Software Engineering notes

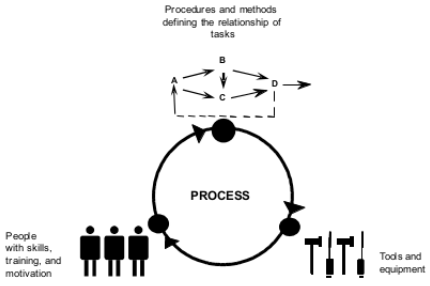
Matteo Salvino

v0.2 - July 22, 2020

Contents

1 Capability Maturity Model Integration

Nowadays companies want to deliver products and services better, faster and cheaper. At the same time, in the high-technology environment of the twenty-first century, nearly all organizations have found themselves building increasingly complex products and services. It's unusual today for a single organization to develop all the components that compose a complex product or service. More commonly, some components are built in-house and some are acquired; then all the components are integrated into the final product or service. Organizations must be able to manage and control this complex development and maintenance process. The problems these organizations address today involve enterprise-wide solutions that require an integrated approach. Effective management of organizational assets is critical to business success. In other words, these organizations are product and service developers that need a way to manage their development activities as part of achieving their business objectives. In the current marketplace, maturity models, standards, methodologies, and guidelines exist that can help an organization to improve the way it does business. However, most available improvement approaches focus on a specific part of the business and do not take a systematic approach to the problems that most organizations are facing. So, focusing on improving one particular business area, these models are hindered by barriers that exist in organizations. **CMMI** for development (CMMI-DEV) provides an opportunity to avoid or eliminate these barriers. It consists of best practices that address development activities applied to products and services. It addresses practices that cover the product's lifecycle from conception through delivery and maintenance. The goal is to build and maintain the total product. CMMI-DEV contains 22 process areas. Of those process areas, 16 are core process areas, 1 is a shared process area and 5 are development specific process areas. The Software Engineering Institute (SEI) in its research, has found some key points on which organizations can focus on improving their business.



In particular, these key points are: people, procedures and methods, and tools and equipment. These points are kept together by the processes used in our organization. They allow us to address scalability and provide a way to incorporate knowledge of how to do things better. Processes allow us to leverage our resources and to examine business trends. We are focusing on methodologies, not because people and technologies aren't important, but since the technology changes at an incredible speed and people can works for different companies, in this way we can provide the infrastructure and stability necessary to deal with an ever-changing world and to maximize the productivity of people and the use of technology to be competitive. The SEI has taken the process management premise "the quality of a system of a product is highly influenced by the quality of the process used to develop and maintain it" and defined CMMs that embody this premise. A CMM, including CMMI, is a simplified representation of the world, which contains the essential elements of effective processes. CMMs focus on improving processes in an organization, describing an evolutionary path from an immature process to a disciplined mature process with improved quality and effectiveness. CMMI can be used in process improvements also as a framework, which provides the structure needed to produce CMMI models, training, and evaluation components. To allow the use of multiple models withing the CMMI framework, model components are classified as either common to all CMMI models or applicable to a specific model. The common material is called **CMMI Model Foundation** (CMF). The components of the CMF are part of every model generated from the CMMI framework. Those components are combined with material applicable to an area of interest to produce a model. A **constellation** is defined as a collection of CMMI components that are used to construct models, training materials, and evaluate related documents for an area of interest. CMMI-DEV previously introduced is the development constellation for a model. All CMMI models contain multiple **Process Areas** (PAs). A process area is a cluster of related practices in an area that, when implemented collectively, satisfies a set of goals considered important from making improvements in that area. Some of these areas are: Causal Analysis and Resolution (CAR), Configuration Management (CM), etc. We can define two types of goal:

- **Generic**: it's called generic because the same goal statement applies to multiple process areas. A generic goal describes the characteristics that must be present to institutionalize processes that implement a process area. It's a required model component and is used in the evaluation to determine whether a process area is satisfied or not.

Institutionalization is an important concept in process improvement since it implies that the process is ingrained in the way the work is performed and there is a commitment and consistency to perform the process. The progress of process institutionalization is characterized by the type of generic goal:

- **GG1 Performed process**: a performed process is a process that accomplishes the work necessary to satisfy the specific goals of a process area.
- **GG2 Managed process**: a managed process is a performed process that is planned and executed in accordance with some policy. It employs skilled people having adequate resources to produce controlled outputs. It's controlled, monitored, reviewed, and evaluated for adherence to its process description. The process can be instantiated by a project, group, or organizational function. Management of the process is concerned with institutionalization and the achievement of other specific objectives established for the process, such as cost, schedule, and quality objectives. The control provided by a managed process helps to ensure that the established process is retained during times of stress. The requirements and objectives for the process are established by the organization. The status of the work products and services are visible to management at defined points. Commitments are established among those who perform the work and the relevant stakeholders and are revised as necessary. Work products are reviewed with relevant stakeholders and are controlled. A critical distinction between a performed process and a managed process is the extent to which the process is managed. A managed process is planned and its execution is managed against the plan. Corrective actions are taken when the actual results and execution deviate significantly from the plan. A managed process achieves the objectives of the plan and is institutionalized for consistent execution.
- **GG3 Defined process**: a defined process is a managed process that is tailored from the organization's set of standard processes according to the organization's tailoring guidelines. It has a maintained process description and contributes process-related experiences to the organizational process assets. Organizational process assets are artifacts that relate to describing, implementing, and improving processes. These artifacts are assets because

they are developed or acquired to meet the business objectives of the organization and they represent investments by the organization expected to provide current and future business value. The organization's set of standard processes, which are the basis of the defined process, are established and improved over time. Standard processes describe the fundamental process elements that are expected in the defined processes. Standard processes also describe the relationships among these process elements. A project's defined process provides a basis for planning, performing, and improving the project's task and activities. A critical distinction between a managed process and a defined process is the scope of application of the process descriptions, standards, and procedures. For a managed process, the process descriptions, standards, and procedures are applicable to a particular project, group, or organizational function. As a result, the managed processes of two projects in one organization can be different. Another critical distinction is that a defined process is described in more detail and is performed more rigorously than a managed process. Finally, management of the defined process is based on the additional insight provided by an understanding of the interrelationships of the process activities and detailed measures of the process, its work products, and its services.

- **Specific**: a specific goal describes the unique characteristics that must be present to satisfy the process area. It's a required model component and is used in the evaluation to determine whether a process area is satisfied or not.

1.1 Understanding levels

Levels are used in CMMI-DEV to describe an evolutionary path recommended for an organization that wants to improve the processes it uses to develop products or services. CMMI supports two improvement paths using levels. One path enables organizations to incrementally improve processes corresponding to an individual process area selected by the organization. The other path enables organizations to improve a set of related processes by incrementally addressing successive sets of process areas. These two improvement paths are associated with the two types of levels: **capability levels** and **maturity levels**. These levels correspond to two approaches to process improvement called **representations**. In turn, the two represen-

tations are called **continuous** and **staged**. Both representations provide ways to improve our processes to achieve business objectives and use the same model components. The continuous representation is concerned with selecting both a particular process area to improve and the desired capability level for that process area. The staged representation is concerned with selecting multiple process areas to improve within a maturity level. Both capability levels and maturity levels provide a way to improve the processes of an organization and measure how well organizations can and do improve their processes. However, the associated approach to process improvement is different.

1.1.1 Capability levels (Continuous representation)

To support those who use the continuous representation, all CMMI models reflect capability levels in their design and content. The four capability levels, each of them is a layer for the process improvement, are designated by the numbers 0 through 3:

- **0. Incomplete**: an incomplete process is a process that either is not performed or is partially performed. One or more of the specific goals of the process area are not satisfied and no generic goals exist for this level since there is no reason to institutionalize a partially performed process.

- **1. Performed**: a performed process is a process that accomplishes the needed work to produce work products; the specific goals of the process area are satisfied.

- **2. Managed**: a managed process is a performed process that is planned and executed in accordance with some policy; it employs skilled people having the adequate resources to produce controlled outputs; it involves relevant stakeholders; it's monitored, controlled, reviewed, and evaluated for adherence to its process description. The process discipline reflected by capability level 2 helps to ensure that existing practices are retained during times of stress.

- **3. Defined**: a defined process is a managed process that is tailored from the organization's set of standard processes according to the organization's tailoring guidelines and contributes work products, measures, and other process improvement information to the organizational process assets.

1.1.2 Maturity levels (Staged representation)

To support those who use the staged representation, all CMMI models reflect maturity levels in their design and content. A maturity level is a defined evolutionary plateau for organizational process improvement. Each maturity level matures an important subset of the organization's processes, preparing it to move to the next maturity level. The maturity levels are measured by the achievement of the specific and generic goals associated with each predefined set of process areas. The five maturity levels are designated by the numbers 1 through 5:

- **1. Initial**: in this layer processes are usually ad hoc and chaotic. In spite of this chaos, maturity level-1 organizations often produce products and services that work, but they frequently exceed the budget and schedule documented in their plans. These organizations are characterized by a tendency to overcommit, abandon their processes in a time of crisis, and be unable to repeat their successes.
- **2. Managed**: as before.
- **3. Defined**: as before.
- **4. Quantitatively managed**: it's a defined process that is controlled using statistical and other quantitative techniques. Quantitative objectives for quality and process performance are established and used as criteria in managing the process. Quality and process performance is understood in statistical terms and is managed throughout the life of the process.
- **5. Optimizing**: it's a quantitatively managed process that is improved based on the understanding of the common causes of variation inherent in the process. The focus of an optimizing process is on continually improving the range of process performance through both incremental and innovative improvements.

1.2 ISO family

ISO 12207 defines and structures all activities involved in the software development process. Its main goal is to provide a common language to involved stakeholders. It's based on a functional approach: a set of coordinated activities transforming an input in an output. It's based on two basic principles:

- **Modularity**: it means processes with minimum coupling and maximum cohesion.
- **Responsibility**: it means to establish responsibility for each process, to facilitate the application of the standards in a project where there are many people involved.

ISO 9000 is maintained by ISO and is administered by accreditation and certification bodies. This family of ISO addresses "Quality management". Its fundamental building blocks are:

- **Quality management system**: it deals with general and documentation requirements that are the foundation of the management system. In particular, the previous requirements can be explained in details:
  - **General**: they are general requirements like how the processes of the management system interact each other or how you will measure and monitor the processes.
  - **For documentation**: they are requirements focused on the documentation. They can require what documentation is needed to operate the system effectively or how it should be controlled.
- **Management responsibility**: it manages high-level responsibilities like set policies and objectives or plans on the objectives.
- **Resource management**: it deals with the people and physical resources needed to carry out the process. People should be competent to carry out their task and physical resources and work environments need to be capable of ensuring that the customer's requirements are satisfied.
- **Product-service realization**: it deals with the processes necessary to produce the product or to provide the service.
- **Measurement, analysis, and improvement**: it deals with measurements to enable the system to be monitored. For example, we can measure if the processes are effective or if the product satisfies the customer's requirements.

ISO doesn't itself certify organizations. There are accreditation bodies that authorize certification bodies. Organizations can apply for ISO 9001 compliance certification to a certification body. The various accreditation bodies

have mutual agreements with each other to ensure that certificates issued by one of the Accredited Certification Bodies (CB) are accepted worldwide. An ISO certificate is not a once-and-for-all award, but must be renewed at regular intervals recommended by the certification body, usually around three years. **Quality requirements** are a set of process requirements and resources that constitute the Quality Manual (QM) of the organization. This latter specifies the organization's quality policy regardless of the specific commitments and customers. It's adapted to specific projects, generating several Quality Policies (QP). The ISO 9001 certification required that processes are described in the two previous specific documents (QM and QP).

2 Software Development Process Models

Software products are not tangible. To manage a software project, the project manager needs special methods. So, the monitoring process is based on the explicit definition of activities to be performed and documents to be produced. These documents allow us to monitor the progress of the process and give us an idea of its quality. Software Development Process Models and their instances, differ from each other for the required activities and the produced documents. Now, we will see several software process models, discussing their pros and cons.

2.1 Waterfall model

A Waterfall model is a typical process in which there are separate and distinct phases of specification and development. So, first, we describe all the specifications of the software and then we have a subsequent phase that takes care about development. Its main phases are:

- **Requirements analysis and definition**
- **System and software design**
- **Implementation and unit testing**
- **Integration and system testing**
- **Operation and maintenance**.

Each phase depends on the deliverables of the previous one and corresponds to a specialization of tasks. One of its main drawbacks is the difficulty of allows changes after the process is started. Another drawback is that

the users don't have a vision of the overall system (uncertainty). So, they have to wait until a working version of the system is available. In turn, programmers have to wait for the analysis phase before starting their job. The solution is to use an iterative approach.

2.2 Process iteration

The system requirements always evolve during the project development, so process iteration where earlier stages are reworked is always part of the process for large systems. Iteration can be applied to any of the generic process models. Typically, an iteration follows one of the following approaches:

- **Incremental delivery**: we start building a small system (prototype) and next we enlarge it (incremental way). A **prototypal model** is constituted by a customer interaction to obtain customer's requirements, then we build a prototype also called **mock-up** (when we build a system with feels and looks similar to the final system, but without working software behind), and then we present it to the user, in order that he can test the prototype and say "Ok, I'm satisfied", or vice versa. If the answer is positive, then we can start to implement the functionality behind the mockup. The **incremental model** is formed by iterations which are constituted by analysis, design, implementation, and test phases. The result of a generic iteration i typically is the system with version i. It's very similar to the prototypal model but in this case the intermediate version is full working and it allows for a more accurate design. Its main drawback is that if we figure out a wrong functionality in a specific system version, we have to throw away a lot of jobs previously done, whereas in the first model we will build only the mock-up system. In the incremental development, we define the requirements, then we assign each of them to a specific system release version, design the whole system architecture, then develop, validate and integrate the system increment, and finally validate the system. If this isn't the last iteration, then we continue to increment the functionality of our system; otherwise, we have built the final system. In other words, delivering part of the required functionality. Then the user requirements are prioritized and the highest priority requirements are included in early increments (early versions). Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve. The trade-off is the length of these iterations, the bigger the interval is then we move towards the waterfall model, the smaller the interval is then

will be more difficult to build a very good software architecture. Its advantages are the following: the customer can see at each iteration an increment of the system functionalities, early increments act as a prototype to help to ask requirement for future iterations, lower risk of overall project failure, and the highest priority system requirements tends to receive the most testing.

- **Spiral development**: in spiral development, the process is represented as a spiral (rather than a sequence) of activities. Each loop in the spiral represents a phase in the process. There not exist fixed phases such as specification or design (the loops in the spiral are chosen depending on what is required). The risks are explicitly evaluated and resolved throughout the process. In particular, the spiral model is constituted by the following sectors:
  - **Objective setting**: they are specific objectives for the current phase.
  - **Risk assessment and reduction**: the risks are evaluated and activities punt in place to reduce the key risks.
  - **Development and validation**: in this sector, we choose a development model for the system.
  - **Planning**: the project is reviewed and the next phase of the spiral is planned.

2.3 Formal methods

Formal methods are formalisms based on logic or algebra for requirement specification, development, and test. They don't use the natural language, because it's very ambiguous, but tends to write the specification of the software in some formal languages like Z, Z++, etc.

2.4 Extreme programming

It's a part of the Agile model family. It's an approach based on the development and delivery of very small increments of functionality. It relies on constant code improvement, uses involvement in the development team, and pairwise programming.

2.5 Core process activities

Now, lets quickly review the main activities involved in the development process:

- **Software specification**: it's the process of establishing what services are required and the constraints on the system operations and development. These requirements can be functional or non-functional (they regard system quality). This process is also called the requirements engineering process. First of all, it performs a feasibility study to understand if building the system is feasible. Typically this phase produces a document called feasibility report. Subsequently, the requirements are defined, analyzed, and validated. The results of these phases constitute the so-called requirement document.

- **Software design and implementation**: it's the process of converting the system specification into an executable system. The software design phase is a process in which we design the software structure (architectural, component, data structures, etc.) that fulfill the specifications. In the implementation phase, we translate the previous structure into an executable program. In this phase, our goal is also to remove as many errors as possible from the program generated, typically using a program testing. These two phases are closely related and may be interleaved.

- **Software validation**: the verification and validation are intended to show that the system is conform to its specifications (verification) and meets the customer's requirements (validation). Typically the system is tested over test cases that are derived from the specification of the real data to be processed by the system. We can use several types of testing:

  - **Unit test**: individual components are tested independently.
  - **System test**: test the whole system.
  - **Acceptance test**: testing with customer data to check that the system meets the customer's needs.

- **Software evolution**: software is very flexible and can change over time. A change of the requirements must be reflected also in the software.

3 Scrum

Scrum is an agile process that allows us to focus on delivering the highest business value in the shortest time. It allows us to rapidly and repeatedly inspect actual working software (every two weeks to one month). The business sets the priorities. Team self-organize to determine the best way to deliver the highest priority features. Every two weeks to a month anyone can see real working software and decide to release it as is or continue to enhance it for another sprint. The requirements of the customer are captured as items in a list of "product backlog". The core values of Scrum are the following:

- **Commitment**: teams commit to their goals for the sprint, product owners commit to ordering the product backlog and ScrumMasters commit to removing any obstacles along the way to simplify the flow of product development. The Scrum team should do whatever is necessary to meet their goals, and it's crucial that they are empowered to do so.

- **Focus**: for a team to be able to complete its work, its members must be allowed to focus. The ScrumMaster doesn't allow changes in the sprint's commitment so that the team may keep its focus. When a team gives its full attention to the problem, its work is much more productive, predictable, and fulfilling.

- **Openness**: as Scrum uses empirical process control to make progress through a project, it's essential that the results and experience of an experiment (i.e. a sprint) are visible. Once visibility exists, inspection and adaption can occur.

- **Respect**: to be its best, a team's members need to respect for each other and, the knowledge that each brings to the table, experiences, working styles, and personalities. Respect doesn't come for free, it's earned. Scrum team members should be dedicated, cross-functional, empowered, and self-organizing.

- **Courage**: it takes buckets of courage for a ScrumMaster to apply Scrum the way it was intended. One of the primary responsibilities of the ScrumMaster is to help the organization identify its weaknesses so that it may improve. This takes courage. Sometimes, a team has to push back on the product owner when asked to take on too much during a sprint. It takes courage to say no to that sort of pressure.

A product owner must have courage when communicating with other stakeholders about the reality of a project.

3.1 Roles

Scrum team The Scrum team includes the product owner, Scrum Master, and the team members, whereas the Scrum delivery team is a subset made of only the technical team members. The whole Scrum team huddles around a problem (i.e. a requirement from the Product Backlog) and innovates solutions. Scrum teams should be five to nine team members, dedicated to the life of the project, cross-functional, empowered, and self-organizing. Scrum teams plan, estimate and commit to their work, rather than a manager performing these activities for them. The end goal of the team is to deliver a potentially shippable product increment that meets an agreed-upon Definition of Done every sprint.

Product owner The product owner is responsible for the product's success. In other words, while the team is responsible for delivering a quality solution, the product owner is responsible for knowing his market and user needs well enough to guide the team towards a marketable release sprint after sprint. In a project, there should be one and only one product owner who makes final decisions about the direction of the product and the order in which features should be developed. The product owner, since he represents the "what" and "why" of the system, should be available to the team to have a regular dialog about the requirements in the product backlog; additionally, the product owner must make the product vision clear to everyone on the team and regularly maintain the product backlog in keeping with the product vision. The product owner always keeps the next set of product backlog items in a ready state so that the team always has work in the queue for the next sprint.

ScrumMaster The ScrumMaster safeguards the process. He/she understands the reasons behind and for an empirical process, and does his or her best to keep product development flowing as smoothly as possible. This leader protects team members from interruptions to keep them focused on their sprint commitments. The ScrumMaster also facilitates all Scrum meetings, ensuring that everyone on the team understands the goals and that they share a commitment as a true team and not just as a collection of individuals. She/he removes obstacles that prevent a steady flow of high-value features.

3.2 Ceremonies

A sprint is an iteration defined by a fixed start and end date; it's kicked off by sprint planning and concluded by the sprint review and retrospective. The team meets daily, in a daily scrum meeting, to make their work visible to each other and synchronize based on what they've learned. Let's discuss in detail these phases one at a time.

Sprint planning During sprint planning, the product owner and the team discuss the highest priority items in the product backlog and brainstorm a plan to implement those items. The set of chosen product backlog items and their subsequent tasks collectively is referred to as the team's sprint backlog. The sprint planning meeting is time-boxed to eight hours for a 30-day sprint, reduced proportionally for shorter sprints. The meeting is constituted by two parts: the first one is driven by the product owner who presents the most important product backlog items (with the support of drawings, mockups, etc.) and clarifies question from the development team about what he/she wants and why he/she wants it. The second part is driven by the Scrum delivery team who work together to a brainstorm approach and eventually agree on a plan. It's at the start of this second part that the sprint begins. Of course, teams are always searching for ways to make planning faster and more efficient. The result of sprint planning is a sprint backlog that is comprised of selected product backlog items for the sprint, along with the correlating tasks identified by the team in the second part of sprint planning.

Sprint review The sprint review provides an opportunity for stakeholders to give feedback about the emerging product in a collaborative setting. In this meeting, the team, product owner, ScrumMaster, and any interested stakeholders meet to review and talk about how the product is shaping up, which features may need to change, and perhaps discuss new ideas to add to the product backlog. It's common for a ScrumMaster to summarize the event of the sprint, any major obstacles that the team ran into, and so on and, of course, the team should always demo what they've accomplished by the sprint's end. This meeting is time-boxed to four hours for a 30-day sprint.

Sprint retrospective During the final spring meeting, the sprint retrospective, team members discuss events of the sprint, identify what worked well for them, what didn't work so well, and take on action items for any

changes that they would like to make for the next sprint. The ScrumMaster will take on any actions that the team doesn't feel it can handle. The ScrumMaster reports progress to the team regarding these obstacles in subsequent sprints. This meeting is time-boxed to three hours.

Daily scrum meeting In the daily scrum meeting, team members make their progress visible so that they can inspect and adapt towards meeting their goals. The meeting is held at the same time and in the same place, decided upon by the team. Even though a team makes its best attempt at planning for a sprint, things can change in flight. In this 15-minute meeting, team members discuss what they did since yesterday's meeting, what they plan to do by tomorrow's meeting, and to mention any obstacles that may be in their way. The ScrumMaster record any obstacles that the team members feel they cannot fix for themselves and will attempt to remove them after the meeting. The scrum delivery team members, product owner, and ScrumMaster are participants in the meeting. Anyone else is welcome to attend but only as observers.

3.3 Artifacts

Product backlog The product backlog is the product owner's "wish list". Anything and everything that they think they might want in the product goes into this list. The product owner maintains the product backlog, although other stakeholders should have visibility of and the ability to suggest new items for the list. The product owner prioritizes the product backlog, listing the most important or most valuable items first. Once a team selects items for a sprint, those items, and their priorities are locked; however, priorities and details for any not-started work may change at any time. Through this mechanism, teams can focus on this sprint's work while the product owner retains maximum flexibility in ordering the next sprint's work.

Sprint backlog Owned by the team, the sprint backlog reflects the product backlog items that the team committed to in sprint planning, as well as the subsequent tasks and reminders. Team members update it every day to reflect how many hours remain on his/her task; team members may also remove tasks, add tasks, or change tasks as the sprint is started.

Product increment The product increment is a set of features, user stories, or other deliverables completed by the team in the sprint. It should be potentially shippable (i.e. it must have enough quality to give it to

the users). The product owner is responsible for accepting the product increment during each sprint, according to the agreed-upon Definition of Done and acceptance criteria for each sprint deliverable. Without a product increment, the product owner and other stakeholders have no way to inspect and adapt the product. A team must keep its progress visible at all times. It will create many additional artifacts to ensure visibility. Some common visibility tools are the release and sprint burndown charts. A burndown chart display of what work has been completed and what is left to complete. It is provided for each developer or work item. It's updated every day and makes the best guess about hours/points completed each day. A possible variation of this chart is called release burndown chart, which displays how much work remains in the release backlog at the end of each sprint (it shows overall progress).

4 Layers vs Tiers

Typically, an information system is constituted by three layers: presentation, application logic, and resource management layer. The **presentation layer** is devoted to present the system's user interface to clients. The **application logic** determines what the system does. It takes care of enforcing business processes. The application logic can take many forms: programs, constraints, etc. The **resource manager** deals with the organization (storage, indexing, and retrieval) of the data necessary to support the application logic. This is typically a database but it can also be a text retrieval system or any other data management system providing querying capabilities and persistence. A client is any user or program that wants to perform an operation over the information system. Clients are independent of each other: one could have several presentation layers depending on what each client wants to do. One can take advantage of the computing power at the client machine to have more sophisticated presentation layers. This also saves computer resources at the server machine. It introduces the concept of API, an interface to invoke the system from the outside. It also allows designers to think about federating the systems into a single system. The resource manager only sees one client: the application logic. This improves quite a lot the performance since there aren't client connections/sessions to maintain. A **tier** is a physical level. How can we divide logical layers over physical objects? The first architecture developed is the so-called **1-tier architecture**, in which all layers are running on the same machine (i.e. a mainframe). The second approach is called **2-tier architecture**, in which

we can have one machine (server) that contains the application logic and resource management, and another one (client) that contains the presentation layer. This approach is also called client-server architecture. The evolution of this concept is to introduce microservices, and using a **3-tier architecture**, in which the presentation layer runs on the client machine, the application logic layer on a middle machine and the resource management layer on a dedicated machine. The problem is that now, we have more connections than the previous architectures. Indeed, the middle's developers need to write the code for receiving requests from the client and the server. So, the people realize that needed something more, a specific technology called **middleware**, that could make the development of the code as much simple as if the code is still running in the mainframe (on the same machine). Once you know how to split the software into three layers, you can split each of them into more layers on a different machine, to make load balance. The result of this idea is called **N-tier architecture**.

4.1 Middleware

A middleware can serve the requests in two ways:

- **Synchronous**: traditionally, distributed applications use blocking calls, in which the client sends a request to a service and waits for a response of the service to come back before continuing its work. This type of interaction requires both parties to be online: the client makes the request, the receiver gets the request, processes it, and sends a response to the caller, the client receives the response. Since it synchronizes client and server, this approach has several disadvantages:
  - **connection overhead**: synchronous invocations require to maintain a session between the caller and the receiver. Maintaining a session is expensive and consumes CPU resources. There is also a limit on the number of sessions that can be active at the same time. For this reason, client/server systems often use connection pooling (pooling of connections, with a thread associate at each of them and allocate one of them when needed) to optimize resource utilization. Synchronous interaction requires a context for each call and a context management system for all incoming calls. The context needs to be passed around with each call as it identifies the session, the client, and the nature of the interaction.
  - **higher probability of failures**: if the client or the server for some reason fails, the context is lost and resynchronization is

difficult. Finding out when the failure took place may not be easy.

To the previous problems there are two solutions:

- **Enhanced support**: we can use transactional interaction to enforce exactly once execution semantics and enable more complex interactions with some executions guarantees. Another approach is service replication and load balancing, to prevent the service from becoming unavailable when there is a failure.
- **Asynchronous interaction**: see below.

- **Asynchronous**: using asynchronous interaction, the caller sends a message that gets stored somewhere until the receiver reads it and sends a response. The response is sent in a similar manner as before. This type of interaction can take place in two forms:
  - **non-blocking invocation**: a service invocation but the call returns immediately without waiting for a response, similar to batch jobs.
  - **persistent queue**: the call and the response are persistently stored until they are accessed by the client and the server.

A middleware can be implemented with two approaches:

- **Programming abstraction**: it's an approach intended to hide the low-level details of hardware, network, and distribution. Middleware is primarily a set of programming abstractions developed to facilitate the development of complex distributed systems; to understand a middleware platform one needs to understand its programming model. From this latter can be determined its limitations, general performance, and applicability of a given type of middleware. Furthermore, this model also determines how the platform will evolve.

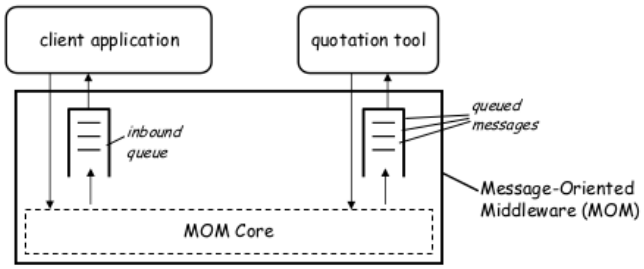
- **Infrastructure**: as the programming abstraction reaches higher and higher levels, also the infrastructure that implements the abstractions must grow accordingly. Additional functionality is always implemented through some additional software layers. The additional software layers increase the size and complexity of the infrastructure necessary to use the new abstractions. The infrastructure is also intended to support additional functionality that makes development, maintenance, and monitoring easier and less costly. It takes care of all

non-functional properties typically ignored by programming models and languages such as performance, maintenance, resource management, etc.

Typically a middleware is based on RPC, in which the client invokes a local procedure talking with its stub (client stub), which can communicate with the remote object via a communication module. A request is created putting the client parameters, serialized (marshal operation) and send to the server stub. The server dispatcher delivers the serialized request to its stub, which deserializes (unmarshal operation) it, extract the parameters, and finally calls the server procedure. When there are more entities interacting with each other, RPC treats the calls as independent of each other. The main drawback of RPC is that recovering from partial system failures may be very complex. So, a programmer instead to implement the whole infrastructure for a distributed application can use an RPC system, which hides distribution behind procedure calls, provides an interface definition language (IDL) to describe the services, generates all the additional code necessary to make a procedure call remote and to deal with all the communication aspects and provides a binder in case it has a distributed name and directory service system. The solution to this limitation is to make RPC calls transactional, i.e. instead of providing plain RPC, the system should provide TRPC. It's based on the same concepts of RPC but, uses additional language constructs and runtime support to bundle several RPC calls into an atomic unit.

4.1.1 Message Oriented Middleware

MOM is software or hardware infrastructure supporting sending and receiving messages between distributed systems, with the following structure:



The participants of the communication are divided into two groups:

- **Publisher**: the entity that sends messages

- **Subscriber**: the entity that expresses interest in certain categories of messages.

The main disadvantage of this approach is that it requires an extra component in the architecture called **event service** that receives messages from the publisher and cross-check them with the interests of subscribers. As with any system, an introduction of another component can lead to performance and reliability reduction, and can make the whole system more difficult and expensive to maintain. The information can be processed in two ways:

- **Topic-based**: the publisher is responsible for defining the topics to which subscribers can subscribe.
- **Content-based**: filters can be used for a more accurate selection of information to be received.

The publisher can specify which topics are available for a subscription (topic-based) and subscribers can specify some particular attributes to filter the messages of interest. The notification can be made in two ways:

- **push**: the subscribers are invoked with a callback, using a reference communicated at the time of subscription.
- **pull**: the subscribers poll the event service when they need messages.

To-Do: add the pseudocode!

5 Web Services

Web services are the current evolution of middleware technology. Basically, they are offered in a way over the Internet. In other words, a web service is a software functionality exposed over the Internet. This means that any piece of code and any application component deployed on a system can be transformed into a network-available service. The main difference between middleware and web services is the Internet because the latter emerged at the beginning of 2000 (when vendors invented new technologies and standards). So, web services perform business functions such as:

- a self-contained business task (for example a funds deposit service)
- the whole business process (for example the automated purchasing of office supplies)

- an entire application (for example demand forecasts and stock replenishment)
- a service-enabled resource (for example access to a particular back-end database containing some interesting data).

Once we have these functionalities exposed over the Internet we can mix and match them to create a complete process. Until the emergence of web service technology the client and server should be on the same web platform (OS or programming language). With web service, we can use whatever programming language and operating system that talks to each other (i.e. platform independent). When we develop a web service we need to take into account the billing model, i.e. the mode in which the customer can pay to use specific functionalities of the service exposed. An **Application Service Providers (ASP)** is based on the idea to rent its applications to subscribers such that:

- the whole application is developed in terms of the user interface, workflow, business, and data components that are all bound together to provide a working solution.
- ASP hosts the entire application and the customer has little opportunity to customize it beyond the appearance of the user interface.
- an alternative of this is where the ASP is providing a software module that is downloaded to the customer's site on demand (situations in which the software can operate remotely via a browser).

When we have an ASP model it offers the whole application suite to the customers, which introduces the concept of software-as-a-service (i.e. pay for the application but we don't own it). This approach has several limitations such as: inability to develop highly interactive applications, inability to provide complete customizable applications, and inability to integrate applications. With Web Services is different because we aren't paying for the whole application, but many different providers offer the functionalities that are needed for someone to develop the software. This is the key difference between ASP and Web Service approach. When we have an ASP, it can complete a full-fledged application, and then the customer can exploit the application viewing it in the browser. When we have Software-as-a-Service, we have an entire application deployed by the service provider, and this software is downloaded on-demand on the customer site and he performs its executed software on his side without accessing a browser. When we have

a web service, the idea is that the ASPs offer basic components and then the customer can compose them to run on his premise or to have another ASP that composes the services offered by third-party. A question to ask ourselves is where are services used? Typically, we have two types of usage:

- **within an enterprise**: web services are intended for being used for enterprise application integration. In this case, they are used to improve the reuse, reduce the skill requirements, can save on infrastructure, deployment and management costs, since we have these components already packed, and can accelerate and reduce the cost of integration as creating a new application is simpler than assembling pieces.

- **between enterprises (e-Business integration)**: web services are used to provide services to the company's customers, allow to access services from the partners and suppliers of a company. In this case, the fact that web service technology is based on standards, allows to accelerate the deployment, allow to reduce the technological barriers, and to set-up common infrastructure, and this dynamicity opens new opportunities.

Typically the types of web services are classified into two main categories:

- **Informational services**: they are those services that are relatively simple from the point of view of the computing that they are provided. They simply provide access to some data with as few modifications as possible through some application or request-response sequence (e.g. weather report or stock quote info).

- **Complex services**: often they include the composition of other services. Indeed, they are services that coordinate a complex application logic and sometimes they involve the invocation of some other external service (e.g. supply-chain application involving order taking, stocking orders, etc).

The properties according to which we classify the services are the following:

- **function and non-functional properties**: functional properties are the description of the operational characteristics that define the behavior of the service (i.e. functionality offered by the service). We need also non-functional properties such as the quality of the service, scalability, availability, etc.

- **Stateless or stateful services**: if we have a stateless service we can invoke it without maintaining the context between the call (e.g. simple informational service). Instead, a stateful service requires its context to be preserved from one invocation to the next (e.g. business processes).

An important aspect when we design service is the **decoupling** and the **granularity** of the service. Decoupling is a way of measuring the degree of dependency that there is among two systems. Typically, web services are loosely coupled, since as they can connect each other through the Internet, we don't need to know too many details of how an application is offered as a service by business partners, because they satisfy standard interfaces, which allows us to know as few details as possible. Instead, granularity is a way of measuring how much functionalities are embedded inside the operation. For example, simple services are quite discrete since they typically exhibit normal request and reply modes of operation and are of fine granularity. Complex services are coarse-grained since they will probably require a lot of interactions to perform a specific task. Typically, this type of service granularity implies larger and richer data structures (supported by XML).

Another way to comparing services is the **synchronicity** and **well-definedness** of the service. Synchronicity means that when we have to design a web service we can follow two programming style: **synchronous** or **RPC-style**, with which the client invoke the service as if the operation is a request with a set of arguments and waits for the return that contains the return value (e.g. simple informational services). The second approach is **asynchronous** or **document-style**, with which the client when invoke the service it will send as parameters an entire document (e.g. a purchase order). The well-definedness means that the service should be able to describe (WSDL) exactly the rules for interfacing and interacting with other applications. As we said service is the natural evolution of object-oriented programming. So, in service-oriented computing is well clear the difference between the **service interface** and **service implementation**. The first one is the definition of the service functionalities visible to the external world and describes how is possible to access them. It's important that in the service interface to have a description of the characteristics of the interface such as which type of information is available, which are the parameters and data types, etc. The service implementation realizes the service interface, hiding to the client how the interface is implemented. Potentially different service providers can use any programming language of their choice to implement the same interface.

In a **service oriented architecture (SOA)** we make a clear distinction between roles:

- **service providers**: they are the organizations that provide the service implementation, supply their service descriptions, and provide related technical and business support behind the service.
- **service clients**: they are the end-user organizations that want to use some service.
- **service registry**: it's a kind of searchable directory, where the service provider can publish the service description and the service clients can look for the service they're interested in.

As we said when dealing with services, is very important to have the definition also of additional non-functional properties that qualify the service. Typically in a web service environment the usual Quality of Service (QoS) measurements are availability, accessibility, how much the service in conformance to standards, integrity, performance, etc. A **Service Level Agreement (SLA)** is a kind of contract between the provider and client that formalize all the details of the web service such as contents, price, delivery process, etc. Typically, an SLA agreement contains different parts such as purpose, parties, validity period, scope, restrictions, penalties, etc. The impact of web service was very high and allowed to converge to a unique set of technologies between business, Enterprise Application Integration, middleware and the web. The main advantages of web service technology are: offers a standard way of exposing legal application as a set of services, represent a standard, easy and flexible way to help overcome application integration issues, allow to develop and assemble applications that are internet-native both for intra-enterprise application and extra-enterprise application, and they are a way to create cross-enterprise systems with also service-level agreements. The first disadvantage is performance because web services are naturally less performant than traditional middleware. In some scenarios, the transaction management should be managed at the application layer. Another bad point is the lack of business semantics, the dynamicity that they promise somehow should be solved outside the technology landscape. Finally, it's important to have widespread agreement and harmonization, but this implies long processes on standardization.

5.1 SOAP

SOAP is an evolution of traditional middleware technology; in fact towards the ends of 90*s* emerged a very interesting way to develop distributed application and in particular modules that were able to communicate over the network according to the RPC interaction paradigm. The background for the evolution of SOAP was **Inter-Application Communication (IAC)**. Conventional distributed applications use distributed communication technologies (e.g. Java/RMI, Corba) based on object RPC protocols that were the idea to connect object orientation and network protocols. The basic idea of RPC based on the object-oriented approach is the fact that you have an identifier used to locate the target object inside the server process. There are some weaknesses such as: both the ends of the communication link need to be implemented under the same distributed object model, and these protocols are very hard to work over firewalls or proxy servers (for example most firewalls are configured to allow HTTP traffic but not application-level protocol like IIOP). So, the idea was that at the beginning of 2000 to define a new protocol called **Simple Object Access Protocol (SOAP)**, that was an XML based communication protocol build on top of HTTP to be able to overcome firewall complexity. SOAP is a messaging protocol that was used by web services. The objective of SOAP is inter-application communication. The idea is that in SOAP the scheme for request and response is coded in XML directly on top of HTTP seen as a transport protocol. We have a service provider, on top of it is running an application object and is equipped with a SOAP-based middleware, which converts procedure calls to/from XML sent through HTTP on the specific programming language and model according to which the servant object is built. On the client-side, we have again an application object, which goal is to invoke the operations of the service. Then we have a SOAP-based middleware, to exchange the messages on top of HTTP. In SOAP we can have two communication mechanisms:

- **RPC**: the client will ask for the execution of something and the servant object will reply providing the information of the computing that is requested. In terms of communication style, the web service offers an operation for example for asking the price of a given product. The program accessing to the database answer back by providing the response and providing the price. In particular, it appears as a remote object to a client application. From the point of SOAP we have a SOAP envelope and inside there is a body, which contains the name of operations and the parameters coded in XML. In this case, the clients express their request as a method call with a set of arguments, which

returns a response containing a return value.

- **document**: we have that the client ask for a long-running operation and therefore instead of executing an action, he's asking for starting a kind of business process and typically sends very complex document. In this case, the body will contain not the coding of a procedure call, but it will transfer on the server-side a document (i.e. complex data structure). So, the servant object will typically perform the procedure started by the client and at the end will send out another document that will contain the response.

In the SOAP envelope, there is also a SOAP header, which is used to extend the usage of the protocol, allowing for example to specify security checksum, transactional context, etc. In SOAP the binding is the definition of how the XML description of the message should be sent using the specific transport protocol. The typical binding for SOAP is HTTP. It can use GET and POST. With GET, the request is not a SOAP message but the response is a SOAP message. With POST both the request and response are SOAP messages. The error and status code are the same used in HTTP so that HTTP responses can be directly interpreted by a SOAP module. From the point of view of the infrastructure we need an implementation of the client that will implement an application logic that deals with the SOAP engine, the component that will be in charge of coding the request and decoding the response in XML, but we will transmit them to an HTTP engine. SOAP has many advantages like simplicity, portability, firewall friendliness, uses open standards, interoperability, and universal acceptance. However, it has some disadvantages such as: it relies too much on HTTP (which is stateless) and perform the serialization by value and not by reference (when we have to send information we need to serialize them and send them as a data structure typically coded in XML over the other side). In addition to SOAP, the basic elements that constitute a web service are the description language. A service description is needed because in this way the service can be discovered and used by other services and applications. Web service consumers should be able to determine the precise XML interface of a web service. This can be done by leveraging on top of XML schema, but describing some elements:

- **Service description**: it should be machine-understandable standards that describe what are the operations offered by the web service and also the wire format and transport protocol that the web service uses to expose his functionality. Finally, we need also to describe a type

system that describes the types of parameters that will use in the payload.

At this point having a service description in place and a SOAP infrastructure we can really have machine and implementation language-specific elements that allow service requestors and service providers to be independent as each one can discover the other dynamically, and this makes the whole system interoperable. This language was called **Web Service Description Language (WSDL)**, which is an XML-based service representation language used to describe the details of the interfaces exposed by web services and thus means to access the web service. Basically, WSDL represents a contract, which is in place the client and servant object. Assume that we have a service provider. It describes in WSDL the contract, i.e. declare which are the specification conditions according to which the clients can use the service, not only at the technical level (parameters, type system, etc.) but potentially we can also describe the SLA of my service and other characteristics that we want to enforce. Then the service requestor is able to find in WSDL and he can decide to accept or no the previous contract. Once that he has accepted the contract, he can connect and bind through SOAP to the specific service and therefore is again WSDL that governs this binding and connection via SOAP. In WSDL a contract between a service requestor and provider should specify the following elements:

- **what** a service does, i.e. the operations the service provides.
- **where** it resides, i.e. details of the protocol-specific address.
- **how** to invoke it, i.e. details of the data formats and protocols necessary to access the service's operations.

A WSDL document is organized in two different sections:

- **service-interface definition**: it describes the general web service interface structure. This contains all the operations supported by the service, the operation parameters, and abstract data types.
- **service implementation part**: it binds the abstract interface to a concrete network address, to a specific protocol and which concrete data structures are used. In this way, the client may bind to such implementation and to invoke the operations.

This enables each part to be defined separately and independently, and reused by other parts. A WSDL document is organized according to the following tags:

- $< types >$: data type definition
- $< message >$: operation parameters
- $< operation >$: actions/operations description that the service provides
- $< portType >$: set of operation definitions
- $< binding >$: operations bindings (i.e. technical mapping of operations on a specific transport protocol)
- $< port >$: it's a concrete endpoint associated with the binding
- $< service >$: it's the real location/address for each binding
- $< import >$: it allows us to reference other XML documents.

WSDL defines four main messaging exchange patterns. When we have to interact with a remote module we can use two modes: the sender sends a message to the receiver asking for something, and he does not wait for any response. This is what is called **one-way message exchange**. Conceptually, we can see it as an invocation of a procedure. Instead, with the second approach, the sender sends a message to the receiver asking for something, and he asynchronously waits for a response. This is what is called **two-way message exchange**. Then in SOAP were proposed other two types of interaction called **Notification messaging** and **Solicit messaging**. The first one means that the receiver sends something to the sender and he doesn't expect any response from the sender. Solicit means that the receiver sends something to the sender and he expects a response from the sender like an acknowledgment.

5.2 Registry and UDDI

The third element of a web service infrastructure is the registry. In SOAP-based services, the registry is called **UDDI**. As we said in SOA we should have these types of interactions: the service provider publishes the service and service description, introduce the registration of the requests in a service registry. The service requestor queries for all matching services in the registry on the basis of the needs that he may have. He discovers the results, then requests the possible service information and the service registry returns the information of the selected service. At this point, the service requestor can invoke the requests including possible outputs and achieve

the invocation results. The service registry solves an important problem in the infrastructure that is how to find the service that a client wants to use among a large collection of services and providers.

Universal Description, Discover, and Integration (UDDI) was the standard for the definition of the registry in SOAP web service and was at the beginning a real implementation of this description that should support the web service publishing and discovery processes. In the UDDI philosophy a business should **describe** its business and its services, **discover** other businesses that offer desired services and **integrate** with these other businesses. The design of UDDI is organized in three main components:

- *white pages*: it's an entry-point which gives us the organizational details like the address, the contact, and other key points for contacting a person or an organization.
- *yellow pages*: it's a way of classifying the business (i.e. the information of the white pages) on the basis of standard taxonomy.
- *green pages*: it represents technical capabilities and information about services (i.e. they can be seen as a manual).

Typically UDDI is itself a web service, i.e. it can be accessed through SOAP HTTP. It offers two types of operations: **inquiry URL** is used by service requestor to requires some information of a web service, and **publishing URL** is used by the service provider to publish services description and then the WSDL. In UDDI while the white and yellow pages are offered by the UDDI registry, then the green pages aren't stored on the UDDI registry but are stored as a simple pointer back to the WSDL description in the service provider. Suppose that a service requestor is looking for a service description that contains specific data types in the parameters of operation. In this case, the UDDI should conceptually enter into each UDDI entry, but to be able to filter out which entry contains the specific data types in the operation definition, it needs to retrieve back from each service provider the WSDL file to be able to look into it. Technically this means that we need to being a registry and not a repository, to materialize on the fly all the possible service description for each query, and this is very costly, because the information is not stored locally in the service registry, but are stored remotely in the service providers. This is one of the issues that UDDI encountered during its practical application. The UDDI was designed as a registry and not as a repository because the design of the registry is very flexible from the point

of view of the service provider. The service provider, since it doesn't store in the central registry the technical information is free to update and evolve the service description on the fly without continuously updating back in the UDDI service description. One of the drawbacks of UDDI, is that it's really a provider-oriented service, but not service requestor oriented.

5.3 RESTful

A service mash-up is a web application that combines data from one or more sources (e.g. open APIs) into a single integrated tool. The difference with SOAP web services is that the way of building the application is completely different. This approach is based on several technologies such as SOAP, RESTful, and Atom/RSS. The RESTful web services started to emerge toward the end of 2000 and they started to be considered as an alternative to WSDL for SOAP-based web service. To compare the two approaches is important to analyze what are conceptually their differences. Typically, when dealing with web services is common to classify the possible standards that are built on top of web services in four different layers:

- **Messaging**: how a client and servant can interact.
- **Single service**: what are the standards and technologies that helps to develop and to describe single services.
- **Multiple interacting services**: what are the technologies and services that deals with the composition and orchestration of multiple services.
- **Registry/Repository and discovery**: it deals with registry or repository, and possible discovery of services.

Conversely, in RESTful based services with don't have the last two layers. REST is not really a technology, but it's an architectural approach. It's a kind of method in which by adopting specific conventions to use the technology we can achieve the results we may be interested in. The idea of REST is that we can transmit data directly over HTTP without including any additional layers as SOAP. A traditional web page is consumed by a user through a web browser. Why can't we consume a web page through a program? We can do that, but this requires a style of building this infrastructure (the REST one) since the information from a web page should be extracted and returned to the client. We know that in HTTP we have basic verbs such as GET, POST, UPDATE, and DELETE. RESTful web

services are based on the concept of these HTTP verbs, and the resources will represent the states of required data items, i.e. their values. REST is a kind of RPC, but the idea is that it isn't an RPC system in which is the developer that each time through the definition of the service interface defines the name and the semantics of the methods. Instead, it's an RPC system in which the methods have been already defined in advance, and the developer has only to associate to this predefined methods (CRUD operations) the resources the methods may want to access. The RESTful basic principles are the following:

- **Addressability**: it means every resource should have an URL.
- **Uniform interface**: we already said that, all the services have the same interface, which is based on the HTTP verbs.
- **Stateless interaction**: there isn't any concept of session (i.e. each interaction is stateless). We access the resource and read it.
- **Self-describing messages**: it means that the messages are self-describing because they are part of HTTP.
- **Hypermedia**: they are just another type of link.

REST is incompatible with endpoint because the resources are provided through URLs. Instead, in RPC we can either address data objects (REST does this one) or software components (endpoint RPC does this one). In REST there exist three different levels of the maturity of the API, where:

1. **HTTP as a tunnel**: REST just using HTTP as a way to do RPC, but we don't model properly.
2. **Resources**: when we correctly identifies all the resources of our application and access them through HTTP.
3. **HTTP verbs**: we correctly adopt the HTTP verbs.
4. **Hypermedia**: we are addressing in the proper way also hypermedia content.

RESTful was considered so trendy because it's easy and lightweight, and starting from 2010 the big companies like Amazon, Yahoo and Google started to offer their web services as RESTful.

6 Function points

Function points are a method that allows software engineers to evaluate the dimension of software projects to give an estimation of how much time and how much is the cost to be required to the client. Typically, to measure the software, there are different software metrics:

- **Direct metrics**: they are those that we can measure directly on the software. Some of them are the following:
  - **Line of Code (LOC)**: the number of lines of code that has been written.
  - **McCabe index**: it's an index to measure the complexity of the software, i.e. how much cycles and decision points are present in the software.
  - **Transactional (FP)**: the number of transactions that the software has to perform.
- **Indirect metrics**: they are those that we can derive for example by basing on the SLA or those that are considered the users' opinions.

The metrics that deal with dimensions are mainly based on LOC. It appears very simple to compute because is just looking at the source code of a program and count how many lines have been written. During the years have been proposed other metrics and, in particular, one of them that is still widely used is the **function points (FP)**. It's an empirical formula based on functionalities. The concept when we develop a program is the following: we have an application that is the one being considered. This application interacts from one side with the users and on the other side interacts with possibly other applications. The idea of FP is to evaluate our application at the border, which is the border between software and users. We consider measuring what is inside. To perform this measure we will consider the following measurements:

- **Internal logical file (ILF)**: it's defined as a user-identifiable group of logically related data or control information maintained within the boundary of the application. The primary intent of an ILF is to hold data maintained through one or more elementary processes of the application being counted.
- **External interface file (EIF)**: it's defined as a user identifiable group of logically related data or control information referenced by

the application, but maintained within the boundary of another application. The primary intent of an EIF is to hold data referenced through one or more elementary processes within the boundary of the application counted. This means an EIF counted for an application must be in an ILF in another application.

- **External input (EI)**: it's an elementary process that processes data or control information that comes from outside the application boundary. The primary intent of an EI is to maintain one or more ILFs and/or to alter the behavior of the system.

- **External output (EO)**: it's an elementary process that sends data or control information outside the application boundary. The primary boundary of an external output is to present information to a user through processing logic other than, or in addition to, the retrieval of data or control information. The processing logic must contain at least one mathematical formula or calculation, create derived data maintain one or more ILFs, or alter the behavior of the system.

- **External inquiry (EQ)**: it's an elementary process that sends data or control information outside the application boundary. The primary intent of an EQ is to present information to a user through the retrieval of data or control information from an ILF or EIF. The processing logic contains no mathematical formulas or calculations, and doesn't create derived data. No ILF is maintained during the processing, nor is the behavior of the system altered.

The FP are weighted on the basis of three weights: **Low**, **Medium** and **High**. We create an adjusted FP, i.e. when we have calculated the non-weighted FP, the empirical formula says that we can change $\pm 35\%$ through a corrective formula that captures general characteristics of the system through a set of 14 indicators. Typically, for each indicator, we need to consider if it's not relevant or essential. Each indicator takes a value between 0 and 5. The formula for the AFP is the following:

$$AFP = Total \times (0.65 + 0.01 \times \sum_{i=1}^{14} F_i).$$

The FP advantages are the following: they are widely used and accepted, certified personnel is available, the calculation is objective, UFP is independent on the technology, they can be used early in the development process and are equally accurate as SLOC. However, they also have some bad points such as they are semantically difficult, they are incomplete, there is a lack of

automatic count and there are many different versions. Let's start talking about the components of data functionalities (ILF and EIF):

- **Data element type (DET)**: it's a user identifiable single field within an ILF/EIF.
- **Record element type (RET)**: it's a user identifiable group of fields within an ILF/EIF.

Naturally, the way of computing the complexity of an ILF or EIF is to count the number of RET/DET that we may derive from it. Now, let's deal about the components of the transactions (EI, EO, and EQ):

- **File type referenced (FTR)**: we will count how many references to ILF or EIF we consider.
- **Data element type (DET)**: as before.

Let's see an example on this topic. Our goal is to build an invoice data. The sales division wants to handle the following customer's information: name, FC, IVA, legal address, corresponding address, phone, and invoices. Then for each invoice, we have an invoice number, the issue, and payment date. Then, for each item, we want to have the description, the number of items bought, IVA percentage, and unit cost. All the information about an item comes from an external application. The first task is to estimate the FP of an application for CRUD operations. The customer search is carried out using the FC, while that of invoices is done by considering the invoice number. The second task is to print an invoice from the screen and visualize it. The third task is to print a customer at the screen and visualize it. One approach is not to deal only at a low level, but to have also a conceptual schema (like ER schema or UML class diagram). At this point, is simpler to evaluate an ILF for customer and invoice and an EIF for an item. For the customer how many records do we have? For the customer, we have 6 DET and 1 RET. For the invoice, we have 4 DET (including also #items) and 1 RET. Instead, for the item, we have 3 DET and 1 RET. Now, according to the ILF/EIF complexity table, we obtain for the two ILF a low complexity, which corresponds to 14 FP taking as value 7. In the same way, we obtain for the EIF a low complexity as well, which corresponds to 5 FP, taking as value 5. If we switch to the transactional part, we will have 6 EI: insert, delete, and update both for customer and invoice. Looking at the complexity, we will have for customer 1 FTR for all the previous operations, and 6, 1 and 5 DET respectively. For the invoice, we will have 1 FTR for the delete operation, 3

FTR for insert and update operation, and 1, 8 and 7 DET respectively. The transaction complexity for insert and update operation are high: We will have 4 EI with low complexity and 2 EI with high complexity, with 12 FP each. For EO we will 3 FTR and 10 DET, and according to the complexity table, we will figure out that it has a medium complexity, which corresponds to 5 FP. Then we have 1 EQ, to which correspond 1 FTR and 6 DET. It has a low complexity, which corresponds to 3 FP. If we sum up all the FP obtained so far, we get 51 UFP.

7 Effort estimation

Effort estimation is a technique used to estimate the effort to develop a software project. The idea is the following: given a set of requirements, effort estimation can be seen as a black-box, which allows us to derive three important metrics, cost, effort, and time. The steps that will follow to derive these estimations are:

- we start from the requirements and we derive FP
- then we move from FP to Line of Code
- we use LOC to derive two important metrics (time and effort)
- once we have the effort we can derive the cost.

To derive metrics from LOC we will use a technique called **Constructive Cost Model (CoCoMo)**. The CoCoMo main idea is to estimate an effort M (the unit can be man-day, man-month or man-year)and the optimal timing T (the unit can be years, months, or weeks) for delivering a project. It relies on statistics and follows a waterfall model. It provides an effort indication on four phases: analysis and planning, design, development, integration, and test. From the previous two metrics, we can derive another metric called **manpower**, which represents the number of people working during the project execution ($manpower = \frac{M}{T}$). In CoCoMo we have some adjusted parameters that estimate the context in which the software is developed. Typically, several parameters are evaluated on an ordinal scale with the following values: very low, low, nominal, high, very high, and extra high.

7.1 1981 model

In 1981 there are three types of models to be used according to the difficulty of the project, and the project can be estimated only in its dimension or according to the correction factors described above (represented here as c_i). If we evaluate only the project dimension, assuming that the requirements don't change and representing with S_k the estimated lines of code, we have the following formulas:

- **Simple**: $M = 2.4 \cdot S_k^{1.05}$ and $T = 2.5 \cdot M^{0.38}$
- **Intermediate**: $M = 3.0 \cdot S_k^{1.12}$ and $T = 2.5 \cdot M^{0.35}$
- **Complex**: $M = 3.6 \cdot S_k^{1.2}$ and $T = 2.5 \cdot M^{0.32}$.

If we consider the global correction coefficients we obtain:

- **Simple**: $M = 3.2 \cdot S_k^{1.05} \cdot \prod_1^{15} c_i$ and $T = 2.5 \cdot M^{0.38}$
- **Intermediate**: $M = 3.0 \cdot S_k^{1.12} \cdot \prod_1^{15} c_i$ and $T = 2.5 \cdot M^{0.35}$
- **Complex**: $M = 2.8 \cdot S_k^{1.2} \cdot \prod_1^{15} c_i$ and $T = 2.5 \cdot M^{0.32}$.

When working with CoCoMo we need to make some assumption in our project: it's based on waterfall model, T encompasses design-coding-integration and test (requirement analysis and planning are not considered), MM are 152 hours (19 days of 8 hours), the requirements are stable, it employes adequate personnel and we have good project management. With our assumptions, we have an error $< 20\%$ of about 68% of estimates.

7.2 CoCoMo II

For this reason, it was proposed a second model called **CoCoMo II**. The main motivations for this proposal were: new life cycle software models, reuse, and different levels of estimation precision. In CoCoMo II we have two main models:

- **Early design model**: it's useful and suitable for the project initial phase. We don't have too many details, i.e. our estimations are only based on FP. We have about 7 adjusting factors.
- **Post-architecture model**: it's used for the development and the maintenance phase. We have more details and information, i.e. we don't consider only the FP but also the possible reuse of the software. We may have until 17 adjusting factors.

The two models share 5 scaling factors for computing the exponent factors:

- **Precedentedness (PREC)**: familiarity with the work, given by past similar works, it's intrinsic to the project and uncontrollable.
- **Development Flexibility (FLEX)**: flexibility and relaxation during work, intrinsic to the project, and uncontrollable.
- **Architecture / Risk Resolution (RESL)**: combines design thoroughness and risk elimination strategies included in the project.
- **Team Cohesion (TEAM)**: sources of project turbulence given by difficulties in synchronizing the stakeholders (difficulties created by people involved in the project).
- **Process Maturity (PMAT)**: process maturity level measured according to the CMM-levels. If CMM-levels are not available the EPML (Estimated Process Maturity Level) is computed as the percentage of compliance to the 18 Key Process Area goals by CMM, according to the following formula: $EPML = 5 - [(\sum_{i=1}^{n} \frac{KPA\%_i}{100} \cdot \frac{5}{18})]$.

CoCoMo II formulas Let SCED be the required development schedule, n be either $6 + SCED$ or $16 + SCED$, EM_i be the effort multipliers that adjust the model according to the environment and A, B, C, D constants. We compute the person-months, PM , in the following way:

$$PM = A \cdot S^E \cdot \prod_i^n EM_i,$$

where S is the estimated size of the project in KLOC. The PM computation requires the computation of the economy/diseconomy of scales, E (in CoCoMo 1981 there were only diseconomies), where SF_j are the five scale factors that CoCoMo II uses:

$$E = B + 0.01 \cdot \sum_{j=1}^{5} SF_j.$$

The development time T is computed as follows:

$$T = C \cdot PM^F \cdot SCED/100$$
$$F = D + 0.2 \cdot (E - B).$$

Effort multipliers We have 17 effort multipliers that can be used in the second model and only 7 of them can be used in the first model, these multipliers are divided into different categories:

- **Product**:
  - **RELY**: Required product reliability
  - **DATA**: Database size
  - **CPLX**: Product complexity
  - **RUSE**: Intended reuse of software models
  - **DOCU**: Level of required documentation
- **System**:
  - **TIME**: Execution time constraints
  - **STOR**: Main storage constraint
  - **PVOL**: Platform volatility (change frequency)
- **Personal**:
  - **ACAP**: Analyst capability
  - **PCAP**: Programmer capability
  - **APEX**: Application experience
  - **PLEX**: Platform experience
  - **LTEX**: Language and tool experience
  - **PCON**: Personnel continuity
- **Project**:
  - **SITE**: Multisite development
  - **TOOL**: Use of software tools
  - **SCED**: Schedule constraints.

All these multipliers make both models very variable when determining the effort needed to develop the project, even if the models have different multipliers. With the CoCoMo II model, we can estimate with a good approximation the time necessary to the delivery of the software and the effort that we need to complete the project with a certain delivery time, so we can estimate the cost of the project, taking into account the cost that we sustain if we write new code and the cost that we have when we reuse some modules.

7.3 Software reuse

We need to model software reuse since it is not a trivial process since the code we want to reuse can be subject to some modifications and has a certain level of familiarity, readability, and documentation. To take into account all these factors we use a non-linear module that models the effort to adapt an existing module as the effort to develop a new module, measuring the equivalent lines of code (ESLOC). This model is based on two aspects:

- the complexity to adapt software which is derived from:
  - **Software understanding (SU)**: how the software is easy to read, understand and modify to be used in the new project in terms of documentation, readability and modularity of the code (from low to high: [50, 40, 30, 20, 10]).
  - **Assessment and Assimilation (AA)**: if the code can be useful for the application and how its documentation can be integrated with the actual product through tests, evaluation to process, and documentation that needs to be written. (from none to extensive: [0, 2, 4, 6, 8]).
  - **Programmer Unfamiliarity(UNFM)**: of the software to be integrated (from familiar to unfamiliar: [0, 0.2, 0.4, 0.6, 0.8, 1]).
- the percentage of modification, the **Adapting Adjusting Factor (AAF)**, which is determined using the following metrics:
  - **DM**: percentage of the modified design
  - **CM**: percentage of modified code
  - **IM**: percentage of the modification of the integration effort without reusing code.

We can compute the Equivalent SLOC in two ways, depending on the adapting adjusting factor (AAF):

$$AAF = (0.4 \cdot DM) + (0.3 \cdot CM) + (0.3 \cdot IM)$$
$$AAM = \begin{cases} \frac{[AA+AAF\cdot(0.02\cdot SU\cdot UNFM)]}{100} \text{ if } AAF \leq 50 \\ \frac{AA+AAF+(SU\cdot UNFM)}{100} \text{ if } AAF > 50 \end{cases}.$$

After having computed both the Adaptation adjustment factor and modifier, we have that the estimated KLOCs are:

$$EKLOC = AKLOC \cdot (1 - \tfrac{AT}{100}) \cdot AAM,$$

where AKLOC are the adapted lines of code that are modified to be of use in the actual project and AT is the percentage of code that is re-engineered by automatic translation. With this parameter, we can compute the relative cost and modification size required to reuse a piece of code in a project.

7.4 Backfiring

After we have computed our function points we can compute with them the SLOC: to do that we can use tables which indicate the backfiring (the average correspondence between the lines of code SLOC and the function points). The backfiring tables can be consulted from page 59 of CoCoMo slides. The LOC estimation can be done using the given data as follows:

- 51 FP
- C Language
- Backfiring for C language = 128
- LOC = $51 \times 128 = 6528 = 6.528$ KLOC.

8 DevOps

In simple words, DevOps means a combination of software development operations. It's a widespread approach in modern software engineering, in which the operations started to make use of many of the same techniques as developers for their systems work. The business problems that emerged during the last year are the following:

- need more time to respond to market changes
- deployments held off to avoid risks
- slow and error-prone releases
- fix and maintain rather than innovate
- unstable operations as fixes take more time
- IT is frequently seen as the bottleneck in the transition of "concept to cash".

The symptoms of these problems are that the developers tends always to work on their machine to minimize the risk that the software release is unstable on another platform, is needed to have production environment access to diagnose issues, servers are not available for deployment (it could fail due to incorrect configuration), fix performed after a specific day, releases can have a lot of fails. The business trends that move to DevOps are frequent deployments, faster recovery from failures, reduced failure rate, shorter lead times and to reach better customer satisfaction. If DevOps is correctly implemented, it increases the velocity, reduces the downtime and human errors. DevOps is based on the following principles:

- in big companies teams are in charge of doing the development of the application. Typically, they change, modify, and test the software. When the application is ready, they move it to the IT operations team. They represent the team in charge of maintaining the operations run on the infrastructure, monitor the server infrastructure performance and this implies that they have to enhance the stability and the maintenance of the service. The fundamental aspect is the quality of this process. We are moving from a single person's responsibility to collective responsibility, shared understanding, and service delivery.

- DevOps is a set of practices that emphasize the collaboration and communication of both software developers and IT professionals while automating the process of software delivery and infrastructure changes. Its goal is to bridge the gap between agile software development and operations. This is possible by unifying people, processes, and products to have continuous delivery of value to our end users.

- the basic idea is that the code should be managed in a code repository, i.e. a remote repository accessible by all developers and people involved in the process. Then there should be an automatic building process and testing process. Then we have the deployment process managed in a structured way (e.g. through a database), and finally, it will be monitored and potentially improved.

The current software development situation is constituted by well-separated areas: business, development, and operations. In this case, each area doesn't communicate too much with the other ones. The idea with DevOps is that the business moves and we have a unique team that manages the development and the operations. The **continuous integration** is a fundamental aspect in DevOps, since it's the process of integrating code into a mainline

code-base. Its main key elements are: version/source control, frequent commits, build automation as well the testing, test outcome results availability, code, and build stability, code-quality, and coverage. In this ways the bugs are detected very early, we have immediate feedback on the system-wide impact of local changes, we have constant availability of the current build for testing, demo or release purposes, enforces the discipline of frequent automated testing and allow to have faster time to release with a repeatable process. However, it has also some disadvantages such as: automated test suites require a considerable amount of work to set-up and also for ongoing needs, work involved to set-up a build system, the added value depends on the quality of tests and how testable the code is, build queueing up can slow down everyone and partial code could easily be pushed and therefore integration tests could fail until the feature is complete. The **continuous delivery and deployment** are other important aspects of DevOps. The first one means to make sure that your software is always production-ready throughout its entire life cycle, i.e. any build could potentially be released to users at the touch of a button using a fully automated process in a matter of seconds or minutes. Continuous deployment is the practice of releasing every good build to users. It deploys every change that passes the automated tests to production and is the next phase of continuous delivery.

There are 9 types of DevOps tools which are known before choosing for the project:

- **Collaboration tools**: this type of tool is crucial to help teams working together more easily, regardless of time zones or location. It's a rapid action-oriented communication designed to share knowledge and save time
- **Planning tools**: this type of tool is designed to provide transparency to stakeholders and participants.
- **Source control tools**: tools of this sort make up the building blocks for the entire process ranging across all key assets.
- **Issue tracking tools**: these tools increase responsiveness and visibility.
- **Configuration management tools**: without this type of tool, it would impossible to enforce desired state norms or achieve any sort of consistency at scale.

- **Database DevOps tools**: the database needs to be an honored member of the managed resources family.
- **Continuous integration tools**: this type of tool provide an immediate feedback loop by regularly merging code. **Automated testing tools**: tools of this sort are tasked with verifying code quality before passing the build.
- **Deployment tools**: these tools are essential to checking those boxes.

A User story

BDD asks questions about the behavior of the application before and during development to reduce miscommunication. The application's requirements are written as **user stories**. A user story has the following structure:

- **As a** [kind of stakeholder]
- **So that** [I can achieve some goal]
- **I want to** [do some task].

The idea is to use user stories as acceptance tests before the code is written. A measure of team productivity could be the average number of stories/week ? We have a problem that some stories could be more difficult than others. So, a simple fix is to assign to each user story a rate on a simple integer scale: for example, we could assign 1 for simple stories, 2 for medium stories, and 3 for very complex stories. In this new setting, the velocity is given by the relationship between the average number of points and the week. There are some guidelines to define properly a user story. One of those is, if the rate assigned to a user story is greater or equal to 5, then we could think to split this story into simpler stories. Another guideline which each user story should follow, is the SMART approach:

- **Specific and Measurable**: each scenario should be testable. It implies known good input and expected results exist. For example, Given some specific starting condition, When I take specific action X, Then one or more specific events should happen.
- **Achievable**: the ideal case is to complete user stories in one iteration. In the real world, this situation never occurs. If we can't deliver the feature in one iteration then we deliver a subset of stories.

- **Relevant**: of course, the user stories should represent an important feature. (see also 5 Whys approach). Otherwise, we can delete useless stories.
- **Timeboxed**: naturally if a story exceeds the time budget we need to stop it. We can divide it into simpler stories or reschedule what is left undone. This is an important characteristic, because in this way we avoid underestimating the length of the project.

When working with the customers it is useful to present user stories also in graphical form via Lo-Fi UI that gives to the customer a high-level idea of how the application will look. Another important aspect is to include also a storyboard that shows how the UI changes based on user actions.

B Domain Driven Design

It's a way of thinking and a set of priorities, whose objective is to accelerate software projects that have to deal with complicated domains. DDD's main idea is to adopt a model-driven software design approach used to tackle the complexity of software projects. This means that DDD is also a collection of principles and patterns that allow developers to build very elegant and maintainable systems. The definitions of DDD are the following:

- **Domain**: it's a sphere of knowledge or activity. What an organization does and the world it does it in.
- **Model**: it's a system of abstractions that describes specific aspects of a domain and ignores extraneous details. The idea of a model is to explain a complex domain in a simple way. A model is a distilled form of domain knowledge, assumptions, rules, and choices.

In particular, the DDD main principles are:

- speak a ubiquitous language within an explicitly bounded context. A ubiquitous language is a language structured around the domain model and used by all team members to connect all the activities of the team with the software. It should be used consistently in speech, documentation, diagrams, and code. The idea is that if we adopt a change in our language then it should be reflected in the model and code, and vice versa.
- explore models in a creative collaboration of domain practitioners and software practitioners.

- focus on the core domain.
- model and implementation are bound (the developers are also responsible for the model).

The model can be expressed through class diagrams, (UML is the preferred way), sequence diagrams, etc. This design document is not the model, is how we communicate and explain the model. The model is ultimately expressed in the code. Another important aspect of DDD is that it's agile and iterative. The problem here is with the **Big Design Up Front** approach. In this approach at the beginning, we have domain experts and business analysts who create the analysis model and then hand it over the developers (usually it's done in UML). Then in the first iteration, the team of developers starts to have the initial code model that matches the analysis model. In the successive iterations the model evolves with the abstraction, for example, the team discovers an issue with the analysis model and develops away from it and the analysis model starts to become useless. There is no feedback loop, the descriptive domain terms are lost and the insight that we get is not transferred in the model. In the end, the code model no longer reflects the analysis model. The DDD process we work in a different way: at the beginning, we should have the stakeholders that communicate the business goals and the input and outputs of the system, and a development team which captures them as business use cases. Then there is the knowledge crunching phase, in which domain experts and the development team produce a model that satisfies the needs of business use cases. This model should simplify reality as much as needed to understand the problem domain. We have our UML model, we code it, but when we change our code we need to change also the analysis model to keep it in sync with the code model. This is the main idea of the so-called **One Team, One Language, One Model** principle. The DDD's second principle is based on the idea of breaking down a complex domain, which in turn is based on the bounded context concept. It's an operational definition of where a particular model is well-defined and applicable. Typically this can be mapped to a subsystem, i.e. it's a part of the domain, based on a particular conceptual decomposition of the domain. Typically when we use DDD the architecture we want to enforce is the layered one, which divides the layers in:

- **User interface**: it's responsible for presenting information to the user and interpreting the user commands.
- **Application**: it's a thin layer that coordinates the application activity. It doesn't contain business logic, it doesn't hold the state of the

business objects.

- **Domain**: this layer contains information about the domain. This is the heart of the business software. The state of the business objects is held here. We need to keep in mind that the persistence details are delegated to the infrastructure layer.
- **Infrastructure** : this layer acts as a supporting library for all the other layers. it should provide the communication between layers, implement persistence for business objects, etc.

In DDD the models expressed in the software are the following:

- **Entities**: they are objects which have an identity that remains the same throughout the states of the software. Basically, this is the way to distinguish similar objects having the same attributes. The attributes of an entity can change. The entities should have a behavior, but no persistence behavior.
- **Value object**: they are the "things" within your model that have no uniqueness. They are only equal to another value object if all their attributes match. They are immutable (the attributes must be replaced).
- **Aggregates**: it's a cluster of entities and value objects. The idea is that each aggregate is treated as one single unit. Each aggregate has one root entity know as the **aggregate root**. The root identity is global, the identities of entities inside are local. The external objects may have references only to root. The internal objects cannot be changed outside the aggregate.
- **Associations**: they are relationships between concepts, imposing traversal direction. It adds qualifiers to reduce the multiplicity and eliminate non-essential associations.
- **Services**: they are those elements that reside in multiple layers. Services usually manipulate multiple entities and value objects. They are stateless. A service should be offered as an interface that is defined as a part of the model. Its parameters and results should be domain objects.
- **Factories**: it's an object whose responsibility is the creation of other objects. It creates and manages complex domain objects. It's very useful for creating aggregates.

- **Repositories**: it encapsulates domain objects' persistence and retrieval. We have a clean separation and one-way dependency between the domain and data mapping layers. It acts as a collection except for some elaborate querying capability. We should have one repository for aggregate.
- **Modules**: they break up our domain to reduce the complexity. There is a high cohesion within the module, loose coupling between modules. It becomes part of the ubiquitous language and helps with extensibility.
- **Context mapping**: it's a mapping between the contact points and translations between bounded contexts.