# Software engineering, across two centuries

Bertrand Meyer

Schaffhausen Institute of Technology, Innopolis University and Eiffel Software

Bertrand.Meyer@inf.ethz.ch
http://se.ethz.ch/~meyer, http://bertrandmeyer.com
Orcid: 0000-0002-5985-7434

**Abstract.** A survey of fundamental software engineering concepts, and their evolution since the time of IFIP's creation in 1960.

**Keywords:** Software engineering, software quality.

## 1 Introduction

The extraordinary development of information technology since the end of World War II has left almost no area of human activity untouched. It has been driven by the astounding (one runs out of superlatives, but how can it be otherwise with improvements by factors in the tens of billions since 1970 alone?) progress of hardware technology; but what lies at the core of the IT revolution is software. Software powers the world's devices and the world's processes.

Professional software construction is only possible through the systematic principles, methods, practices, techniques, notations and tools of software engineering, the art and craft of constructing quality software systems (a more precise definition appears below). Software engineering as a discipline was born in the late 1960s1, not long after IFIP itself. In the decades since then, many of the basic concepts have remained the same, but the challenges that the field faces have grown enormously, and so has the sophistication of software engineering.

This short survey, devised for the sixtieth anniversary of IFIP, summarizes both parts: the constants of the discipline, and how it has changed.

[1] An often repeated piece of supposed trivia states that the name was coined on the occasion of a 1968 conference, but the term was in use before, as attested by a 1967 reference found by the author.

## 2 Definitions

There exist lengthy and fancy definitions of “software engineering”, but in truth the term defines itself: software engineering is the application of engineering to the production of software. As to the constituent terms:

- “Software” denotes the specification of systems that require computers for their operation, where “computers” are automatic devices for information processing (involving computing, storage and communication). The term “specification” is broad enough to cover, under “software”, not only programs (source- or machine-level) but also auxiliary products such as requirements, designs and test suites.
- Since software distinguishes itself from other engineering products by its changeability (hence “soft”) and more generally by its virtual rather than physical nature, the term “production” should be taken in a broad sense as well, to include not only initial development but also deployment and updates.
- “Engineering” denotes (per its standard definition) the application of scientific methods to the construction of systems. Here one may quibble that not all people having “software engineer” as their job title apply scientific methods in their daily work. A scientific approach implies the use of mathematics: electrical and mechanical engineers routinely specify problems through equations then solve them. Mathematics does not directly play such a central role for most software development; while some areas such as life-critical systems increasingly rely on “formal methods” (mathematics-based approaches requiring proofs of correctness of the programs under construction), most software production remains largely informal. Even there, however, mathematics is indirectly present: the basic tools and concepts of software construction are defined with mathematical-like rigor. Programming languages are akin to mathematical notation, with the same need for precision in their definitions. Algorithms, the basis for programs, must also be expressed with as much rigor as a mathematical presentation requires.

While not strictly part of the definition, two features are essential to characterize software engineering as going beyond mere “software development” and requiring a true engineering approach: size and quality.

*Size:* modern software systems can be large in several respects; not just the sheer length of the programs (in source or object form), but the number and complexity of requirements to be satisfied, the number and diversity of users, the project’s duration (months, years, sometimes decades), the number of people involved in development, the variety of deployment situations, the number of bugs uncovered, the number of changes and extensions requested after an initial delivery... Without software engineering methods and tools, it would be impossible to master that complexity.

*Quality:* successful software must satisfy requirements of ease of use and learning, correctness (doing the job right), robustness (handling abnormal situations), security (handling hostile situations), efficiency (running fast and tight), extendibility (accommodating change of functionality and environment), timeliness (staying within schedule), cost-effectiveness (staying within budget), reusability (letting different developments benefit from each other's products) and others. Achieving these goals is difficult, in particular because of the inevitable tradeoffs, for example between ease of use and security.

To complement these definitions, we note that software engineering can seldom limit itself to the engineering of software only. Software systems typically exist in either a human context (for enterprise systems) or a material-world context (for embedded and "cyber-physical" systems, of which smart phones are a typical example). Although the present discussion limits itself to software concerns, they are often part of a more general systems engineering effort that must also encompass human and physical aspects.

## 3 Some universals of software engineering

Whether in 1960 or in 2020, software engineering is characterized by a number of fundamental concepts, some of which we now explore.

### 3.1 Tasks

All methods of software construction, in spite of their diversity, involve the following tasks. Tasks, not necessarily steps; how to order the tasks in time is a separate question, reviewed in the next subsection.

Any reasonable project must perform a **feasibility study**, meant to decide whether it is worthwhile to build a system. Not all problems have a solution in software (sometimes, for example, it is preferable just to change the human processes); and for those that do call for a software solution, it may be suitable to reuse an existing software system, or to purchase one from the market. The following tasks assume that a decision has been made either to build a new system ("greenfield" project) or to adapt an existing one ("brownfield").

Any development project needs **requirements**. In 1960 and still in 1980, anyone in the field would have defined requirements as "*specifying what the system will do*". The modern view can be expressed by the "4 PEGS" acronym (devised by the author but reflecting, we believe, a general understanding): Project, Environment, Goals and System. The requirements set the parameters of the project; they express the properties of the environment (in the sense of the business or natural-world features that bind the future system, and which the development team has no power to change); they reflect business goals for the commissioning organization; and they specify the behavior of the future system (the old "*what the system will do*"). Requirements engineering is a core part of software engineering, critical in particular because the best program is of little value if it does not address the right problem as perceived by project stakeholders, or addresses it in a way that is not acceptable to them.

Software construction requires *design*. This task consists of defining a high-level structure, or "architecture", for the system. "Decomposing systems into modules", the title of a classic article from the 1970s, is a particularly important task for the large systems developed today (think for example of modern operating systems with tens or even hundreds of millions of line of code). A good architecture is, among other qualities, one that clearly delimitates modules, protecting each from errors and security attacks originating in the others and making it possible to develop and later modify each independently of the others.

There will always be a task of **implementation** (or “coding”). With modern programming languages, coding is not radically different from design in its spirit, methods, languages and tools, but focuses less on structure and more on the description of algorithms and data structures.

Software development is a human activity and as a result constantly faces the problem of human error. A fundamental task of software development is “**V&V**”, which stands for “verification and validation”2. The difference between the two tasks is that verification is internal, devoted to assessing that the software is of good quality, and validation is external, devoted to assessing that it meets its specification (in particular, that implementation meets requirements). That difference is often expressed more vividly as “checking that the system does things right” versus “checking that it does the right things”. V&V is applicable to all software products, including requirements and design, but its most visible application is to code (program texts), for which it uses two classes of techniques: *dynamic*, requiring execution of the program, and *static*, working on the sole basis of the program text. Dynamic V&V is also called testing and consists of running the program on example inputs and checking the effect against expected outcomes (defined in advance). Static techniques include:

- Static analysis, which analyzes the program text — or (in “model checking” and “abstract interpretation”) an automatically simplified version of it — to spot potential violations of specified correctness rules, for example, arithmetic overflow or null-pointer dereferencing.
- Program proving, which mathematically ascertains the conformance of the program to a full specification, using special theorem-proving software tools.

Testing is by far the most widely used V&V technique for programs, but can only exercise a minuscule subset of cases and hence is mostly useful to find errors (rather than to guarantee the absence of errors). Program proving is far more ambitious, but still difficult to apply to mainstream program development, in particular because it requires writing a mathematical specification of the intent of very program element. Static analysis does not demand such a specification effort and is increasingly used as a more systematic alternative (or complement) to testing.

The next major task is **deployment**, which consists of making the system available for operational use. For a traditional program, deployment can be trivial (compile and link the code to produce an executable version), but for complex systems it is an engineering effort of its own, as in the case of an automatic-teller-machine system which must be deployed in many different locations with many different versions and under

[2] Often called just “verification”, but this discussion uses the more general and accurate term.

strict security requirements. In some cases, deployment does not even involve transferring any executable program to customers: software is increasingly being deployed “on the cloud”, meaning installed on Web servers and made available to its users through their Web browsers.

Finally, **maintenance** denotes all activities that occur after construction and deployment. The main components of maintenance are:

- Late V&V: finding and correcting errors that were not found and corrected prior to deployment, but come to light during operation of the system.
- Extensions: updating the software to account for users’ criticism and suggestions, new user goals, and changes in the environment.

## 3.2 Lifecycle

Traditional software engineering presentations typically described and prescribed software development in terms of “lifecycle models”, which define specific orderings of the tasks discussed in the preceding subsection, or some close variants.

The starting point for such discussions is generally the “waterfall model”, a strictly sequential ordering: feasibility study, then requirements, then design and so on. Since this model is too rigid to be applicable in practice, its main use is as both a foil (an easy target for criticism, serving as a prelude for advocacy of other models) and as a pedagogical device to present the above tasks.

A variant of the waterfall model is the “V-model”, which emphasizes the symmetry between construction activities (first branch of the V) and V&V activities (second branch) at different levels: unit testing corresponds to implementation, integration testing corresponds to design, and acceptance testing corresponds to requirements. Another variant is the “spiral model”, which makes the waterfall more flexible by applying a simplified version of it to successive *prototypes* of the system, each more refined than the previous one and building on the lessons learned from it; the first one of these prototypes to be judged good enough will be the one deployed. The author has used and described the “cluster model”, which applies a mini-waterfall to successive layers of the system, beginning with the most fundamental ones.

**Agile methods**, which have increasingly permeated the software industry since the early 2000s, use a lifecycle model divided not into tasks but into successive time slots or “sprints”, typically of a few weeks. In these approaches, the emphasis is on deadlines at the expense (if one has to choose) of functionality: if at the end of a sprint some of the planned functions have not been implemented, the sprint’s deadline never gets extended but, after suitable discussion, the functions get either moved to the next sprint or altogether removed from the project’s goals. Agile methods also characterize themselves by emphasizing two kinds of project deliverables, code and tests, and the associated tasks, implementation and V&V, over others such as requirements (typically handled through simple “user stories” describing units of interaction with the system) and design.

## 3.3 Modularization techniques

The presentation of design in subsection 3.1 pointed out the challenge of "decomposing systems into modules". Given the size of some of the programs it produces, software engineering has had to develop unique techniques for multi-level structuring of complex systems. They include (among others):

- **Data abstraction**, also known as abstract data types and *object-oriented* decomposition: the idea of decomposing systems not around their functions (operations) but around the types of objects they manipulate, also known as *classes*, each function becoming attached to the class to which it most directly relates. The notion of class unifies the dynamic concept of *type* (of objects) with the static notion of *module* (of a system).
- **Information hiding**, which directs the designer of any module (for example a class) to distinguish drastically and explicitly between properties that are relevant inside the class only, and properties made available, or *exported*, to other classes. This technique, also known as *encapsulation*, is critical to supporting the goal of module separation mentioned earlier, avoiding the "chain reactions" of changes in many modules that would otherwise occur whenever a module needs to be changed as part of the normal process of system evolution.
- **Inheritance**, which supports the organization of classes (or other kinds of modules) into taxonomies, grouping common elements at higher levels of a taxonomy so that the inheriting classes can use them without having to repeat their description. Inheritance is applicable to programs, for which it enables supplementary modularization techniques of *polymorphism* and *dynamic binding*, but also to other artifacts such as designs and requirements.
- **Genericity**, which allows classes to be parameterized and hence to lend themselves to different variants.

While originally invented for software and more particularly for programs, these mechanisms are general techniques for describing and building complex systems of any kind. They are an example of software-originated concepts that have a potential epistemological application to many other disciplines.

## 3.4 Size and exactness

A unique characteristic of software is its combined requirement for complexity and precision.

Human systems, such as a city, are *complex*, but tolerate many imperfections. (While you are reading this paragraph, many things in the closest large city are not right, such as traffic lights going out of order, accidents happening, people engaging in prohibited actions; but the city as a system is still functioning acceptably.)

Mathematical theories are *precise*, and so are non-software engineering systems, built on the basis of mathematical analysis; but their complexity typically remains far below that of ambitious software systems.

The complexity of such software systems is in a league with that of large human systems. Unlike them, however, software systems have very little tolerance for imprecision. Replacing a "+" by a "—" in one instruction (among millions) of an operating system's source code, or just one bit (among billions) of its executable version, may result in nothing functioning any more. Software construction is a harsh endeavor in which every detail must be right and the slightest error can cause havoc.

Everything in software engineering — all the techniques of requirements, design, implementation, V&V and other challenges of the discipline — is part of this attempt to reconcile the goals of complexity and precision.

## 4 Across two centuries: some fundamental advances

Software engineering 2020 differs from software engineering 1960 as a result both of changes in the environment in which it operates, particularly hardware and networks, and of its own intrinsic developments. Section 4.1 briefly summarizes changes in the context; the following subsections cover important evolutions in software engineering itself (development techniques in 4.2, management techniques in 4.3, and software engineering research in 4.4, with some forays into the future of the field in 4.5).

### 4.1 The evolution of software engineering's external context

The external factors are clear:

- The exponential growth of *computing power* mentioned at the beginning of this article, and the resulting growing ambition of software systems.
- The "*marriage of computing and telecommunications*" explained in a 1977 French report (Nora-Minc). Early software engineering treated computers as the name implies: computing devices, with some input and output. Today's computers are nodes in a network, and their computing functions are inseparable from their communication functions.
- As crucial examples of this evolution, the ubiquity of *the Internet, the World-Wide Web and cloud computing*. These developments, game changers for society, have raised the stakes for software development, in particular by making *information security* one of the dominant concerns. They brought in a new slate of technologies, from blockchain (for distributed trust) to containers (for application isolation).

### 4.2 Key developments in software construction

Here are some of the key concepts that took hold in the last half-century in techniques for designing and implementing programs.

**Structured programming** started in the late sixties and brought in the realization that programming is a demanding intellectual activity demanding discipline and reliance on mathematical reasoning. Some of the basic ideas, particularly the shunning of direct "goto" instructions, have become widely accepted (although the "goto" is not far away from the “break” and “return” instructions of many modern languages). The more ambitious goals of the creators of structured programming, particularly the use of mathematical correctness arguments, have still not become mainstream.

**Programming languages** have become more sophisticated, in particular through their abstraction and modularization mechanisms sketched earlier, but also through the growing reliance on the notion of *type* to frame the semantics of programs. (It should be noted that the recent popularity of the Python programming languages departs from this multi-decade trend, favoring instead the comfort of non-expert developers through “quick and dirty” techniques. But such forms of development do not really qualify as software engineering.)

**Object-oriented programming**, mentioned in 3.3, rapidly conquered much of the software development field starting in the late eighties, providing the engine that enabled software development routinely to tackle much more ambitious developments than ever before (which would not have been possible without the structuring mechanisms of OO technology, particularly classes and inheritance).

The **design pattern** movement, coming out in the nineties and building on the basic concepts of object orientation, brought software design to a new level of professionalism by identifying a number of fundamental architectural schemes that proved at the same time widely applicable, efficient, better than naïve solutions to the underlying problems, and eminently teachable.

**Formal methods** are the application of mathematical techniques to the specification and V&V of systems. They underlie, for example, the discipline of *program proving* mentioned in 3.1. As noted in section 1, formal methods give software engineering the justification for the “engineering” part of its name, which, in other engineering disciplines, implies the use of mathematics. Formal methods are still a minority phenomenon in software development, but play an important role in certain areas where correctness and security are essential. Many of the basic concepts have been known since a few years after IFIP’s creation, but patient work over the following decades made them step-by-step more practical, leading in particular to the construction of program-proving tools that can handle ever more ambitious practical systems.

### 4.3 Key developments in software management

Many changes have also occurred in the way we organize software projects.

The **open-source software** movement, initially a militant initiative to counter the dominance of commercial software, has become less controversial in recent years and contributed enormously (along with its nemesis) to the progress of the field. While the details of open-source legal licenses vary, the general idea is that the resulting software can easily be incorporated into new developments. This philosophy has spurred countless developers to provide the world with open-source products, repeatedly building on each other and providing for example a large part of today’s Internet and Web infrastructure, as well as an operating system (Linux) that runs many of the world’s computers and (in adapted form) phones and other devices, and prompted the development of a profitable industry of its own. Open-source development is often collaborative and distributed, leading from a software engineering perspective to the development of many new techniques, tools and repositories (such as the wildly popular GitHub) for multiple-person, multiple-site, multiple-target software construction.

**Agile methods**, already mentioned in 3.2, have had a profound effect on the practice of software construction by departing from the rigid project management schemes propounded by textbooks of yore and offering instead a flexible development model based on the primacy of code and tests (over requirements and design), the dominant role of the development team, the downplaying of traditional manager roles, a close relationship with stakeholders and business needs, and the reliance on short development iterations (sprints) observing strict time limits. Not all the consequences have been good (the agile rejection of upfront requirements and design documents, in particular, can have a detrimental effect on project success), but overall agile methods have brought a new level of excitement to software development and made teams far more reactive to true user needs.

**Configuration management** has developed as a fundamental management technique for software projects large and small. The complexity of software development has several dimensions; in "space", projects include myriad *components*; in time, each of these components can undergo many *transformations* during the life of a project; in a third dimension, many developers may independently perform *changes* to the components. Configuration management errors (such as combining versions of two modules at incompatible stages of their respective evolution) can cause disaster. Modern configuration management tools enable teams to avoid these mistakes and keep the evolution of systems and their components under control. Here too the complexity to be handled lies beyond what one encounters in other fields of engineering.

**DevOps** is a new paradigm of software development made necessary in particular by the frantic growth of Web and cloud applications. The discussion of software tasks in subsection 3.1 presented *deployment* as separate from *development* tasks (requirements, design, implementation). If a system is deployed on the Web — think for example of a search engine — the classical paradigm of working on new versions to be delivered every few months or years no longer applies; it would be unthinkable to force users to stop the previous version, install the new one and start again. Instead, usage never stops, and new versions must be deployed while this usage is proceeding, with most users not even noticing at that time (they might only notice, over time, that the service progressively improves). The term "DevOps" covers this scheme of interwoven development ("Dev"), deployment and operation ("Ops"), and raises fascinating new challenges for software engineering.

## 4.4 Key developments in software engineering research

Software engineering is not only an important applied activity but also a vibrant research field, with numerous journals, conferences (the most famous one, held yearly, goes back to 1975), PhD theses, prestigious IFIP working groups, and all the other trappings of an independent scientific discipline.

Software engineering research falls into four broad categories:

- **Conceptual**: propose new ideas or methods.
- **Constructive**: develop new tools, languages and other artifacts.
- **Analytical**: develop mechanisms for assessing artifacts and their quality.
- **Empirical**: process software artifacts using quantitative methods to derive general results.

In considering the evolution in recent decades, the most remarkable phenomenon has been the spectacular growth of the last category.

Empirical software engineering has come of age as a result of the growth of available subject material, starting with large software repositories (mentioned earlier in the context of open-source software, but also including commercial projects). Projects such as Linux have associated development databases going back many decades and containing (in addition, of course, to large amounts of source code, tests and other artifacts) the record of hundreds of thousands or even millions of individual code contributions and changes, as well as bug[3] reports and bug fixes. This material provides researchers with a fascinating basis to study the process of programming and all the technical and human factors that can affect a project.

In other words, empirical software engineering treats software artifacts the way natural sciences treat the targets of their study (be they from the inanimate or living world): as objects worthy of systematic quantitative study, nowadays using the most advanced techniques of "big-data" analysis, in particular data science and machine learning.

This form of research (so successful that it has come to dominate the field at the sometimes-regrettable expense of other kinds of research mentioned above) has led to many insights on software processes. It makes the majority of software engineering research publications in 2020 very different from what one finds in conference proceedings of 1980, 1990 or even 2000.

One of the consequences of this new focus has been a change of the kind of *mathematics* used for software engineering. Aside from the numerical techniques needed in the early days, math for computer science has traditionally involved logic and combinatorics ("discrete mathematics"). Big data and machine learning imply a shift to linear algebra and statistics as the main mathematical tools.

## 4.5 What next?

Aside from the obvious predictions, such as the ever increasing influence of machine-learning techniques, one may venture that software engineering will probably become *less* mainstream than today.

In the early days, even after the personal computing revolution started, software engineering occupied a special niche, reserved for large projects, mostly in the government and aerospace areas. It did not affect much the practices of developers in more mundane application domains. (At the 1987 main international conference in the field,

[3] "Bug" being, of course, the colloquial term for a software error.

the author suggested, in the closing program committee meeting, that the next conference should invite some of the famous leaders of the PC software industry as keynote speakers. The reaction was that the community had nothing to learn from such amateur bit-players.) The situation then changed drastically: from the nineties on, many of the key software engineering ideas gained influencein the software development community at large, in the same way that techniques first tried out in Formula-1 racing find their place, a technology generation or two later, in the design of mass-market cars.

The chasm is coming back. Perhaps as a consequence of the democratization of programming (the basics of which are increasingly taught nowadays in secondary or even primary school), there has been a regression of the influence of software engineering principles on the mainstream development. The spread of programming languages for quick-and-dirty coding, mentioned in section 4.2, is an example of this regression. The impression here is that as long as your program produces any results at all, no one is going to look into the sausage-making. Clearly, such an approach does not transpose to mission-critical developments. But it is increasingly the dominant one today in most software development: "anything goes". Software engineering proper gets confined to the advanced professional developments, the ones that we cannot afford to get wrong.

While regrettable, this trend of separating mainstream development from professional software engineering is probably going to continue. Being able to program a computer is no longer the mark of a sought-after expert. The true difference is between casual developers, who can somehow put together ("hack") some code, with little guarantee of quality, and actual software engineers, who know and apply the principles and practices of software engineering characterized — as defined in this article — by a fundamental focus on quality.

While professional software engineers may lament the lack of quality concerns in much of today's developments, they can take pride in noticing that it is, for a part, a sign of the very success of their field. A sloppy programmer in 1960 would have produced lamentable, unusable programs. Because of the tremendous development of software engineering languages and tools since then, even a sloppy programmer today can produce acceptable code. The reason is that so much of the work actually gets done bythe underlying layers (operating system, libraries of reusable components to take care of numerous aspects from user interfaces to numerical computation, compilers, development environments, debugging tools, configuration management tools, repositories...) that you can produce a decently working system by just throwing in a few elements of your own, whether properly software-engineered or not, on the top of that mighty technology stack. This ability to let non-professionals benefit from the hard work of the professionals is a sign of the field's growing maturity.

## 5 Conclusion

Modern software systems are among the most complex and ambitious systems of any sort that humankind has ever attempted to build. Software engineering, some of whose concepts and techniques have been sketched in the preceding sections, provides a way to achieve these ambitions and produce systems that will work to the satisfaction of their intended beneficiaries — people and organizations. These concepts and techniques, patiently developed over six decades, provide the closest the human mind has ever produced to a science and engineering of complexity.

The very success of the discipline puts an ever-heavier burden on the shoulders of software engineering professionals, who must constantly bear in mind that their programs are not just elegant intellectual exercises using the best algorithms, data structures and software engineering techniques, but tools to address society's goals. Society relies ever more heavily on software to achieve these goals, forcing software engineers to confront numerous ethical issues (exacerbated, in particular, by the growing use of machine learning, which reproduces existing patterns rather than renewing them) and making ever more central the role of quality, in all its facets, in the pursuit of true software engineering.

Meeting these challenges is hard, but as anyone who has genuinely tried to tackle them can testify, there does not on earth exist a more fascinating pursuit.