CIs belonging to it (as it is done in CVS). In this case, the configuration changes name each time a CI changes; CIs never change names if they are not modified. In this approach, it is easy to identify when the Configuration Items change, but the CI names and configuration names must be managed separately.

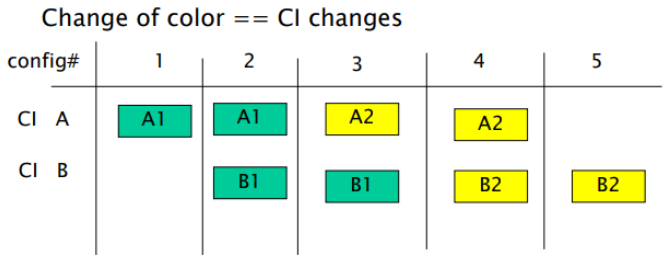

Figure 113 - Different IDs for Configurations and CIs

- Keep an ID for the configuration, and use the same ID for all the CIs belonging to it (as it is done in Subversion and GIT). In this case, a change in any CI in a configuration makes the configuration to change name, as well as all others CIs inside it (even if they are not changed). Using this approach, CIs and Configurations are managed together (and hence it is easier to know which Configuration Items are included in a given Configuration), but it is more difficult to understand the exact moments when a Configuration Item is modified.

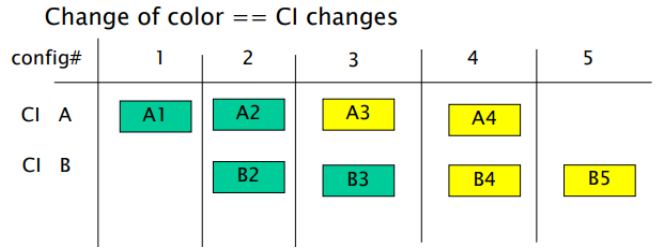

Figure 114 - Same IDs for Configurations and CIs

A **Baseline** is a special configuration that is in a stable form. Not all configurations are baselines. The baseline is often frozen in its state and deployed: modifications start by it, and if something erroneous occurs, the project is rollbacked to the baselyne. There are two types of baseline: a *development baseline* is for internal use, and is a safe rollback point for development; a *product baseline* is delivered to the user/customer, and is a stable version of the program.

Versioning practices can be characterized also by the **Data Management Model**. There are two ways to keep the information about changes on the Configurations and CIs:

- Keep **Differences:** at a new commit, only what has changed from previous version is stored.
- Keep **snapshots:** at every commit, a copy of all the CIs at that moment is stored. If files are not modified from last version, a link to the previous one can be kept.

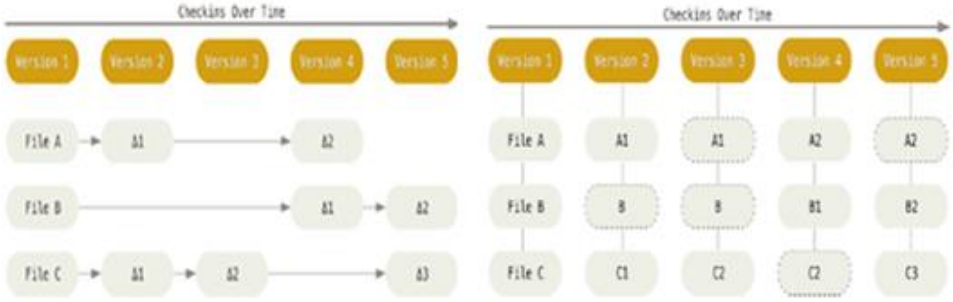

Figure 115 - Data management models: differences vs snapshots

7.3 Change Control

In software development, typically there are teams made of many developers woring on parts of software, that must be shared among them. To share parts of software, common repositories (shared folders) are used, in which all developers can read and write documents and programs.

Each developer does not wor directly in the **Repository** (where Configuration Items and Configurations are stored) but in his **Workspace**, in which copies of the CIs are stored. Hence, for each project there are many workspaces (one for each developer) and often a single repository. Workspaces need to be synchronized with the repository, to export the changes made by the developer locally, and to import the changes made by other developers remotely.

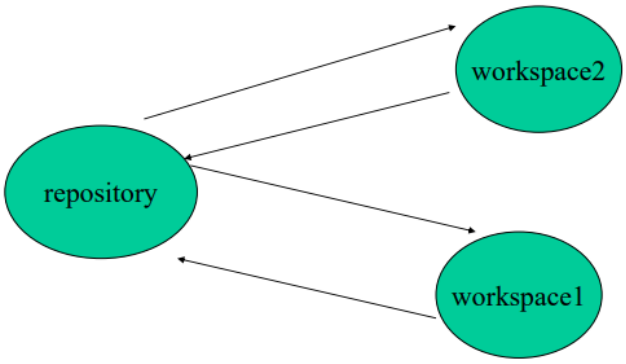

Figure 116 - Repository and Workspaces

The main issue to be addressed in Change Control is the synchronization between modifications performed by different developers. After working on his local version, each developers puts it in the repository, with a *commit* command, overriding the current version. If another developer is working on the same file, the local file in his workspace may be not automatically updated according to the new one committed. There are many different implementations to address Change Control, ranging from shared files on file servers, to CMS tools with change control (checkin/checkout mechanisms).

7.3.1 Shared Files

In this implementation, the simplest one, there is a file server where files can be uploaded and downloaded by developers, without versioning and change control.

With shared files and folders changes are not disciplined: there is no control on changes performed on files. This may cause data loss, because a developer may copy his modifications on a shared file and override the ones performed by others (see the example in figure 117, in which changes by John are lost).

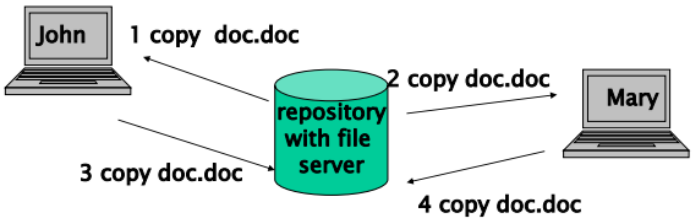

Figure 117 - Repository - shared files

Furthermore, a shared folder does not offer special features of configuration management, like the automatic versioning: all versioning must be made manually, using different names for files. In general, the approach can fit only to very small projects.

7.3.2 Configuration Manager Server (CMS)

The implementation with Configuration Manager Server aims at overcoming the problems of shared folders. Two main operations are performed on files that are extracted from the repository: **check-in** (commit), that updates the repository with the CIs local changes, and **check-out** (pull), that updates the workspace getting the CIs stored in the repository. These two basic operations are used to discipline the changes on the documents.

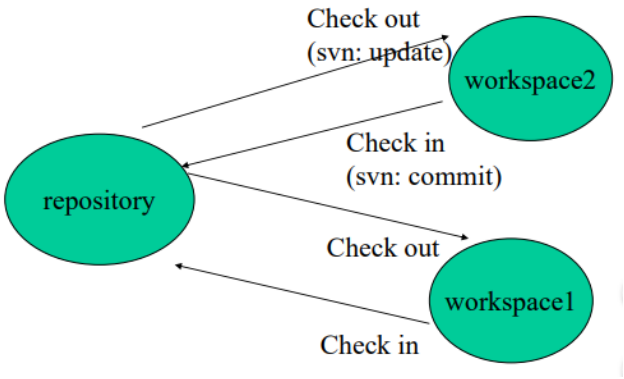

Figure 118 - Check-in and Chec-out operations

The use of a Configuration Manager Server is safer compared to a shared folder. With a CMS, to modify something, the user needs to check it out, notifying the other users coming next that the file is under the modification of some one. After the modifications, files have to be check in again against the repository, and the modifications performed are compared to others already made by other users. With shared folders, there is no such control over files in the repository: everyone can access and modify any file, without any other user knowing it.

CMS are not only useful for Change Control. As a Configuration Management system, they can:

- Revert files back to a previous state;
- Revert the entire project back to a previous configuration (or baseline);
- Compare and monitor changes over time;
- Monitor who last modified something, and if such modification introduced any issue.

Check-ins and Check-outs are regulamented and several choices are available for them: first of all, they can be inhibited for a set of users, thus introducing privileges in the system. A checked-out CI can be locked or not: if a CI is locked, only one user at a time can modify it, while many users can still read it. A checed-in CI can increment its version or not at each modification: if it is not so, the old versions are lost at each modification; otherwise, the history of the CI is maintained for possible rollbacks.

Based on these three main choices that can be made about check-ins and check-outs, two main control strategies can be identified: Copy Modify Merge and Lock Modify Unlock.

- **Lock Modify Unlock** is like a serialization of the changes. A developer tries to win a lock over the CI using the check-out operation; if nobody already has the lock on the CI, then it can be changed by the developer who requested the lock; otherwise, the developer cannot checkout the CI and must wait for lock release. The lock is released only when the developers holding it checks in.
  The main problem of the Lock Modify Unlock approach is that if the locker forgets to unlock a CI, no other developer can modify it. Also, there is no possibility of parallel working: just a single developer can work on a CI. Hence, this approach can be too rigid for large projects and large teams of developers.

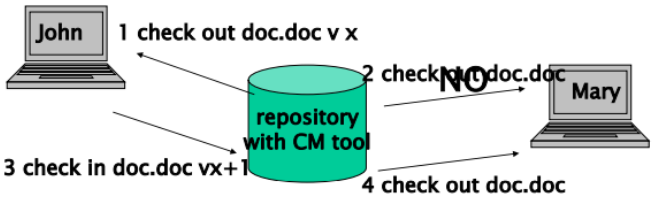

Figure 119 - Lock Modify Unlock

- The **Copy Modify Merge** is a less rigid strategy, that allows many developers to check out the same file and then perform parallel work on it. The only issue is the necessity of confronting and merging the changes introduced by two or more developers on the same file, which can include conflicts (there are tools that perform automatic merge on modified CIs).

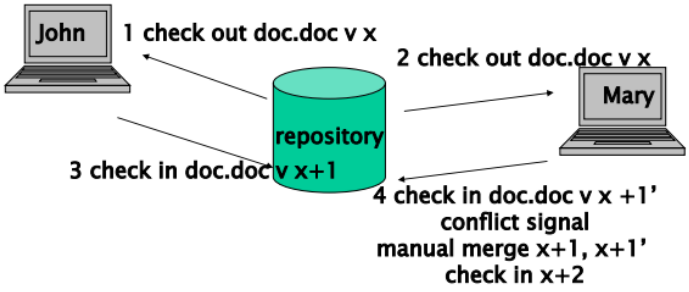

Figure 120 - Copy Modify Merge

Another choice that has to be done is where to store the CMS. Three main strategies are available:

- A **local** repository (e.g., RCS) is stored in the same machine of the user's workspace. A local repository allows only to perform version control, not to share files among many developers. Hence, there is no need of change control mechanisms.

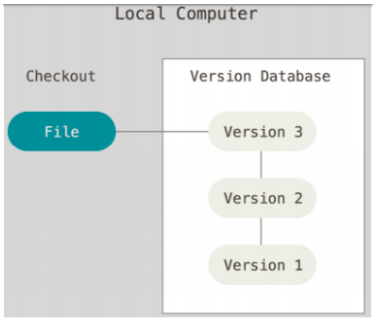

Figure 121 - Local Repository

- In the **centralized** approach (e.g., Subversion, Perforce, CVS) the repository is on a single server that keeps the history of files, while many users (clients) can access it and check out CIs. The main problems in a centralized approach are that a developer cannot work without connection, and that the central server is a single point of failure of the system.

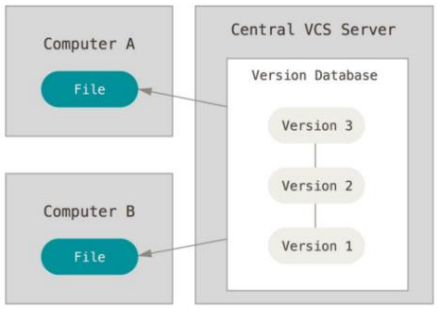

Figure 122 - Centralized Repository

- In the **distributed** approach (e.g., GIT) the repository is mirrored on all servers/clients. This approach tries to overcome the problems of the centralized one, storing also in the clients the history of configurations. A central server, anyway, is considered to make the architecture simpler (each modification is pushed to the server, and each client is synchronized with the server).

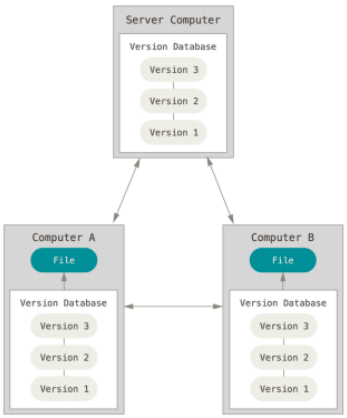

Figure 123 - Distributed Repository

7.3.3 Branches and Merges

A *branch* is like a thread of the development, a duplication of an object (or of the entire project) under change control, so that parallel modifications are permitted. Versioning applies also to branches, and it is fundamental to keep trace of the common origins of parallel branches.

Typically, every project has at least two branches: one used by developers to perform defect fixes, and another to add new functionalities.

When parallel operations on code are performed on different branches, it is often necessary to *merge* them, i.e. obtain a new single branch that includes all the parallel modifications (or a selection of them, after some have been discarded).

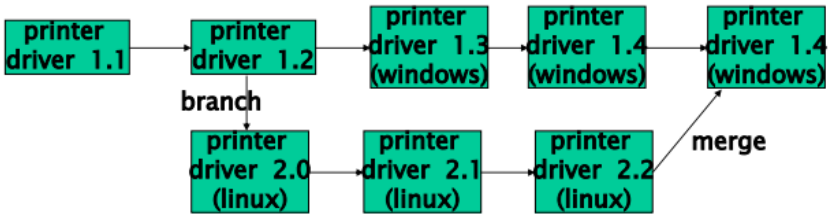

Figure 124 - Branches and merges

7.3.4 CM Planning

Configuration Management should be carefully planned and documented, to avoid issues in the operations performed on different Configuration Items and Branches of the project. Avoiding to document the Change Management operation would put the development team at risk of performing conflictual or unnecessary work.

A **Configuration Management Plan**, that can be written according to various existing templates, contains key Change Management related choices and policies for a project:

- If a Configuration Management tool is used, and which one (e.g., CVS, RCS, Subversion, Git, IBM Clearcase, Microsoft Bitkeeper);
- Which documents should be treated (or not) as Configuration Items;
- The organization of the project in repositories and workspaces;
- The change control policy over defined Configuration Items;
- What are the roles and responsibilities for Change Management, especially who is the CM Manager.

As an example, for a product that is a hierarchy of several subsystems (each one an executable and several modules of source files), one responsible can be assigned to each module and subsystems, and one repository can be designated for any subsystem, with check-in/check-out procedures and dedicated workspaces for developers.

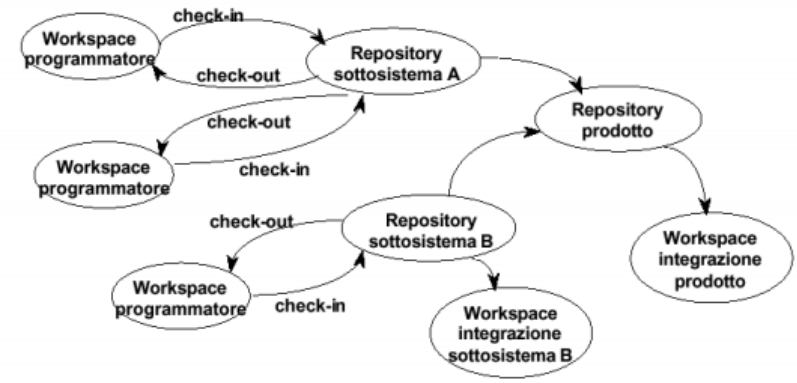

Figure 125 - Organization in repositories and workspaces

7.4 Build

Software Building is the process of compiling and linking software components into a stand-alone, executable form, starting from the possible different combinations of components that can be chosen. Based on the project, the build process can be simple or very complex. For large-sized projects, build can be very boring

and error-prone if performed manually: hence, it is typically carried out in an automated way, and driven by *build scripts*.

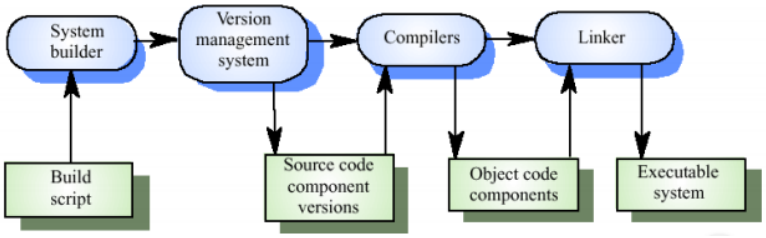

Figure 126 - System building steps

Starting from the build script, the correct versions of the source code are extracted from the Version Management system and passed to the Compiler for compilation. The Object Code components obtained from the Compiler are then passed to the Linker, for the creation of the executable system. The most important part of the building process is to check the component dependencies (i.e., the links between different pieces of code, for instance specified by C includes). Mistakes can be made, as the users of build tools lose track of which objects are stored in which files; *System Modelling* Languages address this problem, by using logical system representations. To build correctly the system, the linker has to check if all the dependencies are consistent, and if all files are present: this is a quite complex operation, and especially in projects with many components it is easy that linking errors are detected and signalled to the developers.

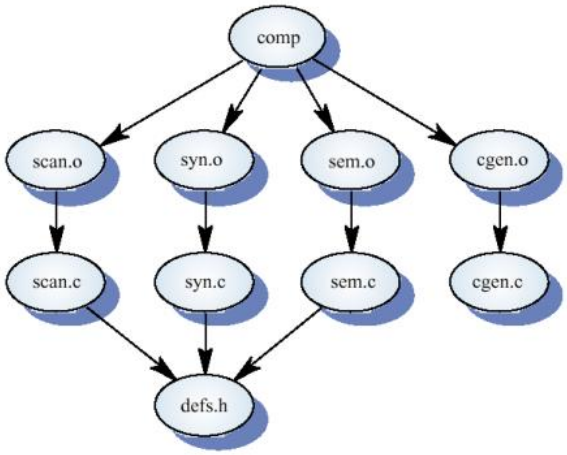

Figure 127 - Component dependencies

The common problems encountered during the build process are:

- Missing components in the build instruction: it is easy to miss one out of many hundreds of components, in very complex systems. As explained before, those errors are generally handled and signalled by the linker;
- Wrong component versions specified: inappropriate versions of individual components are specified in the build script. A system built with wrong components may be built and work initially, but failures can happen after delivery;
- Unavailability of data files: builds should never rely on “standard” data files, that can be missing or vary from place to place;

- Wrong data file references within components: using absolute names in code always generate problems due to naming conventions, since naming conventions may differ from place to place;
- Wrong platform specified for built: many Oss or hardware configurations should require specific settings for the build process;
- Wrong version of compiler (or other build tool) specified: different compiler versions may actually generate different code, and the compiled component may exhibit different behaviour with respect to the desired one.

Automatic build provides also the capability of re-building only the components which have changed: if, in the system tree of dependencies, only a leaf is modified, only the leaf itself and other components depending on it are re-built. This feature is useful to save building times, that may be relevant for large projects.

Some examples of automated building tools are Make, Ant, Apache Maven, Gradle.

7.5 Configuration Management with Git

Git is a system used primarily for Source Code Management in software development, created in 2005 for the development of the Linux kernel. In latest years, Git has affirmed itself as the most used code management system by developers: it has overtaken Subversion among developers' preferences already in 2014, according to a statistic conducted in the Eclipse community.

Git is a distributed Configuration Management system: Git directories in every computer are considered as full repositories. As for the Data Management Model, Git is based on snapshots. Git is based principally on local operations: all the required information is stored in the current comptuer, no other is needed from other computers in the network. Git provides integrity features: checksums are computed at every commit, before anything is stored; no untracked changes of any directory or file exist.

7.5.1 Git States

Documents managed by Git can be in one of three different states:

- **Committed:** data has been stored safely (with checksums) on the local database;
- **Modified:** the file has been changed locally, but modifications have not yet been committed to the local database;
- **Staged:** the file has been changed locally, and has been marked in its current version, to be added to the next commit that will be performed.

7.5.2 Git Project Sections

The files under a Git project can be placed in one of three different sections:

- **Git Directory:** it stores the object database, along with the related metadata, for the whole project. It is what is copied when a repository is cloned, from another computer or from online hosting services (like GitHub). Files that are modified and staged are stored permanently in the Git Directory at each commit.
- **Working Directory:** also called *Working Tree*, it is a checkout of a version of the project. It contains all the files as they are extracted from the compressed Git Folder, and that are stored locally to be used or modified by the developer.

- **Staging area:** also called *Index*, it contains information about all the files that will be part of the next commit. Since the Data Management Model of Git is based on snapshots, snapshots of the files are added to the Staging are when they are staged.

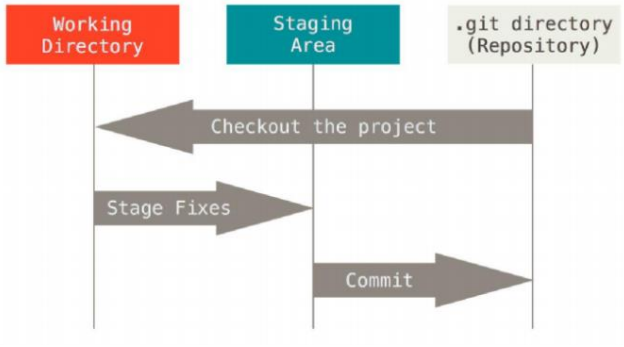

Figure 128 - Typical Git Workflow

7.5.3 File states

File managed by Git must be in one of four different states:

- **Untracked:** files that have been added to the working directory, but that were not in the last snapshot (from which the project was cloned) or in the staging area. They can also be unmodified files, that are removed from Git (with the Git rm command).
- **Unmodified:** all the files of a repository when it is cloned; they are as they were in the last commit.
- **Modified:** files that have been edited since the last commit.
- **Staged:** files that have been selected to go in the next commit.

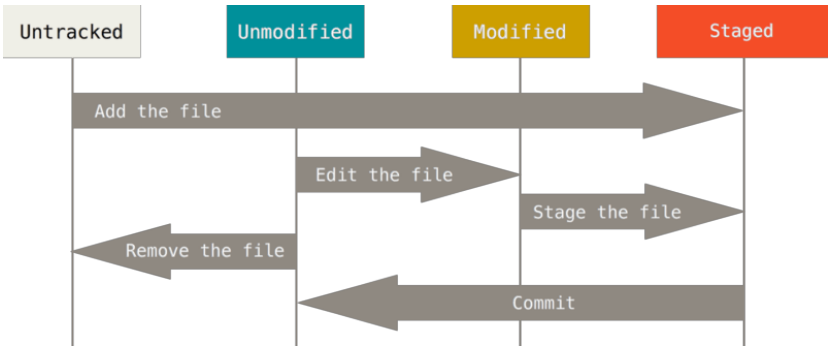

Figure 129 - Lifecycle of files

VIII PROJECT MANAGEMENT

8.1 Concepts and Techniques

A **Project** is a collaborative effort to achieve a goal, with defined limits of time and money (e.g., Manhattan project). A **Program** is the management of several related projects (e.g., Apollo program).

A project is carried out by **Resources:** they can either be persons (the workforce of the company performing development) or tools that are used for development.

An **Activity** is identified as the time that a resource spends to perform a defined and coherent task. A **phase** is a defined set of activities, that are part of a project.

A **Milestone** is a key event or condition in the project, that can also serve as a synchronization point for its transitions between one phase to another. A milestone has effects on the subsequent activities: for instance, a milestone may be the acceptance (or refusal) of a requirement document by the customer. Whether it has been accepted or not, the activities to be performed later may have to change.

A **Deliverable** is a product of the process. It may be either final or intermediate (e.g., a requirements document, a prototype, or the deployed application). A deliverable can be internal, if it must be used only in the producer company, or external, if it is done for the customer or must be validated by it. Some deliverables (like the requirements management or the design document) may have contractual values between customer and producer.

8.1.1 Objectives of PM

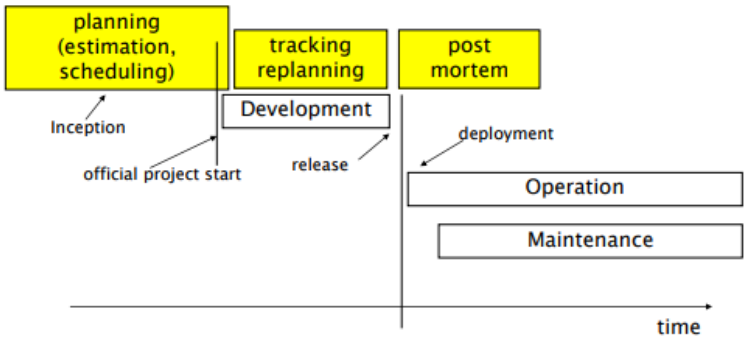

Figure 130 - The PM Process

The main objectives of the project management phase are to characterize the development process, with estimations of how much the project will last, and tracking of the actual time required by its various phases. Project management define schedules and deliveries on calendar: this can be defined upfront to forecast how long the development will take, and can be updated by means of tracking process. Finally, project management is also about team organization and work allocation, i.e. defining who is in charge of performing individual tasks in the development process (resource allocation).

The Project Management process starts before the development itself: projects do not start “in zero time”, but an **Inception** (or **Planning**) phase is necessary. In this phase, the initial requirements and architecture is defined (they can be refined later), along with a preliminary estimation of duration and costs of the project. A commercial proposal is also formulated. Once the proposal is signed (and taken as a technical annex for

legal contracts) the **Development** phase of the project can start. The Inception phase has its costs, that for large projects may be significant: in such cases, Inception costs may be paid to the vendor, as *concept development* fee.

8.1.2 Dimensions of Software Projects

Software projects can be characterized according to three different dimensions:

- **Customers:** the product can be developed for a single customer, or many. In case of a single customer, a product is called *bespoke*: it is, for instance, the case of an application developed for an university, to interface with its students. On the other hand, a product can be developed for many customers: in this case, it is called *commercial* or *COTS (Commercial off the shelf)*. An example of COTS software may be a document editing suite, or an OS like Microsoft Windows.
- **Internal/External development:** the product can be developed in the organization that will eventually use it, or by an external organization. It is more likely that products used by smaller companies is developed externally: larger organizations have full internal IT department, that may be able to perform software projects from Inception to deployment. For instance, an application for students management in a university may be entirely developed from the IT department of the company, while the website for an even big company (e.g., Amazon) may be developed externally.
- **Ownership:** it characterizes the property of a given software. In the case of copyrighted software, the property remains to the producer, whereas the user is given rights to use it (without copying, reverse-engineering or modifying it). On the other hand, the open-source (copyleft) ownership policy gives users full rights to use, copy and modify the software. For instance, the application for students management in a university is property software (since it is used by the same company who developed it); the usage of an operating system like MS Windows in, instead, copyrighted: the users of the software buy a license for using it.

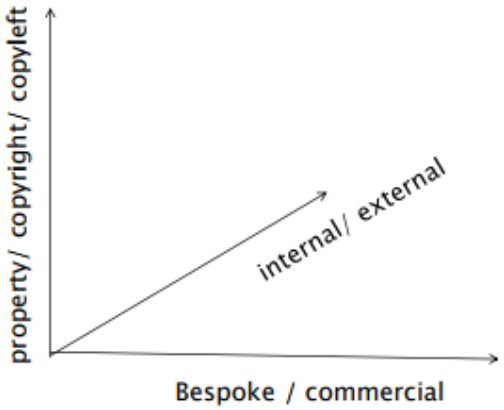

Figure 131 - Dimensions of software projects

The combination of the three dimensions of software projects creates a series different scenarios for software development:

- Bespoke, external, property (new): it is the case in which a company C is in need of a custom software product P. C comes in touch with a software company, which develops it. In a first Inception phase, requirements analysis is performed (supervised by the stakeholders of company C), and a contract is negotiated between company C and the software company. After the contract is signed, the software

company starts the Development phase for the software. The software is finally delivered to company C, who can put it in use.

- Bespoke, external, property (maintenance): it is the case in which a company C has an existing, owned software product P, which is in need of maintenance. The maintenance is performed by the same software company who developed the product. This case is similar to the first one, but typically a contract is signed for a given period of time (e.g., one year) and involves a fixed amount of work for the period, that will be dedicated to maintenance.
- Bespoke, internal, property: it is the typical case of banking and insurance companies. Instead of delegating the development to an external vendor, the software is developed by the internal IT department of the company. In this case, there is no legal contract between a contraent and a contractor, but there are similar negotiation phases, between the user department and the It department of the company (i.e., for what concerns the functionalities or the maintenance effort requested).
- COTS, external, copyright: a company C develops and sells (licenses for) a mass market product. The company has to carry on, in parallel, a maintenance thread (for fixes and patches on the current release) and a development thread (for work on the next release).

8.1.3 Criticalities for a software project

A software project can be characterized by a set of constraints, that can be due to the kind of project and/or to the domain it belongs to. A software project may be classified according to its **Criticality**: safety critical projects are software projects whose malfunctionings may harm the well-being of objects and people; mission critical projects are software projects that are fundamental for an organization or business. Norms and laws are applied to critical software systems (e.g., ISO IEC 61508, defining functional safety standards for software in industry).

Constraints may be posed on a software project also by the domain it belongs to. Particular laws are applied especially for domains that are critical by themselves, like aerospace, medical, automotive, industry, banking, insurance.

8.1.4 Work Breakdown Structure and Product Breakdown Structure

A **Work Breakdown Structure** (WBS) is a chart showing the activities of a project, and the relationships they have to each other. The WBS decomposes hierarchically activities in subactivities. The temporal dimension is not considered: no precedence or consequence relation is given between activities.

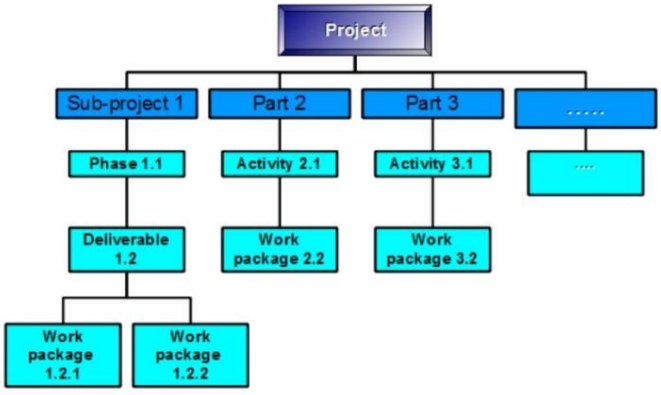

Figure 132 - Work Breakdown structure

A WBS is useful in assisting the key allocation of resources for a project, scheduling, project budgeting and product delivery.

A **Product Breakdown structure** is a hierarchical decomposition of the product; the individual elements of a PBS may be physical or conceptual, and may include documents like requirement specification and safety certifications.

8.1.5 Gantt and PERT charts

A **Gantt chart** is a chart that displays the schedule of a project against time. Each activity or task is represented by a horizontal bar, parallel to a calendar; the start and end point of each bar represent the stacked bars represent overlapping tasks or activities.

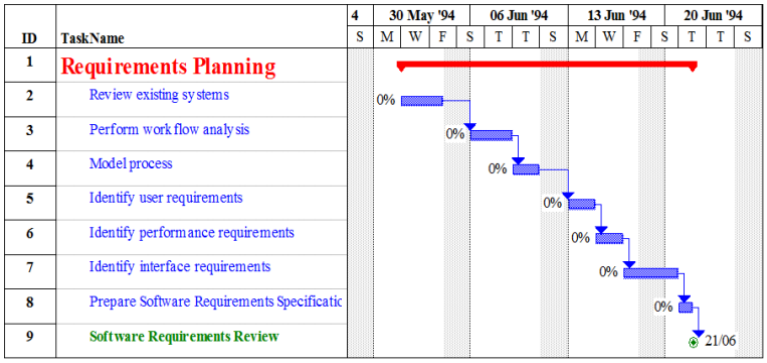

Figure 133 - Gantt Chart

A **PERT chart** depicts the scheduled activities and their dependencies in the form of a directed acyclic graph. Each node represent an activity of the project and each arrow represent a precedent relationship between one activity and another. The scheduled duration of the activity is represented inside the relative node.

Pert charts allow to perform Critical Path analysis, i.e. the identification of the path with the longest duration in the graph, that corresponds to the scheduled total time for carrying out the project. Activities belonging to the critical path are called critical activities, and delays in them would cause the whole project to be concluded late. For activities that are not critical, it is possible through the graph analysis to compute the *Late start* (i.e., the latest time an activity can be started without changing the end time of project), and the *Slack time* (i.e., the admissible delay to complete an activity without changing the end time of project).

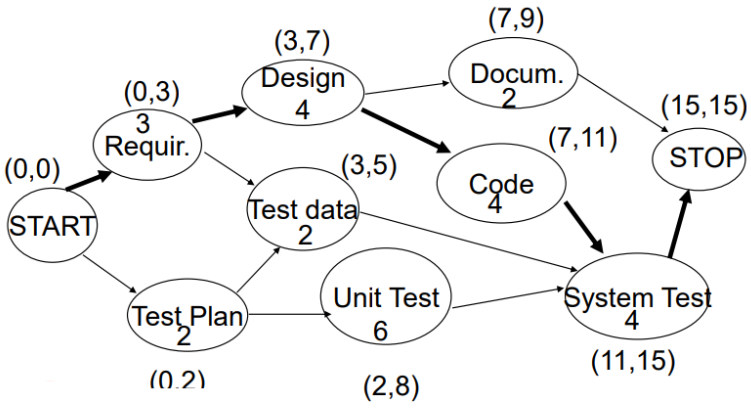

Figure 134 - PERT chart (with critical path highlighted)

8.2 Measures

Measures can be given for both the process itself, and for the product (which is the output of a development process). The former ones aim at characterizing the expenses substained for carrying out the process (for instance in terms of time and effort), and the quality of the process (in terms of faults verified, or changes performed); the latter aim at measuring the functionalities provided by the finished product, or other characteristics like its size, modularity and final price.

8.2.1 Calendar time

Calendar time expresses the duration (in days, weeks, months) of the project. It can be defined either in relative form (e.g., month1 from project start) or in absolute form (e.g., September 12). Relative forms are used typically in the planning phase of the Project Management process, while absolute ones are used typically in the controlling phase.

8.2.2 Effort and Cost

Effort expresses the time taken by the staff working in the project to complete a task. It depends on calendar time, and on people employed in the company. It is measured in person hours by standard IEEE 1045, but other measures are available (like person day, person month, person year: the length of each effort interval depends on national and corporation parameters).

Effort translates into cost, since cost can be computed as person hours times cost per hour. A relation exists also between calendar time and effort, even though it cannot be formulated with mathematical certainty: it is not ensured that a task that can be done in 6 person hours, which is performed in 6 calendar hours by 1 person, can be performed in 1 calendar hour if 6 persons work in parallel.

8.2.3 Cost

Cost is defined differently according to different roles involved in a project: the developer (or vendor) and the user (or buyer).

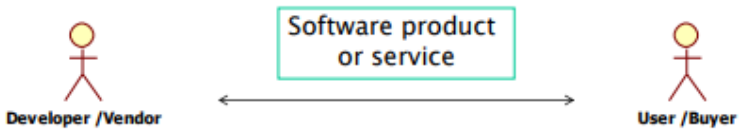

Figure 135 - Costs: roles

During a software project, the vendor has to face the following costs:

- Personnel: salary of personnel and related overhead costs (office space, heating and cooling, telephone, electricity, cleaning);
- Hardware: development platform (may be the target platform);
- Software: licences of operating systems, databases, and tools to be used;

For the user, the cost to be sustained is defined as the **Total Cost of Ownership (TCO)** which considers the complete time window involving the product, from before the Inception to the deployment. Four phases are considered in the TCO:

- Before Acquisition: costs that are substained to define the requirements, and select the product (market analysis, feasibility studies, requirement definition, vendor / product evaluation, contract negotiation);
- Acquisition: the price that is corresponded to the vendor, for an unlimited usage (one time fee) or for a fixed-time fee. The acquisition cost covers the cost faced by the vendor, and its profits. Acquisition cost becomes less important if the time frame in which the product will be used is long;
- After acquisition: deployment costs to install the software in all user machines, and to train the users with an appropriate learning curve; operation costs of servers and network facilities; maintenance costs for collection and correction of anomalies, and evolution of the software;
- Dismissal: costs for uninstalling the product, and performing related activities (e.g., back-up or conversions of data).

The acquisition cost for the user is defined also as *Price*. In general, Price should be equal to the Cost for the vendor, plus a profit. However, it is not possible to define any general relation between cost and price, because price is influenced by many factors (some given in table).

Many factors may influence software pricing, ranging from conditions and opportunities in the market, to volatility of requirements (that may encourage vendors to charge high prices for changed requirements), to the financial health of developers, that may encourage them to make lower profit instead of running out of business.

8.2.4 Size

Size measures can be defined for individual parts of the whole projects: for source code, the typical size measure is the **number of LOCs** (i.e., Lines of Code); for the documentation, size measures can be the number of pages, words, characters, figures or tables; for the test suites, a size metric can be the number of test classes or test cases. To measure the size of the entire project, it is possible to use *Function Points*, and again the LOCs, virtually including in this case also all the non-code documents produced in the development of the application (for instance, if a project produces 1000 LOCs and 10 documents, by convention the project size is still 1000 LOCs, as if LOCs are representative of other documents).

Some conventions must be adopted when using LOCs to measure the size of (part of) a project: it must be decided whether to consider in the computation also comments, declarations, blank lines, definitions of libraries or reused components. It is also not meaningful to compare LOCs for projects developed with different languages (e.g., C vs Java, or Java vs C#).

8.2.5 Productivity

With terms from Economy, productivity is defined as the ratio between *Output* and *Effort*. Hence, for software productivity should be the ratio between the amount of software produced and the effort for producing it. For software, the output can be identified as the number of LOCs, the number of functionalities measured in Function Points, or the number of Object Points (both Function Points and Object points are detailed later).

Using LOCs as a size measure have inherent problems tht are amplified when size is used to measure productivity. In fact, the lower level the language, the more a programmer would be considered productive: writing the same functionality may need few lines of code in a high-level language, and many more in a lower-level one. Programmer would also increase their measured productivity by being verbose: measures of productiity based on LOCs would discourage compactness in code writing. In general, any metric based on

size is inherently flawed for measuring productivity, because the quality of the product is not taken into account.

Different factors can influence the productivity of developers: first of all, experience in the application domain strongly increases productivity, as well as a right support from technology and a quiet and supportive working environment. Large projects, without sufficient process quality, seriously hamper the productivity of developers.

8.2.6 Quality

Quality can be measured as a function of Failures (i.e., malfunctions perceived by te user), Faults (e.g., defects in the system, that may cause failures or not), and fixing Changes that must be operated during development.

For Failures, the data to be collected and measured may be: the calendar time (or project, or execution time) in which they happen, their effect, their gravity, and the related faults identified. Available measures are average time intervals between failures, the total number of failures, the effort needed to close a failure, the number of failures discovered per v&v activity, and the count of failures per class.

For Faults, the data to be collected and measured is: the effects (related failures, if any), the location of the fault, the type of fault (e.g., uninitialized variable), the cause, the detecting method and the effort for finding it. Typical measures for faults are the number of faults per module / development phase, and the fault density (i.e., the ratio between the number of faults and the size of the project). The fault rate can also be characterized by the status of the faults (e.g., opened, resolved, fixed, active).

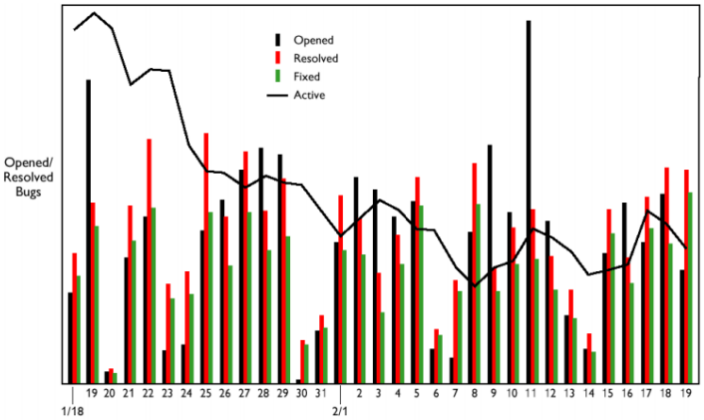

Figure 136 - Fault Rate per status

For Changes applied to the system, the data to collect is: the location of the change, the cause (a related fault if it is corrective, or the reason if it is adaptive or perfective), the effort for performing it. A measure for evaluating changes can be the average amount of changes per document in the project.

8.3 Planning

The Planning phase of the PM process is about identifying the activities and/or deliverables of the software project, estimating effort and cost, and defining and analyzing the schedule (leveraging Gantt and PERT charts). The outcome of the planning phase is the **Project Plan**: it is a *living* document (i.e., it is often updated after the planning phase, during tracking) that includes the list of deliverables, activities, milestones, roles and responsibilities of the project.

Estimation is the first task of the Planning phase: it is about forecasting the cost and efforts that will be required all the activities of the project, from inception to deployment. Estimation is strongly based on analogy, since it requires experience from the past to forecast the future ones: such experience can be qualitative (i.e., non-formalized expertise gained by developers) or quantitative (i.e., data and measures collected from past projects). The more a project is similar to past one, the better the estimation will result.

Estimations can be characterized by their *accuracy*, i.e. the relative difference between the actual costs required by the final project and the forecasted one. The cost, effort and size of a software system can only be known accurately when it is finished (e.g., the size of a project is influenced by several factors, like the programming language that is used and the possible use of commercial off-the-shelf components). As the development processes progresses, estimations can be adjusted, and be come more accurate.

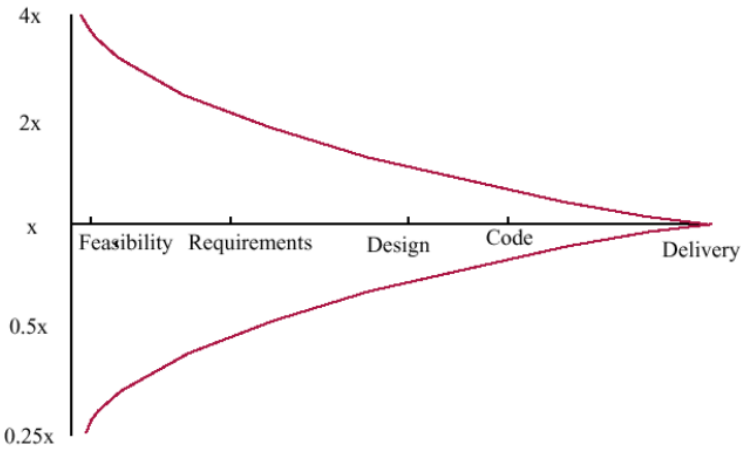

Figure 137 - Estimate uncertainty

Several estimation techniques exist: two simple naive Estimation techniques, Parkinson's Law and Pricing To Win, are not capable of providing accurate estimations but are still widely used. More refined Estimation techniques can be based on judgment (Decomposition, Expert Judgement, Delphi), on data from the company (Analogy, Regression) and on data from outside the compony (Cocomo, Function Points, Object Points).

8.3.1 Parkinson's Law

According to Parkinson's law (formulated by Cyril Northcote Parkinson in 1955, for *The Economist*), the project will cost whatever resources are available. Its implication is that independently from how extensive the resources are, the demands from the system will inevitably grow so that all of them are used (e.g., *the financial demands from the project will increase to exhaust any usable budget*).

The advantage of using the Parkinson's law to estimate the cost and effort required for a project is that it is impossible to overspend with respect to the initial estimate; on the other hand, it can lead to the development of unfinished systems.

8.3.2 Pricing to win

In the Pricing to Win strategy, the project cost is estimated as equal to the resources owned by the customer. The objective is to identify the balance of capability that can be delivered to the customer.

The advantage of adopting the pricing to win strategy is to easily obtain a contract with the customer; on the other hand, there is a low probability that the customer gets the system he or she wants, since the costs substained by the customer do not accurately reflect the work required, that may be more than the estimated one.

8.3.3 Decomposition

The rationale behind decomposition techniques is that it is easier to estimate the cost and effort to produce smaller parts of the system, instead of the system as a whole. Decomposition is supported by WBS and PBS (cfr section 8.1.4):

- Decomposition by activity identifies the activities of the project (in the form of a Work Breakdown Structure), estimates the effort for each activity, and then linearly aggregates the individual effort to obtain an estimate for the whole project.
- Decomposition by product identifies the products of the project (in the form of a Product Breakdown Structure), estimates the effort for each product, and then linearly aggregates the effort to obtain an estimate for the whole project.

Activity	Estimated effort (person days)
Review existing systems	5
Perform work analysis	5
Model process	1
Identify user requirements	10
Identify performance requirements	4
TOTAL	25

Table 21 - Estimation by decomposition: requirements

8.3.4 Expert Judgement and Delphi

In the Expert Judgement estimation technique, one or more experts of the domain of the project, chosen in function of their expertise, propose an estimate of the cost and effort required for carrying out the project.

Delphi is an evolution of the expert judgement technique. It is based on the assumption that the forecasts from a structured group of people are more accurate than the ones coming from unstructured groups. In the Delphi structured meetings each participant proposes an anonymous estimate, and the final estimation is based on the beta distribution of the individual estimates ((a + 4m + b)/6, where a is the best estimate, b the worst and m the mean).

8.3.5 Analogy

The Analogy technique is based on a set of data extracted from projects developed by the same company performing the estimation: a number of attributes, with respective values (e.g. size, staff experience, technology used, duration), is collected for a set of projects.

Once the attributes for the new project are defined, the closest projects (according to a distance function) are identified, and are used to estimate the effort for the current project (e.g., the effort can be estimated as a mean of the closest ones).

8.3.6 Regression Models

Like Analogy, it is based on a database of past projects from the same company, of which information about size and effort is available. A form of regression (e.g., linear) is applied to estimate the productivity, for all projects. For new projects, it is then sufficient to estimate the size to compute the required estimated effort.

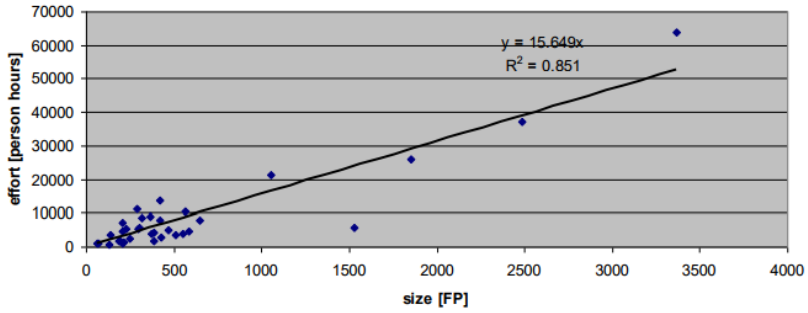

Figure 138 - Linear Regression

8.3.7 Function Points

A Function Point is a tool which helps in approximating the size of a software project. The size in Function Points for any product is a 3-step process.

1 Classify each component of the product in one of five classes and compute their number (see table 22): EI, EO, E, ILF, EIF. Then, classify any component as simple, average or complex, and assign the appropriate number of Function Points (see table 23). The sum of the Function Points of the individual components give UFP (Unadjusted Function Points) for the product.

*UFP = A*EI + B*EO + C*EQ + D*EIF + E*ILF;*

EI	*Number of input items*
EO	*Number of output items*
EQ	*Number of inquiries*
EIF	*Number of external interface files*
ILF	*Number of internal logical files*

Table 22 - Types of items for function points

Component	***Simple***	***Average***	***Complex***
Input Item (A)	*3*	*4*	*6*
Output item (B)	*4*	*5*	*7*
Inquiry (C)	*3*	*4*	*6*
Interface (D)	*5*	*7*	*10*
Internal Logical Files (E)	*7*	*10*	*15*

Table 23 - Function Points coefficients

2 Compute the Technical Complexity Factor (TCF), through the assignation of a value from 0 ("not present") to 5 ("strong influence throughout") to a set of 14 factors, such as transaction rates, portability of the system, and so on (see table 24).
The 14 numbers are then added, obtaining the Total Degree of Influence (DI), based on which the TCF can be computed. TCF lies between 0.65 and 1.35.

*TCF = 0.65 + 0.01 * DI;*

1	*Data communication*
2	*Distributed data processing*

3	*Performance criteria*
4	*Heavily utilized hardware*
5	*High transaction rates*
6	*Online data entry*
7	*End-user efficiency*
8	*Online updating*
9	*Complex computations*
10	*Reusability*
11	*Ease of installation*
12	*Ease of operation*
13	*Portability*
14	*Maintainability*

Table 24 - Factors for TCF

3 Compute the number of function points (FP), given by the formula *FP = UFP * TCF.*

Function Points is a measure of the size and of the complexity of a system. It is a very long and expensive evaluation, that should be always performed by FP experts. Once Function Points are computed, conversion tables exist to estimate, based on them, size or effort measures.

Function Points and LOCs are both size measures. As LOCs, FPs can be computed before a project starts and after a project ends (respectively, for estimated and actual size), and can be used to characterize the productivity (in terms of FP/effort) and to present offers for potential customers (in terms of price per function point of the developed system).

The main advantage of using function points is that the provided estimation is transparent to the technology used for actually developing the system, and independent of the programmers. It is also a well established and standardized techniue. The principal downside of Function Points is that by definition they are oriented towards the estimation of Transational systems, so they are not suitable for real-time or embedded systems (on the other hand, a size estimation based on LOCs is suitable to all kinds of systems).

	Function Points	**LOCs**
Depend on prog. language	No	Yes
Depend on programmer	No	Yes
Easy to compute	No (need training)	Yes (tool based)
Applicable to all systems	No (transaction oriented)	Yes

Table 25 - Function Points vs LOCs

8.3.8 COCOMO

COCOMO (Constructive Cost Model) is a well-documented and "independent" model for predicting the effort and schedule for software development. It has a long history, since it original version, published in 1981 by Barry Boehm (COCOMO-81): various instantiations have led to COCOMO 2, which takes into account different approaches to software development.

The original **COCOMO-81** model is based on a dataset 63 projects of different nature and domain (scientific, MIS, embedded systems), that was enriched with time. Waterfall development process was assumed: a defined sequence, without feedbacks, of requirement analysis, design, implementation, integration and test (cfr. Waterfall Model in chapter IX). COCOMO-81 model allows the computation of an estimate of *PM (*effort to develop the project in person months) based on *KDSI (K Delivered Source Instructions)* and a multiplier M. Table 26 shows the formulas for computing PM, according to the complexity of the project. The M multiplier

is obtained based on a combination of Drivers related to the project (e.g., required software reliability, database size, product complexity).

Project Complexity	Formula
Simple	PM = 2.4(KDSI)^1.05 * M
Moderate	PM = 3.0(KDSI)^1.12 * M
Embedded	PM = 3.6(KDSI)^1.20 * M

Table 26 - COCOMO-81

COCOMO 2 (presented in 1997) is a 3 level model, that allows increasingly detailed estimates to be prepared as development progresses. Estimations are based on different formulas according to the level they are performed at.

- **Early Prototyping Level:** the estimates are based on Object Points (a way of estimating effort similars to source LOCs), and a simple formula is used for effort estimation (in Person Months). This level supports projects where the reuse is extensive.
- **Early Design Level:** the estimates, made after the requirements are agreed, are based on function points, then translated to LOCs. Multipliers in the formula take into account the novelty of the project, development flexibility, risk management approaches used, process maturity, reliability and complexity of the product, platform difficulty, personnel experience and acpability, team support facilities. The resulting PMs reflect also the amount of automatically generated code.
- **Post-Architecture Level:** the formula is the same used in early design, but it is adjusted to tae into account the volatility of reuirements, the rework required to support change, and the extent of possible reuse in the development of the system.

8.3.9 Scheduling

In the planning phase, as well as effort estimation, calendar time must be estimated and staff allocated. The scheduling procedures can be performed taking advantage of **Gantt / PERT charts** (cfr. paragraph 8.1.5).

An estimate of calendar time, independent of staffing, is also given by COCOMO-2, which provides a formula for **TDEV**:

$$TDEV = 3 * (PM)^{(0.33+0.2*(B-1.01))}$$

where PM is the effort computation in Person Months according to COCOMO-2 formulas, and B is an exponent depending on five different scale factors (precedentness, development flexibility, architecture / risk resolution, team cohesion, process maturity). The TDEV measure predicts the nominal schedule for the project.

Staff required cannot be easily computed by dividing the development time by the required schedule: the number of people working on a project varies depending on the phase of the project. The **Staffing Profile** (i.e., a graph indicating the number of people working on the project vs. time) has typically a bell shape. The duration of the project is constrained by the staffing profile and the total effort estimated.

In general, the more people working on the project, the more total effort is usually required. Very rapid build-ups of people also may correlate with schedule slippages.

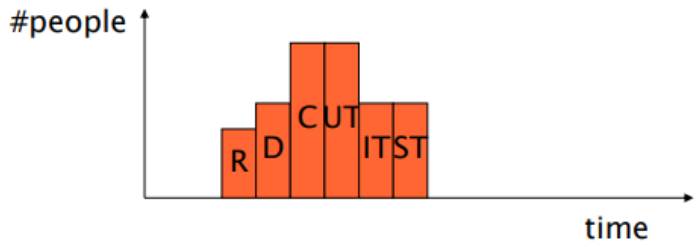

Figure 139 - Staffing Profile

8.4 Tracking

Once the project has started (i.e., the Planning phase has been completed) procedures must be performed in order to monitor its status. The Tracking process is a set of activities with the objective of collecting project data, defining the actual status of the project, and compare it with the estimated one (for instace, the actual roadmap of the project is compared to the Estimated Gantt). If deviations are detected, corrective actions must be performed (e.g., change activities, deliverables or personnel working on the project). At each corrective operation performed, and at each deviation detected, re-planning activities and updates of the Gantt and PERT diagrams must be done.

The status of a project can be expressed in three different ways: as the effort spent, as a combination of spent effort and closed activities, or as the earned value during the process.

- **Effort spent** tracks the process by just measuring the amount of effort spent in carrying out the already performed activities. The effort spent is compared with the estimated, to understand how much work has been completed and how much is still to perform. The limit in using effort spent as a tracking measure is that it confounds an input measure (the effort) with an output measure (the completion of the product): the relation between effort and completion may vary –even considerably- among different phases of development.
- The **Activities Closed** can be used as tracking estimation, and are more reliable than effort spent, being themselves an output measure. There are two different ways to define when an activity is closed: if all the effort planned for the activity has been spent (but again in this case an input measure is confounded with an output measure); or when a defined uality gate or level for an activity is achieved (e.g., 95% coverage of nodes for unit testing, majority in accordance for requirements inspection).
- **Earned Value** is a measure based on the completion of activities of the project, and is used to measure the progress of a project.

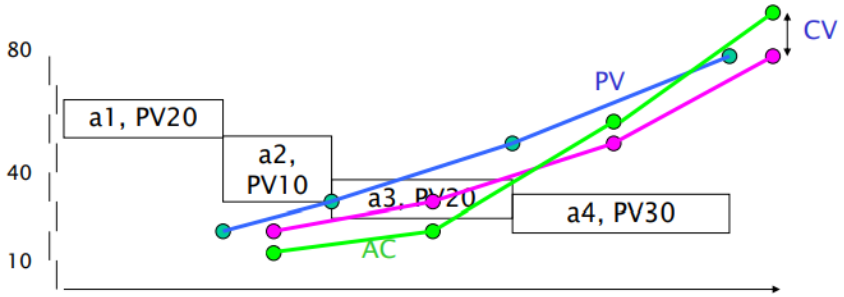

Figure 140 - Earned Value

All the activities of the project are identified and scheduled, and each is assigned a Planned Value PV. Then, a rule is defined for computing Earned Value based on PV and the level of completion of activities: for instance, a 0/100 rule would assign the project an earned value equal to the PV of an activity when the activity is finished; a 0/50/100 rule would assign the project an earned value equal to half PV of an activity when the activity is half-completed, and an earned value eual to the full PV of an activity when the activity is finished. Earned Value and Planned Value are then plotted against time, along with the Actual Cost (AC) of the project.

8.5 Post Mortem

Post Mortem Analysis (PMA) is a process performed at the end of a project, to collect various kinds of key information from the project: estimated and actual efforts and faults, achievements obtained, problems and their causes. It is a form of organizational learning, that –through the understainding of a finished project- aims at making information available for future ones, mitigating future risks and promoting experienced best practices.

When used appropriately, a PMA ensures that the team members reflect on what happened during the project, and recognise and remember what they had done and learned. Improvement opportunities are identified, and means are provided for initiating sustained change in the development process.

Two types of PMA exist: general PMAs, that can applied to any kind of project, and focused PMAs, that are aimed at understanding and improving a project's specific activity.

8.5.1 PMA Process

The PMA process is composed by four main activities: Preparation, Data Collection, Analysis, and Experience Documenting.

- In the **Preparation** phase, the project history is studied, through a review of all available documents, to understand what has happened. The goals for the PMA (e.g., identify major project achievements and further improvement opportunities) are identified.
- In the **Data Collection** phase, all relevant project experience is gathered, with a focus on negative and positive aspects of the project. During the data collection phase group discussion is conducted through semi-structured interviews and KJ sessions, in which every participant writes down up to four positive and negative experiences on post-it that are then attached on a whiteboard, re-arranged in groups and discussed.
- In the **Analysis** phase, feedback sessions are performed, to know whether all the relevant facts have been elicited, and if the analysts have understood what the project members have told. The Ishikawa diagram can be used to find causes of positive and negative experience during the project.

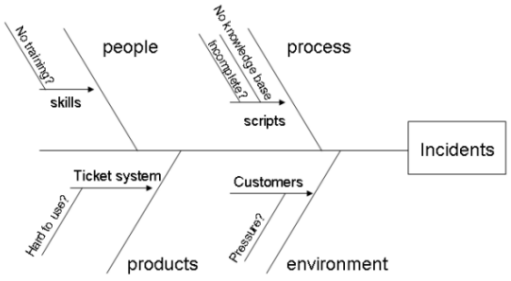

Figure 141 - Ishikawa Diagram

- In the **Results and Experience** phase, the PMA results are documented in a project experience report, which includes a description of the projects, the main problems encountered (with Ishikawa diagrams), the main successes obtained, and the scribe of the meeting as an appendix.

8.5.2 Collecting and using measures

A process should be defined and implemented to collect data, derive and analyze measures. Data collected during this process should be maintained as an organizational resource, so that comparisons across projects can become possible based on the measurement database. The product measurement process should define clearly the measurements to be made and the components on which they will be collected; after the components are actually measured, anomalous measurements should be properly identified and analysed.

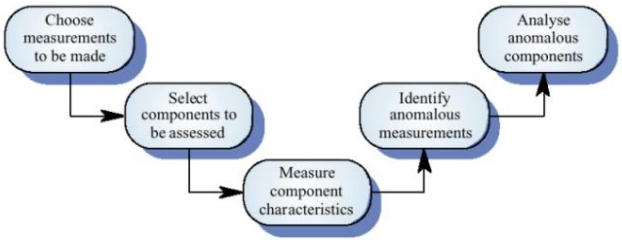

Figure 142 - Product Measurement Process

A tool for collecting data is the **GQM** approach: it is a top-down approach which aims at focusing on a few important measures, instead of collecting anything and then analyzing all (even unnecessary or non-meaningful) data. The fundamental idea is that an objective (Goal) is defined first, then comes the formulation of a set of questions whose answers can help determining if the goal is achieved or not. Questions are answered through pertinent metrics.

Data collection should be performed immediately (not in retrospect) and, if possible, automatically, to reduce possible errors on metrics measured; data should also be controlled and validated as soon as possible.

Once collected and analyzed, data should be presented through reports, web reports or dedicated dashboards.

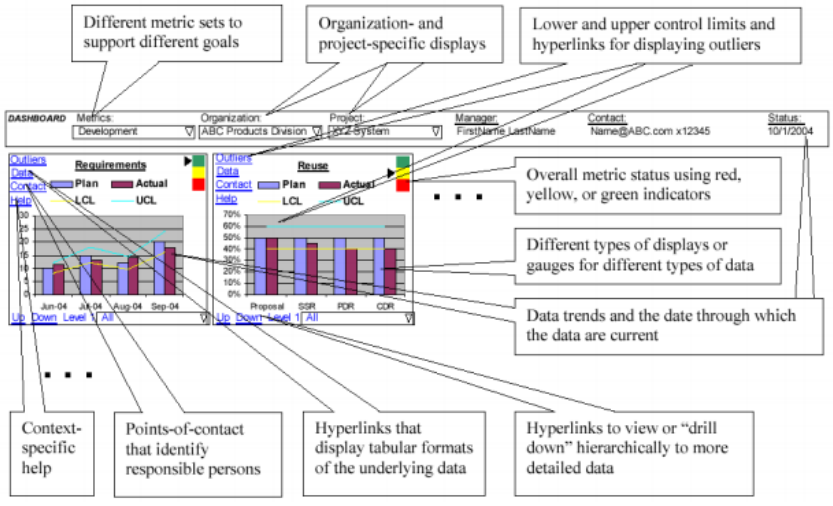

Figure 143 - Data Presentation Dashboard

8.6 Risk Management

Two different strategies are possible for approaching risks in the software process: the *Reactive* strategy copes with risks as soon as they happen ("fire-fighting mode"); the *Proactive* way, also called Risk Management, aims at identifying, analyzing and quantifying the effects of risk up-front, defining strategies and plan to handle them.

Risk management aims at defining, for each risk, its impact, probability, control, and exposure:

- **Risk Impact** is the loss associated with the event described by a risk;
- **Risk Probability** is the likelihood that the risk will occur during the process;
- **Risk Control** is the degree to which the outcome of an occurring risk can be changed;
- **Risk Exposure** is computed according to a combination of probability and impact.

8.6.1 Categories of Risks

A **Risk** is a future event that can have bad impact on a software project. Risks can first of all be categorized according to their nature:

- A **Project Risk** may either regard the project plan (budget, personnel, timings, resources, or customers), or management (wrong support from management for the project, missing budget or people);
- A **Technical Risk** regards the feasibility of the project and quality factors for technical components (e.g., scalability, security, availability);
- A **Business Risk** regards the market or the company: a market risk is the possibility that the product will have no market once developed; a strategic risk is the possibility that the product is not in scope with the company plans; a sales risk is the possibility that the sales department will not be able to sell properly the finished product.

Any kind of risk can arise from three possible cases:

- A **Known Risk** is identified before or during risk management (e.g., unrealistic deadlines or missing requirements);
- A **Predictable Risk** is a risk that the team is aware from previous experience, but it is still not known if it exists in the project or not (e.g., poor communication with customer or personnel turnover);
- An **Unknown Risk** is a risk about which the organization has no idea. They are typically linked to technology.

Personnel shortfalls	instability of COTS (Commercial Off-The-Shelf) components
Unrealistic schedules and budget	interface with legacy
Developing the wrong functions	stability of development platform (hw + sw)
Developing the wrong user interfaces	limitations of platform
Gold-plating	multi-site development
Continuing stream of requirements changes	use of new methodologies / technologies
Shortfalls in externally-performed tasks	standards, laws
Shortfalls in externally-furnished components	development team involved in other activities
Real-time performance shortfalls	Communication problems
Straining computer science capabilities	language problems

Table 27 - Common Risks

8.6.2 Phases of the RM Process

The risk management process is subdivided in two main phases, Assessment and Controls, which are further subdivided into a total of five activities.

Risk Assessment	Identification
	Analysis
	Ranking
Risk Control	Planning
	Monitoring

Table 28 - RM Process

In the **Identification** phase, risks are identified starting from checklists, taxonomies and questionnaires available in literature; brainstorming between members of the team and experience of members involved in previous similar projects also is useful in this phase.

In the **Analysis** phase, for each risk its probability, impact and exposure is computed. The probability of the risk is measured in a scale of five values (very high, high, medium, low, very low), while the impact can be one of four different values (catastrophic, critical, marginal, negligible). The exposure value can be derived based on probability and impact, with the aid of standard tables.

Impact/Probability	Very high	High	Medium	Low	Very low
Catastrophic	High	High	Moderate	Moderate	Low
Critical	High	High	Moderate	Low	Null
Marginal	Moderate	Moderate	Low	Null	Null
Negligible	Moderate	Low	Low	Null	Null

Table 29 - Exposure of risks

In the **Ranking** phase, risks are ordered by exposure and by qualitative assessments. Only higher exposure risks are handled.

In the **Planning** phase, for selected risks (those with high exposure) corrective actions are selected, and the cost for those actions is evaluated, to decide if they are acceptable or not. If so, corrective actions are inserted in the project plan. For each risk, three possible strategies are possible: avoiding the risk, i.e. changing requirements to stop considering the risk; transferring the risk, i.e. transferring the risk to another system or buying insurance; assuming the risk, i.e. accepting its existence and controlling it.

In the **Monitoring** phase, the project plan is followed, and corrective actions for found risks are taken. The ranking of the risks is updated every time a new risk is identified. In the monitoring phase, the risk log document is computed, and the risk reviews are performed.

IX PROCESSES

The process modeling concerns how to organize all phases and activities of development and maintenance, and the techniques that can be used to perform them.

The **ISO/IEC 12207** standard defines the main processes in software development, along with the entities responsible of them, and their products.

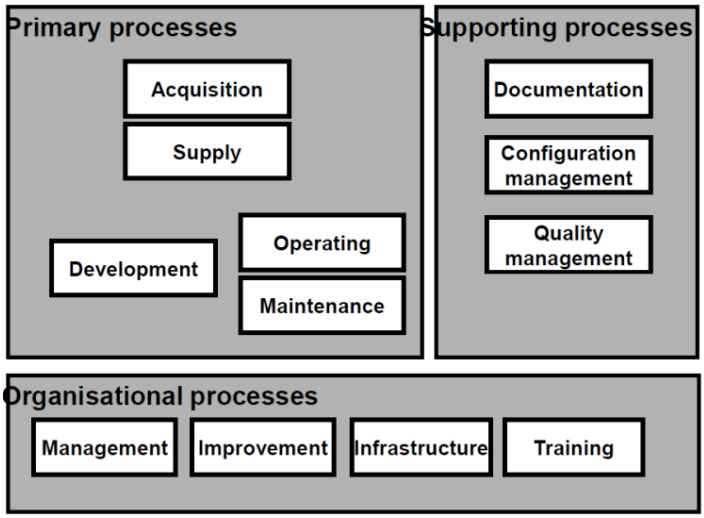

Figure 144 - ISO/IEC 12207

Five main processes are identified: acquisition (the management of suppliers); supply (the interaction with customers); development of software; operation (deployment, and operation of services); maintenance of services. Three supporting processes helping the primary one are defined by the standard: documentation of the product, configuration management, and quality assurance (i.e., verification and validation, internal audits, reviews with customers and problem analysis and resolution). Finally, as organizational processes are identified: project management, infrastructure management (of facilities, networks and tools), process monitoring and improvement, and personnel training.

In the ISO standard, the processes are divided into activities (e.g., among Development, one activity is software development). Activities are again divided into tasks: for instance, the *v&v* activity, that is part of the *Quality Assurance* process, is composed by five tasks: *coding and verification of components, integration of components, validation of software, system integration, system validation*. Finally, tasks are divided in subtasks: for instance, the *coding and verification of components* task is further divided in *definition of test data and test procedures, execute and document tests, update documents and plan integration tests, evaluate tests.*

The ISO standard only enumerates the activities and tasks: it does not define the way to perform them. For this reason, the standard is independent of the adopted process model, technology, application domain and documentation.

9.1 Process Models

The software process is made by many activities. The role of a process model is to organize them, assigning tasks and giving temporal constraints to each one. Many process models exist, originating from standards

and documents (e.g., Iso 15288, Iso 12207, Iso 9001, Iso 9000-3, CMM-I) or literature (e.g. Waterfall, RUP, Agile).

In general, every company has its typical approach, the *company process model*, with the definition of suggested or required activities, documents and milestones for every project carried out by the company. However, the process model cannot be the same for all the projects of the same company: the *project process model* depends on each project, based on its criticality, cost, size, technology and application domain (who enforces standards and laws). The selection of a specific process model for a project is reviewed and enforced by the quality team of the company. During the execution of the process, the *process conformance* can be checked: it is the consistency between the actual process followed in a project, and the process model defined in the documents.

Process models can be divided according to the documentation they produce. There are three types of models: document based (sequential, iterative, and so on), Agile, and Formal.

9.1.1 Build and Fix

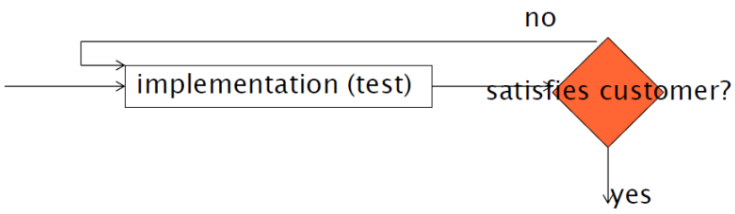

Figure 145 - Build and Fix

Build and Fix is the typical solo-programming model, that fits only very small projects. There is no requirements description, no design nor validation. There is often only one developer who writes code then tests it, checking for errors and fixing them until the software is stable. The model obviously does not scale up for larger projects.

9.1.2 Waterfall

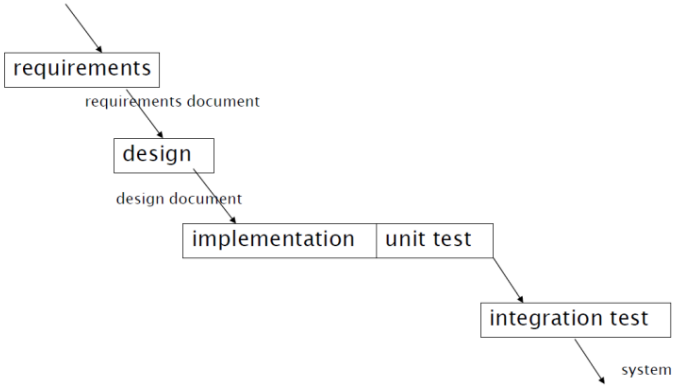

Figure 146 - Waterfall Model

The Waterfall model (Royce 1970) advocates a strictly sequential flow of activities. The irst one is the documentation of requirements. Then, when requirements document are ready and completed, the design activity starts, followed by the actual implementation, and finally the testing procedures. In the Waterfall model, the next activity is started only when the previous is over, and freezes the deliverable: the output of

the previous activity cannot be changed anymore. Any change to the previous steps implies re-doing all the activities depending on its output (e.g., having changes in the requirements means redoing all the activities). The Waterfall model is strongly document-driven.

The main problem of the waterfall model is its lack of flexibility: no changes can be made (without heavy impacts) on documents, and this can create tension to avoid changes that are often necessary to solve errors. Furthermore, the worst impact changes are the ones in requirements and design, that normally are the most faulty. The Waterfall model is also a strongly burocratic process model.

9.1.3 V Model

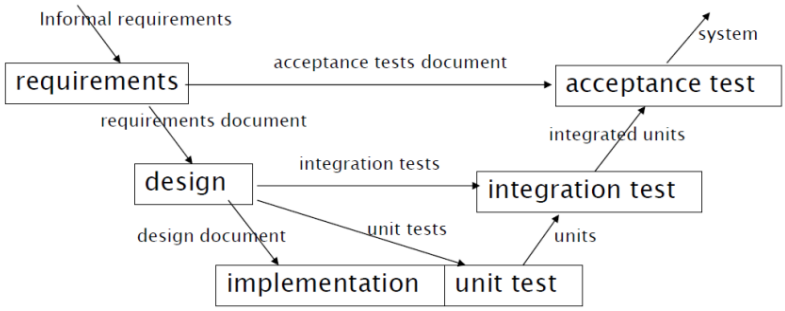

Figure 147 - V Model

The V Model is a sequential model similar to the waterfall, but gives emphasis on the V&V activities. Tests are made for each activity: acceptance tests are made with requirements, and unit/integration testing during the design phase. For this reason, the V model is faster than the waterfall and then ofter preferred.

An example of usage of the sequential V model is ISO 26262 / IEC 61508, a standard for the development of road vehicles system, in which the emphasis is on functional safety. There are two different lifecycles, one related to safety and one related to software: the standard uses a waterfall model for safety, and a V model for the software lifecycle. This is due to the fact that, for what concerns safety, requirements must be clear, well written and fxed at the start of the process. Many company will produce components conformant to the standard, and all those must be coherent with what was defined in the requirements. In general, the application of sequential models is very common in safety critical software development.

9.1.4 Prototyping + Waterfall

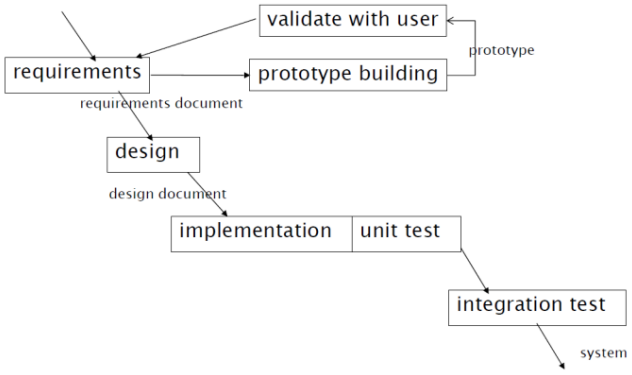

Figure 148 - Prototyping + Waterfall

Another variant of the sequential process model is given by the Prototyping + Waterfall model. In this model, there is a first step of quick and dirty prototyping of the system, to perform a first validation and analysis of the requirements. After the initial phase, the process proceeds as the waterfall model.

The prototyping step allows a faster development of the system, because the requirements should be already checked and tested after it, then it is less likely that requirements will still contain errors and that all the subsequent waterfall phases will have to be re-done. However, it may be difficult and may require specific skills to build the prototype (with specific prototyping language) and if the prototype is successful the business departments may make pressures to keep it as the final deliverable, skipping all the remaining phases of the process.

For a software, the prototype may be a similar software with less functions, or developed on a different platform (easier to develop on). The idea of prototyping can be applied to other parts of a project rather than simply to code: e.g. GUI prototyping with the creation of mockups, design prototyping, or prototyping for performance measurements.

9.1.5 Incremental

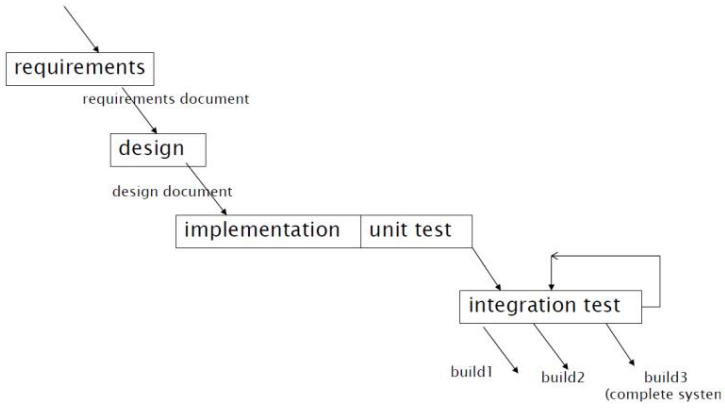

Figure 149 - Incremental model

The incremental model is similar to the waterfall model, but the integration (that are at the end activities producing the entire system in Waterfall) is split up: every loop in the integration model produces a part of the system, that is delivered in several increments. Each increment is called *build*: all the builds except the latest one are partial, and integrate only some functionalities of the system; the final build is the complete system delivered to the user.

In the iterative model each iteration is planned and incremental, but concerns only the final step of the development: if the requirements (or the design) change, all the waterfall (and the code implementation) should be done again. The advantages of the incremental model are an earlier feedback from the user or customer, and the possibility of delaying increments that depend on external components.

9.1.6 Evolutionary

The evolutionary model solves the problem of re-doing the whole waterfall that is still present in the Incremental model: it is similar to the Incremental model, but reuirements can change at each iteration. In some ways, it can be associated to the prototyping model, since at each iteration an incremental prototype is produced. The last prototype developed is the final product.

The evolutionary model can be very appreciated by the stakeholders, since it allows them to change their requirements after they have seen the partial results. The evolutionary model can cause issues to the development company, since changing requirements means a significant amount of work to be performed. To cope with this problem, the contractual agreements may be on effort and not on provided functions.

9.1.7 Iterative

The iterative model is an extremization of the evolutionary model. This model consists in many iterations of a single process, modeled like a waterfall-like process.

The difference with respect to the evolutionary model is that at the first iteration not all the requirements are written, but also the are made increentally, whereas in the Evolutionary model the requirements are written at the first loop and then can be modified to adapt to user's wills.

The assumption on which the iterative model is made is opposite to the one done in the Waterfall model: in the Iterative approach, things are made fast, knowing that it is not possible to do them perfectly, and then are reviewed later. The iterative model is the one able to produce software in the smallest amount of time: partial deliverables can still be objects of evaluation, even if the product is not finished.

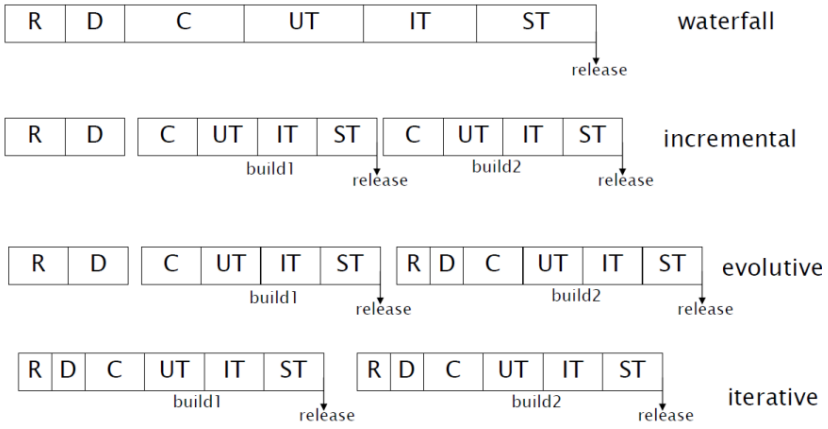

Figure 150 - Comparison between process models

9.1.8 (R)UP

(R)UP (Rational Unified Process) is an example of application of the iterative model. The model is an incremental odel based on architecture, developed by Rational software (now part of IBM) for big software projects. The model shows how, with the iterative approach, the work can be parallelized.

RUP has 4 main phases:

- **Inception**, in which are performed the feasibility studies, risk analysis, essential requirements elicitation and business cases (i.e., use cases for the market). It is also possible to perform prototyping (not mandatory);
- **Elaboration**, that aims at defining the structure of the system. The domain analysis and the first step of architecture design belong to this phase. At the end of this phase, the project plan and an architecture description must be defined;
- **Construction**, the phase in which the real development takes place. In this step, the first version of the system, with the first implemented features, is realized;

- **Transition,** the last step that concerns the deployment of the system to users, with the performance of beta testing, performance tuning, training, writing of manuals, packaging.

Iterations take place at each step. This breakdown has many advantages, regarding the project evaluation and risk analysis.

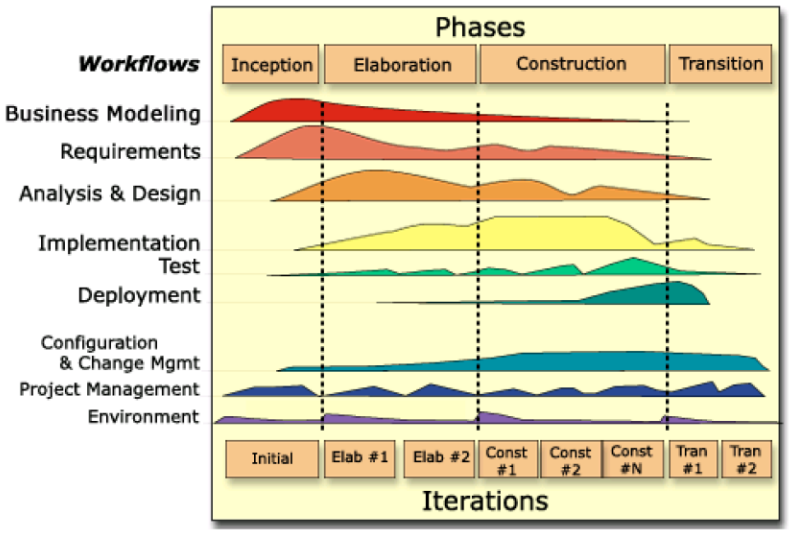

Figure 151 - (R)UP model

9.1.9 Synch and Stabilize

Another example of an iterative model is **Synch and Stabilize**, developed by Microsoft between 1993 and 1995, and driven by the necessity of a fast time to market. It is indicated for complex products (with up to millions lines of code) with several interacting components, whose design and requirements is difficult to be fixed and frozen early on.

1	Divide large projects into multiple cycles with buffer time (20-50%)
2	Use a "vision statement" and outline feature specifications to guide projects
3	Base feature selection and priority order on user activities and data
4	Evolve a modular and horizontal design architecture
5	Control by individual commitments to small tasks and fixed project resources

Table 30- Synch and Stabilize principles

The approach is based on parallel work made by small teams (three to eight people), with frequent synchronization and a 1:1 ratio between testers and developers. Three main phases are identified:

- **Planning:** mostly requirement elicitation activity, in which the goals or the new product are defined, and the functions to be implemented are ranked according to a prioritization of the user-activities. The result of this phase isa specification document, a schedule of the activities to be performed and the formation of a "feature team", composed of 3-8 developers and an equal number of testers.
- **Development:** an iterative phase in which three or four subprojects, each one lasting from 2 to 4 months, are planned. Between each iteration there is a buffer time, to ensure that all projects are closed in the iteration. Many feature teams are assigned to each subproject: during the subproject,

they go through the complete cycle of development, feature integration, testing and fixing. At the end of each subproject, the product is stabilized.

- **Stabilization**: it comprises internal testing of the complete product, and various forms of external testing (e.g., with beta sites and end users). The release of the software is prepared in this phase.

9.1.10 The City Model

The City Model (Kazman 2014) is a special model used when there is no clear boundary of the software. This is the model used by Wikipedia, Google, Open street Maps, Twitter and many other social networks. The content (or the code) is made by the users, and the requirements are not known but emerge from individual contributions.

In this model there are no releases, no planning of software evolution, and the software evolves based uniquely on the user content. The system is always changing. This leads to the concept of *emergent behavior*, i.e. unplanned behaviors and functionalities of the system. An example is Twitter, originally born only to tweet emotions and thoughts, and then become a coordination tool for masses.

The kernel of projects developed using the City Model is often closed, slow to change, reliable and always available; it offers APIs so that users can enrich the content of the city, in the *periphery*. The periphery is an users' place to act: it is fast changing and often open to everybody, but typically unclear in its mechanisms and unstable.

9.1.11 Product Line

The Product Line model is used to design a family of products, all sharing some common features. In a product line, the requirements of the individual products are expressed in terms of features shared by all members of the line (*commonalities*) and distinct features of the member (*variabilities*): the identification of variability is a fundamental step of product line development. The commonality and variability analysis requires significant expertise and knowledge of the domain of the product (*domain engineering*).

Using product lines may make the design more complex, since it must be applied also to the domain of the products, for commonalities. On the other hand, higher reliability is ensured for any instance of the product line, and the time to market to produce indivudal products is lower.

9.1.12 Reuse

Most projects reuse components that can be open/closed source, free or not free. Reusing parts already developed gives advantages: the product is immediately available, and often a higher-quality can be guaranteed at lower costs. The disadvantages of reuse is that who reuses often does not own the software, and then has less control over it. This can be acceptable when reusing platforms, but not when reusing applications.

Reuse heavily impacts the process, because in the requirement phase it must be considered what is already available (in existing components) and the requirements have to be changed accordingly. The components should be evaluated and selected carefully in the design phase, considering possible existing constraints and issues (also commercial and business ones) to which the design of the application must be adapted. Trade-offs have always to be considered while selecting components to reuse.

In general, the activities bound to reuse are mainly four:

- Search and analysis of existing components that can (at least partly) cover the requirements;
- Adaptation of the requirements, accepting trade-offs between the original requirements and available components; some functionalities present in the requirements may be discarded in this phase;
- Design, including new components and reused ones;
- Implementation, including integration of components of different nature.

	No Reuse	**Component 1**	**Component 2**
R1	Invoice in Pdf, Png, Jpg	Invoice in Png, Jpg	Invoice in Pdf, Png
Cost	50	10	12
Time	3 months	1 month	1 month

Table 31 - Reuse trade-offs

9.1.13 Maintenance Process

Changes are the key of a Maintenance process. A request for change in software can be originated by both end users or developers of the software, and is received and processed by a group of maintainers. There are three big ways to maintain software:

- **Corrective Maintenance:** changes performed on software to fix existing defects;
- **Perfective Maintenance:** changes performed on software to modify existing functions or characteristics, to improve them;
- **Evolutive Maintenance:** changes performed on software to introduce new functions on characteristics.

The product evolution is typically subdivided in a flow of changes, with regular releases. Major releases are planned, and usually are delivered every few months; minor (or critical) releases are delivered when needed. In general, the product evolution is done in two or more branches of development, one for fixing defects, and at least another one for evolutive maintenance. There are often two baselines of the product: a stable baseline (for releases of the project) and a working baseline, where maintainers work.

The process of maintenance (both for evolutive and corrective maintenance) is made by a set of steps:

- **Reception** of Change Requests (CR), by users or developers.
- **Filtering** of received CRs. Similar CRs (or different CRs requiring the same modifications) are merged. Unfeasible or incorrect CRs are discarded.
- **Assessment** of the CRs. The impact of each CR is evaluated (in terms of effort and cost required, feasibility, and impact on the architecture/design and functionality of the system). CRs are then ranked in terms of their severity (for corrective ones) and importance (for evolutive ones).
- **Assignation** of the CRs to maintainers, in the ranked order defined in the previous phase. The assignation may be performed by maintainers in autonomy (i.e., they pick the highest ranked CRs from a list), by a project manager, or by a board including product architects, market analysts and quality responsibles.
- **Implementation** of the CRs: the maintainer performs design of the required modification, the required development, and then performs unit and integration test for the modified/new functionalities. A quality group then performs a system test on the whole project. Implemented CRs are inserted in the next release of the project.

The main issue about the maintenance process is that the modifications required for a product can change it radically over time (especially if the product is operated for years). The architecture of the system must be preserved from erosion, and the sustainability of the modifications for users/market must be peroperly verified. Those are the main reasons why often only part of the CRs are implemented, even of the corrective ones received.

9.2 Agile Methodologies

Agile methodologies focus more on code and implementation, and less in documents. The goal of Agile Methodologies is to satisfy the client (not only with respect to the contract) giving early and continuous delivery of valuable software, and giving methodologies able to reduce the cost and the time of software development, increasing the quality.

The Agile methodologies are based on the **Agile Manifesto** (*agilemanifesto.org,* by Kent Beck, Robert C. Martin, Martin Fowler) whose main points are:

- Individuals and interactions over processes and tools;
- Working software over comprehensive documentation;
- Customer collaboration over contract negotiation;
- Responding to change over following a plan.

From those principles, it is evident that the highest priority is given to the customer, welcoming changes in the requirements even late in development, and delivering woring software frequently (from a couple of weeks to months, preferring shorter timescales). Furthermore, the Agile principles suggest building projects around motivated individuals, giving them the needed support and an environment with the needed characteristics. These individuals should comunicate quite always face to face, incrementing the collaboration and communication inside the time. Agile suggests being as simple as possible, being clearer in code and documentation, in projecting and modeling: the result of such simplicity is a project that is more readable and easier to modify when needed. The full agile principles are shown in table 32.

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
Business people and developers must work together daily throughout the project.
Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
Working software is the primary measure of progress.
Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
Continuous attention to technical excellence and good design enhances agility.
Simplicity--the art of maximizing the amount of work not done--is essential.
The best architectures, requirements, and designs emerge from self-organizing teams.
At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Table 32 – The Agile Manifesto

9.2.1 Scrum

The Scrum process starts from a *Product Backlog*, that is a collection of ordered requirements (features) that should be implemented in the final production. Scrum is based on iterative *sprints*: they are short iterations, that aim at implementing a subset of the functionalities collected in the sprint baclog. From the sprint backlogs, the requirements that can be implemented in one month are selected. Sprints have a fixed duration of exactly one month, and are not flexible. An *increment* is what is finally done in every sprint: it must be an usable product.

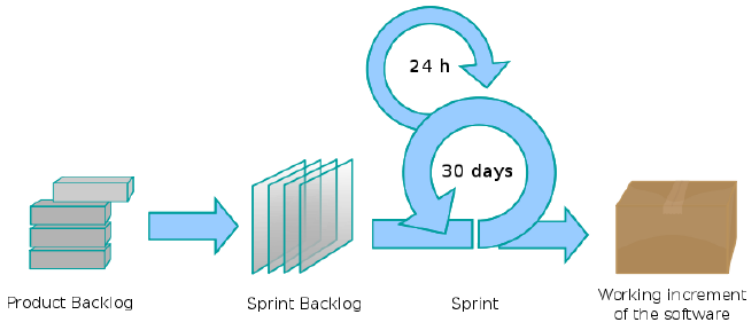

Figure 152 - The Scrum Process

In addition to the sprints, the scrum technique defines some fixed, required meetings. Every sprint must start with a planning meeting, lasting one day (*the sprint planning meeting*). Then, there must be a 15-minute meeting every morning (*daily scrum*) in which the participants stand up (according to a psychology technique to avoid long meetings). At the end of each sprint there is a *sprint review meeting*, in which the demo of the product is presented to the customer; this meeting should last at most 4 hours. Also, a *sprint retrospective*, post-mortem meeting is planned at the end of the project.

Scrum also defines the roles in the team: there must be a *scrum master* (the team leader, synchronizing the meetings), a *product owner* (a person representing the stakeholders and business department, it can be the owner or a delegate), and the members of the *development team* (the self-organized team performing all the required activities in the sprint).

The product backlog needs to be ranked to do the right pick in choosing the sprint backlog (i.e., the functionalities to be implemented in an individual sprint). The product owner has the task of ranking properly the requirements. The picks from the product backlog are done by the scrum master, who has to choose – considering their ranking- a subset of features that can be implemented in the time frame of a sprint.

9.2.2 Extreme Programming (XP)

Extreme Programming (Kent Beck, *Extreme Programming Explained,* Addison-Wesley, 2000) is not a defined process model, but instead a collection of good habits in producing software. Being part of the Agile techniques, Extreme programming is focused on coding within each phase of the software development life cyle. Beyond the agile principles, which are all adopted, the XP programming key elements are pair programming, the systematic use of automated unit test (written before coding) and refactoring, simplicity both in code and in the sign and very fast deliveries, a clear distinction between decisions made by business stakeholders and developers.

The basic fact on which the XP practices are based is that only producing code is actually required to deliver a system, whereas analysis and designe (even if of high quality) are wasted if, at the very end, the system is not finished or never used. This focus on code writing has often caused controversies around XP programming, seen as *reckless coding*: critics of XP point out that it has no clear structure and deliverables and hence cannot create realistic estimates about the projects; it also lacks up-front detailed design and development of infrastructure; finally, in XP there are no specialists: all the programmers participates in all the stages of software development.

Extreme Programming identifies four main values:

- **Communication:** *"Problems with projects can invariantly be traced to somebody not taling to somebody else about something important"*. XP promotes face-to-face comunications, keeping all the staff in the same place;
- **Simplicity**: *"What is the simplest thing that could possibly work?"* XP encourages projects that can be completed rapidly, and in an easy and readable way.
- **Feedbac**k: *"Have you written a test case for that yet?"* XP is focused on delivering as soon as possible a valuable software, with an extensive usage of test cases.
- **Courage:** *"Big jumps take courage"*. XP focuses on flexible plans, and welcomes changes.

Extreme Programming defines thirteen rules (or practices), that are detailed in the following. The twelve rules fit all together, creating a mechanism able to produce good quality code in very short time.

Customer satisfaction	On-site Customer
	Small releases
Software quality	Metaphor
	Test driven development
	Simple design
	Refactoring
	Pair Programing
Project Management	Planning game
	Sustainable development
	Collective code ownership
	Continuous integration
	Coding standards
Environment	Open space, colocated staff, coffee machine

Table 33 - Extreme Programing rules

1 **On site customer:** one of the requirements of XP is to have the customer available, as a part of the development team. Many software projects, in fact, fail because they do not deliver software that meets business needs. The real customer must be in the team, defining business needs through *user stories*, answering questions and prioritizing features. The idea behind the principle is to fit the customer needs, not the developers' ones.

2 **Small releases**: the objective is to put the system into production as soon as possible, in order to have fast feedback. Having a short cycle time allows to deliver valuable features first, and it is easier to plan one or two months instead of 6-12. The longer it is waited to introduce an important feature in the system, the less time will be available to fix it.

3 **System metaphor**: an important quality for Extreme Programming is being able to explain the system design without huge documents. The system metaphor explains, in few lines, how the

whole system work and what is the overall idea o the system. It is used to clarify rapidly the idea of the project, and every change must be done respecting it.

4 **Simple design**: the aim of Extreme Programming is to do always the simplest thing that could possibly work next, and to not design having already in mind extensions for the future. This way, the design will remain clean of things that may be useless or never implemented. A "right" design for XP is a design with no code duplication, with fewest possible classes and methods, while still fulfilling all current business requirements and running all tests without failures.

5 **Refactoring**: the aim of refactoring is to restructure often the system, without changing its functionalities. The goal is to keep the design simple and then change bad design when found, along with removing dead code. Everything that is currently unused must be cleaned in Extreme Programming: it is more likely that it will never be used anymore.

6 **Pair Programming**: All productin code is written by two people looking at the same workstation, one at the eyboard (implementing methods) and one supervising and thinking strategically about potential improvements, test cases, issues. The members of the pairs swap all the time, and they should have the same experience and competences. It is proved that this approach gives many advantages: there is not a single expert of any part of the system, the code is often better due to the inspection made by always at least a member of the pair.

Pair Programming has been questioned because it has been unclear if it wasted developed time and reduced productivity. In a study from Williams et al. (*strenghtening the Case for Pair-Programming*, IEEE software, july/aug 2000) it has been proved that, in an university study with 41 students, pair programming produced higher quality code (in terms of test cases passed), that was developed in less time (40-50% faster with respect to solo programming). In addition to that, pair programming proved to be more enjoyable for developers (preferred by 85% of the participants).

Another study (Dyba et al.) highlights that expertise of developers must be taken into account when deciding to use Pair Programming or not: it is best to use PP for junior experienced developers, but not for senior developers involved in easy tasks.

Programmer expertise	Task Complexity	Use PP?
Junior	Easy	Yes, if increased quality is the main goal
	Complex	Yes, if increased quality is the main goal
Intermediate	Easy	No
	Complex	Yes, if increased quality is the main goal
Senior	Easy	No
	Complex	No, unless task too complex for one developer

Table 34 - When to use Pair Programming (Dyba et al.)

7 **Test Driven Development:** write automated test cases *before* production code. Developers must write unit tests, and users must provide feature and acceptance test. The code must be developed in order to pass those tests. Strong emphasis is posed on regression testing: the unit tests must be executed all time, in each release and in the entirety of test suites. 100% of test suites must be passed.

8 **The planning game:** it is a meeting that occurs once per iteration, often one time per week. There, business decisions (i.e., which stories should be developed and should be defined a priority, along with their release dates) and technical decisions (time estimated for each feature, consequences of business decisions and related team organization and scheduling) are discussed. This meeting reflects what is done in the sprint planning meeting, in the scrum technique.

9 **Sustainable development**: working overtime makes developers tired and demoralized, and less work will be done, not more. A sustainable pace must be maintained during extreme programming: developing full speed only works for very fresh people, and in general developers should not work more than 40 hours per week.

10 **Collective ownership:** any developer can change any line of code, and add any functionality, fix bugs or refactor other developers' code. There is no code owner: individual code ownership tends to create experts. In Extreme Programming, everybody is required when improvements are necessary.

11 **Continuous Integration:** integration happens every few hours of development: the code is released onto an integration machine, in the current baseline. All test cases are run, and in case of errors, the system is reversed to the old version, problems are fixed, and a new integration is tried.

12 **Coding standards:** the team, in Extreme Programming, should adopt a coding standard, in order to make it easier for other people to understand their code. This also avoids code changes because of syntactic preferences by individual developers.

13 **Environment:** the environment in which the developers work should support communications between them. Open spaces should be preferred to closed offices, and common moments and spaces (e.g., coffee machines, shared lunchbreaks, whiteboards) should be encouraged.

Issues can arise when XP is adopted, especially for adopting all of thirteen techniques at once (as suggested by the proposer of the methodology). What is more realistic is a stepwise adoption of Extreme Programming, starting from the recognition of the worst problem in the current way of developing, and the application of the corresponding XP technique.

Another obstacle to the adoption of Extreme Programming may be the necessity of co-locating all the team members: this makes the process scalable for small teams involved in small projects. In general, XP is applicable to teams of up to 10 developers (in some cases, up to 20), where it is easy for the customer representatives to reach the developers on sites, and where it is easy to set up a testing procedure that can be replicated easily at each development iteration.

9.3 Process selection

Given the set of all available process models, which is the best one for a specific product to develop? The selection of a process to follow depends on the specific situation and needs, and an answer valid in every case does not exist.

Figure 153 - Process selection

Every process can be characterized by a set of attributes, that are detailed in the following:

- **Sequential or parallel**: the activities of the process can be performed in parallel or must be done one after another (i.e., documents can be modified in parallel).
- **Iteration or no iteration:** does the process model considers a set of activities that must be repeated? If so, how long does an iteration last?
- **Time framed or not:** does the process model has explicit time boundaries for the activities?
- **Colocation of staff:** is the workforce expected to work al in the same place?

- **Emphasis on documents:** does the process model considers mandatory documents to be delivered at the end of each of its phases?

	Agile	26262	RUP	Synch and stabilize
Iteration number	Many	One+	Few	Few
Iteration duration	Wees	Long	Months	Months
Parallel activities	Yes	No	Yes	Yes
Documents	Few	Many	Many	Few
Colocation staff	Yes	Yes/No	Yes/No	Yes

Table 35 - Comparison between processes

Those process attributes must match the product attributes, that are the following:

- **Criticality** (safety critical, mission critical, others): it depends on the domain and on the objective of the project. If the project is critical, more emphasis is on reliability, safety and security, and norms and laws of the domain apply to the process. Some process models are developed to address safety issues of specific domains: for instance, Iso 26262 is adopted for safety critical functions in the automotive domain.
- **Size** (in terms of LOCs, duration of development, team size, and number of subcontractors of the project): it has effect on the possibility of coordinating the development and maintenance, during the execution of the project. Some process models, especially agile ones, cannot be adopted for very big-sized projects.
- **Domain:** it is strongly related to the criticality of projects. Several domains (e.g., aerospace, medical, automotive, industry, banking, insurance) need the application of norms and laws to the final product and also to the development process. The developers, operators and maintainers can have higher responsibilities with respect to others working in different domains.
- **Lifecycle:** the expected lifecycle (in months or years) of the project imposes tradeoffs between the development costs and maintenance costs: is it better to spend a lot of time on development and save money on maintenance or viceversa? Furthermore, who pays for maintenance?
- **Bespoke / Market driven:** the product may be developed only for one customer or end user (if so, it is called bespoke) or to many customers or end users (market driven). The type of product has effects on the time to market, on the requirements elicitation phase (it takes more time to collect and rank properly requirements for a market driven product), on the cost structure and on the business model to consider.
- **Ownership:** who owns the software, and how is it distributed to its users? The ownership model has effects on the capability of modifying code when the product is released, and on the possibility of adapting the code to new requirements in addition to the original ones.
- **Relationships with the developer:** the relationships with the developers must be taken care of, as it is done with the relationships with the end users. The process model to adopt may vary according to the location of developers, that may be part of an internal department of the company, or work for an external company: the maintenance process may be conducted in different ways in the two scenarios.

The process selection activity should map carefully the product attributes and the process attributes, understand them and rank the product needs properly. Some rules of thumb exist, that can be followed in specific situations, and that are detailed in table 36.

Characteristics of product	Type of process
Reliability, safety	Waterfall-like model (document-based, few iterations, no parallel activities)
Time to market (Market driven project)	Models with frequent iterations
Colocation of staff	No document-based, no parallel activities
Big size	Document-based, many activities
Long lifetime	Document-based

Table 36 - Rules of thumb for process selection