Chapter 1 Introduction to Software Engineering

Dr Soumia LAYACHI

September 2025

1 Computerization

Computerization is the most important phenomenon of our time. Currently, IT is at the heart of all large companies. A company's information system is made up of hardware and software. More specifically, investments in this information system are generally distributed as follows.

- 20% for hardware
- 80% for software. In fact, for several years now, hardware manufacturing has been handled by only a few manufacturers. This hardware is relatively reliable, and the market is standardized. IT-related problems are essentially software problems.

2 Software

Software or an application is a set of programs that allows a computer or computer system to perform a particular task or function (example: accounting software, loan management software).

Software, depending on its size, can be developed by a single person, a small team, or a set of coordinated teams.

In 1995, a study by the Standish Group painted a damning picture of IT project management. Based on a representative sample of 365 companies, totaling 8,380 applications, this study established the following.

- Only 16.2% of projects were in line with initial forecasts,
- 52.7% had experienced cost and time overruns of a factor of 2 to 3 with a reduction in the number of functions offered,
- 31.1% were purely abandoned during their development.

For large companies (which launch proportionally more large projects), the success rate is only:

- 9% of projects completed successfully,

- 37% of projects are stopped during implementation,
- 50% are completed late and over budget.

3 Software Engineering Definition and objectives of software engineering(GL)

3.1 Definition

A field of engineering science whose purpose is the design, manufacture, and maintenance of complex, safe, and high-quality software systems (Software Engineering in English). Today, the economies of all developed countries depend on software systems.

GL is often defined in contrast to 'programming', i.e. the production of a program by a single individual, which is considered 'easy'. In the case of GL, it is the collective construction of a complex system , embodied by a set of design documents, programs, and test sets, often with multiple versions ('multi-person construction of multi-version software') and considered 'difficult'.

3.2 Objectives – the rule of CQFD

The GL is concerned with software manufacturing processes in order to ensure that the four criteria (cost, quality, functionality, and deadlines) are met.

- The system that is manufactured meets the needs of users (functional correction).
- The quality corresponds to the initial service contract.
- Costs remain within the limits initially planned.
- The deadlines remain within the limits initially planned.

These qualities are sometimes contradictory (chic and cheap!). They must be weighed according to the circumstances (critical software / consumer software). It is also necessary to distinguish between custom-made systems and mass-market software products.

4 The principles of software engineering

These principles are very abstract and not directly usable . But they are part of the basic vocabulary of software engineering. These principles have a real impact on many aspects and constitute the most stable type of knowledge, in a field where tools, methods and techniques evolve very quickly. They can be summarized in seven fundamental principles

4.1 Rigor

Software production is a creative activity, but one that must be conducted with a certain rigor. The maximum level of rigor is formality , that is, the case where descriptions and validations are based on mathematical notations and laws. It is not possible to be formal all the time: the first formal description must be constructed from non-formalized knowledge! But in certain circumstances, formal techniques are useful.

4.2 Separation of concerns

It is a common sense rule which consists of considering different aspects of a problem separately in order to master its complexity. It takes a multitude of forms :

- separation in time (the different aspects are addressed successively), with the notion of software life cycle that we will study in detail,
- separation of the qualities that we seek to optimize at a given stage (e.g. ensuring correction before worrying about efficiency),
- separation of the 'views' that one can have of a system (e.g.: concentrating on the 'data flow' aspect before considering the operation scheduling or ' control flow' aspect),
- separation of the system into parts (a kernel, extensions, etc.),
- etc.

4.3 Modularity

A system is modular if it is composed of simpler subsystems , or modules. Modularity allows the content of the module and the relationships between modules to be considered separately (which is in line with the idea of separation of issues). It also facilitates the reuse of well-defined components. A good modular division is characterized by strong internal cohesion of the modules (e.g., functional, temporal, logical, etc.) and weak coupling between the modules (limited number of clearly described inter-modular relationships). The entire evolution of programming languages aims to make modular programming easier, today called 'component-based programming'.

4.4 Abstraction

Abstraction consists of considering only the aspects of a system that are deemed important at a given time, while ignoring other aspects (this is another example of separation of concerns). The same reality can often be described at different levels of abstraction . For example, an electronic circuit can be described by a very abstract mathematical model (logic equation), or by an assembly of logical

components that abstract away the details of implementation (logic circuit) . Abstraction allows for better control of complexity.

4.5 Anticipation of change

The essential characteristic of software, compared to other products, is that it is almost always subject to continuous changes (corrections of imperfections and evolutions according to changing needs). This requires special efforts to anticipate , facilitate and manage these inevitable evolutions. For example, it is necessary to:

- Ensure that changes are as localized as possible (good modularity),
- Be able to manage multiple versions of modules and configurations of module versions, constituting versions of the complete product.

4.6 Genericity

It is sometimes advantageous to replace the resolution of a specific problem with the resolution of a more general problem. This generic solution (parameterizable or adaptable) can be reused more easily. Example: rather than writing an identification specific to a particular screen, write (or reuse) a generic authentication module (entering an identification - possibly in a list - and possibly a password).

4.7 Incremental construction

An incremental process reaches its goal in stages by getting closer and closer to it; each result is built by extending the previous one.

For example, we can first create a core of essential functions and gradually add the more secondary aspects . Or, we can build a series of prototypes 'simulating' the system envisaged more or less completely.

5 The basics of software quality

To assess the quality of a system, let's identify the factors we would like to find in all good software. In addition to providing the required functionality, good software must also possess the following key factors:

- Validity : the ability of a software to perform exactly the tasks defined by its specification,
- Reliability : the ability of software to continuously provide the expected service ,
- Robustness : the ability of software to function even under abnormal conditions,

- Extensibility : ease of adapting software to changes in specification,
- Reusability : the ability of software to be reused in whole or in part,
- Compatibility : ability of software to be combined with each other,
- Efficiency : the ability of a software to make good use of hardware resources such as memory , CPU power, etc.
- Portability : ease of being ported to new hardware and/or software environments,
- Traceability : ability to identify and/or track an element of the specifications linked to a software component,
- Verifiability : ease of preparation of acceptance and certification procedures ,
- Integrity : the ability of software to protect its various components against unauthorized access or modifications,
- Ease of use , maintenance , etc.

Note: It is difficult to optimize all of these factors because some are mutually exclusive (for example, providing a good user interface can reduce efficiency). Small improvements can be very expensive.

6 Modeling, life cycles and methods

6.1 What is a model

A model is an abstract and simplified representation (i.e. one that excludes certain details) of an entity (phenomenon, process, system, etc.) of the real world with the aim of describing, explaining or predicting it. Examples of models :

- Weather model

  based on observation data (satellite, etc.) allows us to predict climatic conditions for the coming days.

- The demographic model

  • defines the composition of a panel of a population and its behavior, with the aim of making statistical studies more reliable, increasing the impact of commercial approaches, etc.

6.2 Why model

Modeling a system before it's built allows for a better understanding of how it works. It's also a good way to control its complexity and ensure its consistency. A model is a common, precise language that is known by all team members and, as such, is a preferred means of communication.

6.3 The software life cycle

The software life cycle refers to all the stages of software development, from its conception to its disappearance. The origin of this division comes from the observation that errors have a higher cost the later they are detected in the production process. The life cycle makes it possible to detect errors as early as possible and thus control the quality of the software, its production times and the associated costs. The software life cycle generally includes at least the following steps:

Analysis and feasibility

- that is to say the expression, collection and formalization of the needs of the applicant (the client) and of all the constraints, then the estimation of the feasibility of these needs;

Specification or general design

- This involves the development of specifications for the general architecture of the software; Detailed design
- This step consists of precisely defining each subset of the software; Coding (Implementation or programming)
- it is the translation into a programming language of the functionalities defined during the design phases;

Unit tests

- they allow individual verification that each subset of the software is implemented in accordance with the specifications;

Integration

- The objective is to ensure the interfacing of the different elements (modules) of the software. It is the subject of integration tests recorded in a document;

Qualification

- i.e., verification of software conformity to initial specifications; Documentation

- it aims to produce the information necessary for the use of the software and for further developments;

Production

- it is the on-site deployment of the software; Maintenance
- It includes all corrective actions (corrective maintenance) and evolutionary actions (evolutionary maintenance) on the software.

The sequence and presence of each of these activities in the life cycle depends on the choice of a life cycle model between the client and the development team. The life cycle allows for taking into account, in addition to technical aspects, the organizational and human aspects.

6.4 Software lifecycle models

6.4.1 Cascading lifecycle model (or Waterfall model)

In this model, the principle is very simple: each phase ends on a specific date with the production of certain documents or software. The results are defined on the basis of the interactions between stages, they are subject to a thorough review and we only move on to the next phase if they are deemed satisfactory.

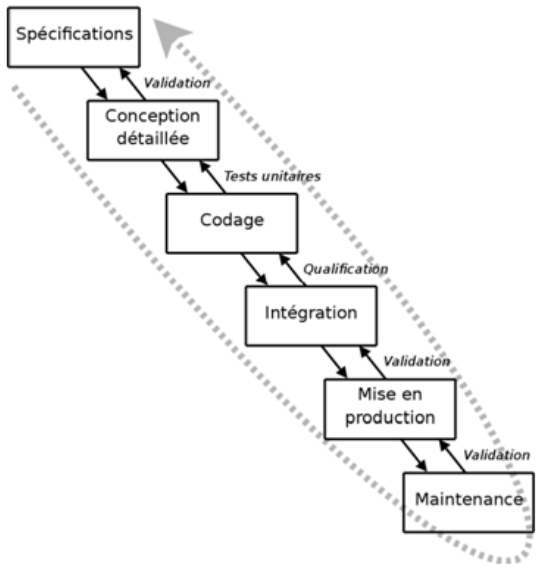

Figure 1: Cascading life cycle model

The original model did not include a backtracking option. This was added later on the basis that a step only calls into question the previous step, which, in

practice, proves insufficient. The major disadvantage of the waterfall life cycle model is that the verification of the correct functioning of the system is carried out too late: during the integration phase, or worse, during production.

6.4.2 shaped life cycle model (or V-model)

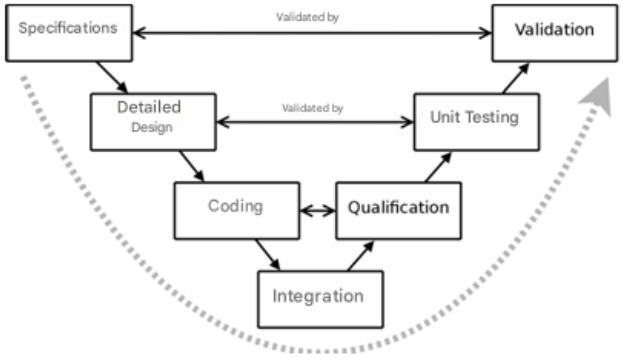

Figure 2: V model

The V model currently remains the best-known and certainly the most widely used life cycle. It is a waterfall model in which testing and software development are carried out synchronously. The principle of this model is that with every decomposition, the recomposition must be described, and that every description of a component is accompanied by tests that will ensure that it corresponds to its description. This makes explicit the preparation of the last phases (validation-verification) by the first ones (software construction), and thus makes it possible to avoid a well-known pitfall of software specification: stating a property that is impossible to verify objectively after implementation.

6.4.3 Spiral life cycle model

Proposed by B. Boehm in 1988, this model is much more general than the previous one. It emphasizes the activity of risk analysis: each cycle of the spiral takes place in four phases:

1. determination, based on the results of previous cycles, or the preliminary analysis of needs, of the objectives of the cycle, of the alternatives for achieving them and of the constraints;

2. risk analysis, evaluation of alternatives ;

3. development and verification of the chosen solution, a “classic” model (cascade or V) can be used here;

4. review of results and verification of the next cycle.

The preliminary analysis is refined during the first cycles. The model uses exploratory mock-ups to guide the design phase of the next cycle. The last cycle concludes with a classic development process.

6.4.4 Incremental Model

In previous models, software is broken down into separately developed components and integrated at the end of the process. In incremental models, only one set of components is developed at a time: increments are integrated into a previously developed software core. Each increment is developed according to one of the previous models. The advantages of this type of model are as follows :

- each development is less complex ;
- integrations are progressive ;
- it is therefore possible to deliver and put into service each increment;
- It allows for better smoothing of development time and effort thanks to the possibility of overlapping (parallelizing) the different phases.

The risks of this type of model are as follows :

- re-enforce the previous increments sighs the core;
- not be able to integrate new increments.

The kernels, the increments and their interactions must therefore be specified globally at the beginning of the project. The increments must be as independent as possible, both functionally and in terms of the development schedule.

6.5 Actual processes

There is no ideal model because everything depends on the circumstances . The waterfall or V model is risky for innovative developments because the specifications and design are likely to be inadequate and often questioned. The incremental model is risky because it does not give much visibility on the complete process. The spiral model is a more general framework that includes risk assessment. Often, the same project can mix different approaches , such as prototyping for high-risk subsystems and waterfall for well-known, low-risk subsystems.

References

[1] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language (UML) User Guide*. Addison-Wesley, 1999.

[2] Benoît Charroux, Aomar Osmani, Yann Thierry-Mieg. *UML 2 : Synthèse et exercices*. Pearson, édition française, ISBN 2-7440-7124-2, 2005.

[3] G. Booch et al. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2007.

[4] Laurent Audibert. Cours UML 2.0. Disponible en ligne : `http://www.developpez.com`