1

Software Engineering: A Discipline Like No Other

On the face of it, software engineering sounds like an engineering discipline among others, such as chemical engineering, mechanical engineering, civil engineering, and electrical engineering. We will explore, in this chapter, in what way and to what extent software engineering differs from other engineering disciplines.

1.1 A YOUNG, RESTLESS DISCIPLINE

Civil engineering and mechanical engineering date back to antiquity or before, as one can see from various sites (buildings, road networks, utility infrastructures, etc.) around the Mediterranean basin. Chemical engineering (Lavoisier and others) and electrical engineering (Franklin and others) can be traced back to the eighteenth century. Nuclear engineering (Pierre and Marie Curie) emerged at the turn of the twentieth century and industrial engineering emerged around the time of the Second World War, with issues of logistics. By contrast, software engineering is a comparatively young discipline, emerging as it did in the second half of the twentieth century. The brief history of this discipline can be divided into five broad eras, lasting approximately one decade each, which are as follows:

- *The Sixties: The Era of Pioneers.* This era marks the first time that practitioners and researchers came face to face with the complexities, paradoxes, and anomalies of software engineering. Software projects of this era were ventures into unchartered territory, characterized by high levels of risk, unpredictable outcomes, and massive cost and schedule overruns. The programming languages that were dominant in this era are assembler, Fortran, Cobol, and (in academia) Algol.

Software Testing: Concepts and Operations, First Edition. Ali Mili and Fairouz Tchier.
© 2015 John Wiley & Sons, Inc. Published 2015 by John Wiley & Sons, Inc.

- *The Seventies: Structured Software Engineering.* This era is characterized by the general belief that software engineering problems are of a technical nature and that if we evolved techniques for software specification, design, and verification to control complexity, all software engineering problems would be resolved. Given that structure is our main intellectual tool for dealing with complexity, this era has seen the emergence of a wide range of structured techniques, including structured programming, structured design, structured analysis, structured specifications, etc. The programming languages that were dominant in this era are C and (in academia) Pascal.
- *The Eighties: Knowledge-Based Software Engineering.* This era is characterized by the realization that software engineering problems are of a managerial and organizational nature more than a technical nature. This realization was concurrent with the emergence of the Fifth Generation Computing initiative, which started in Japan and spread across the globe (the United States, Europe, Canada), and was focused on thinking machines designed with extensive use of artificial intelligence techniques. This general approach permeated the discipline of software engineering with the emergence of knowledge-based software engineering techniques. The programming languages that were dominant in this era are Prolog, Scheme/Lisp, and Ada.
- *The Nineties: Reuse-Based Software Engineering.* As it became increasingly clear that fifth-generation computing was not delivering on its promise, and worldwide fifth-generation initiatives were fading, software researchers and practitioners turned their attention to reuse as a possible savior of the discipline. Software engineering is, after all, the only discipline where reuse is not an integral part of the routine engineering process. It was felt that if only software engineers had large databases of reusable software components readily available, the industry would achieve great gains in productivity, quality, time to market, and reduced process risk. This evolution was concurrent with the emergence of object-oriented programming, which supports a bottom–up design discipline that facilitates product reuse. The programming languages that were dominant in this era are C, C++, Eiffel, and Smalltalk.
- *The First Decade of the Millennium: Lightweight Software Engineering.* While software reuse is not practical as a general paradigm in software engineering, it is feasible in limited application domains, giving rise to *product line engineering*. Other attributes of this era include Java programming, with its focus on web applications; agile programming, with its focus on rapid and flexible response to change; and component-based software engineering, with its focus on software architecture and software composition. The programming languages that were dominant in this era are Java, C++, and (in academia) Python.

Perhaps as result of this young and eventful history, the discipline of software engineering is characterized by a number of paradoxes and counter-intuitive properties, which we explore in this chapter.

1.2 AN INDUSTRY UNDER STRESS

Nowadays, software runs all aspects of modern life and accounts for a large and increasing share of the world economy. This trend started slowly with the advent of computing in the middle of the twentieth century and was further precipitated by the emergence of the World Wide Web at the end of the twentieth and the beginning of the twenty-first century. This phenomenon has spawned a great demand for software products and services and generated a market pressure that the software industry takes great pains to cater to.

Many fields of science and engineering (such as bioinformatics, medical informatics, weather forecasting, and modeling and simulation) are so dependent on software that they can almost be considered as mere applications of software engineering. Also, it is possible to observe that many computer science curricula are slowly inching toward more software engineering contents at the expense of traditional theoretical material, which may be perceived as less and less relevant to today's job market. Some engineering colleges are preempting the trend by starting software engineering degrees in computer science departments or by starting complete software engineering departments alongside traditional computer science departments.

Concurrent with a widening demand for software to serve ever-broader needs, we are also witnessing higher and higher expectations in terms of product quality. As software takes on ever more vital functions in life-critical and mission-critical applications and in applications that carry massive financial stakes, it becomes increasingly important to ensure that software products fulfill their function with a high degree of dependability. This requires that we deploy a wide range of techniques, including the following:

- *Process controls*, ensuring that software products are developed and evolved according to certified, mature processes.
- *Product controls*, ensuring that software products meet quality standards commensurate with their application domain requirements; this is achieved by a combination of techniques, including static analysis, dynamic testing, reliability estimation, fault tolerance, etc.

In summary, it is fair to argue that the software industry is under massive stress to deliver both quantity and quality; as we discuss in subsequent sections, this is both difficult and expensive.

1.3 LARGE, COMPLEX PRODUCTS

The demand for complex hardware/software systems has increased more rapidly than the ability to design, implement, test and maintain them.
Michael Lyu, *Handbook of Software Reliability Engineering*, 1996

Not only is it critical for us to build software products that are of high quality, it is also very difficult, due to their size and complexity. When it was built in the mid-60s, the IBM operating system OS360 (©IBM Corporation), with a million lines of code and a price tag of 500 million dollars, was considered as the most complex human artifact ever produced up to then. This size was subsequently dwarfed by Microsoft's Windows operating systems (©Microsoft): The 1993 version (Windows NT 3.1) is estimated to be 5 millions lines of code, whereas the 2003 version (Windows Server 2003) is estimated to be 50 million lines of code. Completing projects of this kind of size is not only a major engineering undertaking but also a major organizational challenge; it is estimated that the production of the Windows Server 2003 involved 2000 software personnel (programmers, analysts, engineers) for development and 2400 software personnel for software testing.

Another example of software size growth is given by NASA's flight software. A study published in 2009 by NASA's Jet Propulsion laboratory under the title *NASA Study on Flight Software Complexity* (Jet Propulsion Laboratory, 2009) plots the evolution of flight software size of the various human and robotic space programs from 1968 to 2005. Both series (flight software for human missions and flight software for robotic mission) show a near-perfect linear evolution through the years, except that they are plotted on a logarithmic scale for size, meaning in effect that flight software size grows exponentially from year to year. Hence for human missions, flight software grows from 8.5 kilo lines of code (KLOC) for the Apollo program in 1968 to 470 KLOC for the space shuttle program in 1980 to 1.5 million lines of code (MLOC) for the international space station in 1989. For robotic missions, software size grows from 30 line of code (LOC) for the Mariner-6 mission in 1968 to 3 KLOC for Voyager in 1977 to 8 KLOC for Galileo in 1989 to 349 KLOC for DS1 (Deep Space 1) in 1999 to 545 KLOC for MRO (Mars Reconnaissance Orbiter) in 2005. The same Jet Propulsion Laboratory (JPL) report describes the recent evolution of military avionics software in the following terms: between 1960 and 2000, the percentage of flight control functionality that is delegated to software jumped from 8 to 80%, leading to an increase in size from one generation of aircrafts to another; hence it went from 1000 lines of code for the F-4A to 1.7 million lines of code for the F-22 to 5.7 million lines of code for the F-35 Joint Strike Fighter. The authors of the report argue that the increase in the size and complexity of flight software stems from software serving as a '*complexity sponge*,' whereby complexity migrates from other parts of the system to software, on account of its flexibility and its adaptability.

A panel convened by the Software Engineering Institute (www.sei.cmu.edu) in 2005–2006 to analyze software systems of the future and draw a research agenda to manage such systems estimates that future software systems are expected to have sizes up to a billion lines of code. Along with this dry measure of size, such systems will be large in terms of other dimensions, such as (www.sei.cmu.edu/uls/) the amount of data stored, accessed, manipulated, and refined; the number of connections and interdependencies; the number of hardware elements; the number of computational elements; the number of system purposes and user perception of these purposes; the number of routine processes, interactions, and emergent behaviors; the number of overlapping policy domains and enforceable mechanisms; and the number of parties involved in the operation of the system (developers, maintainers, end users, stakeholders, etc.).

Size changes everything: such systems (referred to as ultra-large–Scale (ULS) systems) challenge all our knowledge and assumptions about software and are estimated to have a number of distinguishing features, such as the following:

- Decentralization in fundamental dimensions, such as decentralized development, decentralized evolution, and decentralized operation.
- Conflicting, unknown, and diverse requirements: Whereas the traditional view in software engineering is that requirements must be analyzed, compiled, and specified prior to software design and development, the view taken by the ULS approach is that at no time can we claim that all relevant requirements have been collected and specified.
- Continuous evolution and deployment: Whereas the traditional view of software engineering is that a software product proceeds sequentially through successive phases of development, then maintenance, then phase out, ULS systems are developed, evolved, and deployed concurrently (made up of parts that are at different stages in their evolutionary process).
- Heterogeneous, inconsistent, changing elements: Whereas a traditional software product is developed as a cohesive monolithic system by a development team, ULS systems emerge as the aggregate of many components, which may have evolved independently, using different paradigms and different technologies, by different teams, and from different stakeholder classes. Also, different components of the system are expected to evolve relatively independently.
- Deep erosion of the people-system boundary: Whereas traditional systems are defined in terms of a distinct boundary that separates them from the outside world, ULS systems are envisioned to include human users as an integral part so that when a user interacts with a ULS system, she/he may be engaging human actors along with system behavior.
- Failure is normal and frequent: Whereas in traditional software systems we think of failures as exceptional events and consider that failure avoidance is contingent upon fault removal, in ULS systems, we take a broader view of successful (failure-free) operation, which does not exclude the presence of faults but makes provisions for system redundancy and requirements nondeterminacy to make up for the presence of faults.

1.4 EXPENSIVE PRODUCTS

Not only are software products very large and complex, they are also very expensive to produce. Of course, if a product is large, one expects it to be costly, but what is surprising is that the *unitary* cost of software, that is, the cost per LOC, does, itself, increase with size. Whereas any programmer one asks may say that they can produce a hundred lines of code in a day or more, a more realistic figure, across all areas of software development, is closer to about 10 lines of code per day, or about 200 lines of code per person-month. This figure includes all costs that are spent producing software, including the cost of all phases of the software lifecycle, from requirements analysis and specification to software testing. If we assume the cost of a person-month to be 20,000 dollars (in salary, fringe benefits, and related expenses), this amounts to about \$100 per LOC. If, for the sake of argument, we apply Boehm's COnstructive COst MOdel (COCOMO) cost estimation model to a bespoke (custom-tailored) software project of size 500,000 source lines of code developed in embedded mode (the hardest/most costly development mode), we find 80 source lines of code per person-month.

In most other engineering disciplines, one way to mitigate costs is to use economies of scale, that is, to produce in such a large volume as to lower the unitary cost. Economies of scale are possible because in most engineering disciplines, the production process requires an initial up-front cost that is all the better amortized as the volume of production increases. The same process applies in software engineering: If we invest resources to acquire software tools, to train software professionals, or to set up a programming environment, then the more software we produce the better our investment is amortized. But in software we are also dealing with a phenomenon of diseconomy of scale: the more software we produce within a single product, the more interdependencies we create between the components of the product so that the unitary cost (per LOC) of large software products is larger than that of smaller products. This phenomenon of diseconomy of scale overrides the traditional economy of scale (that comes from amortizing up-front investments); the net result is a diseconomy of scale, which is all the more acute that the software product is larger or more complex; see Figure 1.1.

Many of these costs are mitigated nowadays by the use of a variety of coarse-grained software development methods, which proceed to build software by composing existing components, rather than by painstakingly writing code from scratch line

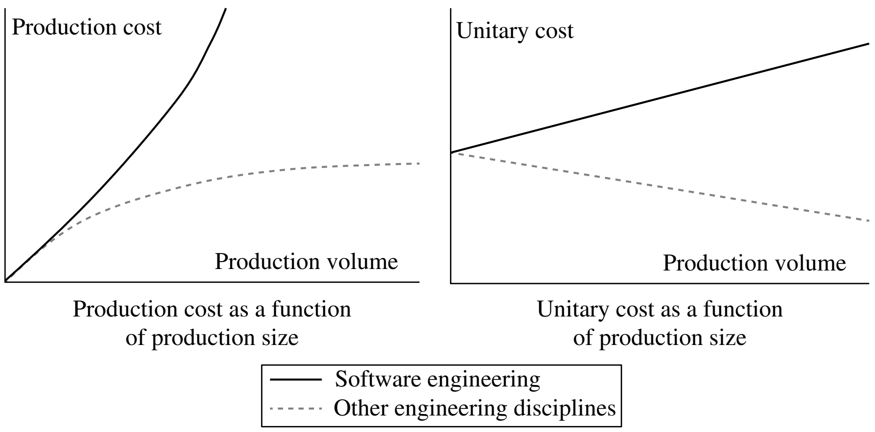

Figure 1.1 *Diseconomies of scale in software engineering.*

by line. Another trend that is emerging recently to address software cost and quality is the use of so-called Agile methodologies. These methodologies control the costs and risks of traditional lifecycles by following an iterative, incremental, flexible lifecycle, where the user participates actively in the specification and development of successive versions of the targeted software product.

1.5 ABSENCE OF REUSE PRACTICE

In the absence of economies of scale, one would hope to control costs by a routine discipline of reuse; in the case of software, it turns out that reuse is also very difficult to achieve on a routine basis. In any engineering discipline, reuse is made possible by the existence of a standard product architecture that is shared between the producer and the consumer of reusable assets: for example, automobiles have had a basic architecture that has not changed for over a century; all cars have a chassis, four wheels, an engine, a battery, a transmission, a cab, a steering column, a braking system, a horn, an exhaust system, shock absorbers, etc. Thanks to this architecture, the design of a new car is relatively straightforward and is driven primarily by design and marketing considerations; the designer of a new model does not have to reinvent a car from scratch and can depend on a broad market of companies that provide standard components, such as batteries, tires, and spare parts. The standard architecture of a car dictates market structure and creates great efficiencies in the production and maintenance of a car.

Unfortunately, no standard architecture exists in software products; this explains, to a large extent, why the expectations that software engineering researchers and practitioners pinned on a discipline of software reuse never fully materialized. Several software reuse initiatives were launched in the last decade of the last century, making available a wide range of software products and sophisticated search and assessment algorithms; but they were unsuccessful because software reuse requires not only functional matching between the available components and the requirements of the user but also architectural matching, which was often lacking. The absence of a standard architecture of software products also explains why software product lines have achieved some degree of success: product line engineering is a form of software reuse that is practiced in the context of a narrow application domain, in which it is possible to define and enforce a reference architecture. As an example, if we define a product line of e-commerce systems, we may want to define the reference architecture as being composed of the following components: a web front-end; a shopping cart component; an order-processing component; a banking component; a marketing and recommendations component; a network interface; and a database interface.

1.6 FAULT-PRONE DESIGNS

In other engineering disciplines, the presence of a standard product architecture, the availability of usable product components, the availability of compiled engineering knowledge, and the application of mandated safety requirements all contribute to reducing the design space of a product so as to make it manageable. The design of an engineering product (e.g., a bridge, a road, or a car) within this limited design space is a fairly straightforward operation that proceeds from requirements to finished product in a systematic, predictable manner.

In software engineering, the situation is significantly more complex, for several reasons, which are as follows:

- There is no standard software architecture, except perhaps for some vague architectures of broad families of software products, such as data-processing applications, transaction-processing applications, event-processing applications, and language-processing applications.
- There is little or no availability of software reusable assets, in the traditional sense of engineering assets that can be used to compose software products; the only assets that may be used widely across the industry are small assets (such as abstract data types (ADTs)) that deliver limited gains in terms of reduced lifecycle costs or reduced process risk.
- There is little software engineering knowledge that may be used across applications in the same way that engineering knowledge is reused in complied form across products in other engineering disciplines.
- Software specifications are very complex artifacts that typically involve vast amounts of detailed functional information; the breadth of the specification space precludes the ability to organize the design space in a systematic manner.

Because the design space of software products is so vast, software design is significantly more error prone than design in other engineering disciplines.

1.7 PARADOXICAL ECONOMICS

While technology can change quickly, getting your people to change takes a great deal longer. That is why the people-intensive job of developing software has had essentially the same problems for over 40 years.
Watts Humphrey, *Winning with Software: An Executive Strategy*, 2001

1.7.1 A Labor-Intensive Industry

If we consider the cost of an automobile, for example, and ponder the question of what percentage of this cost is due to the design process and what percentage is due to manufacturing, we find that most of the cost (more than 99%, perhaps) is due to manufacturing. Typically, by the time one buys a car, the effort that went into designing the new model has long since been amortized by the number of cars sold; what one is paying for is all the raw materials and the processing that went into manufacturing the car. By contrast, when one is buying a software product, one is paying

TABLE 1.1 Lifecycle cost distribution: design versus manufacturing

	Software engineering, %	Other engineering, %
Design	>99	<1
Manufacturing	<1	>99

essentially for the design effort, as there are no manufacturing costs to speak of (loading compact disks or downloading program files). Table 1.1 shows, summarily, how the cost of a software product differs from the cost of another engineering product in terms of distribution between design and manufacturing.

1.7.2 Absence of Automation

The labor-intensive nature of software engineering has an immediate impact on the potential to automate software engineering processes. In all engineering processes, one can achieve savings in manufacturing by automating the manufacturing process or at least streamlining it, as in assembly lines. This is possible because manufacturing follows a simple, systematic process that requires little or no creativity. By contrast, design cannot be automated because it requires creativity, artistic appreciation, aesthetic sense, and so on. Automating the manufacturing process has an impact in traditional engineering disciplines because it helps reduce a cost factor that accounts for more than 99% of production costs; but it has no impact in software engineering because it affects less than 1% of production costs. Hence the automated development of software products is virtually impossible in general.

The only exception to this general rule is the development of applications within a limited application domain, where many of the design decisions may be taken a priori when the automated tool is developed and hardwired into the operation of the tool. One of the most successful areas of automated software development is compiler construction, where it is possible (thanks to several decades of intensive research) to produce compilers automatically, from a syntactic definition of the source language and relevant semantic definitions of its statements. Not surprisingly, this is a very narrow application domain.

1.7.3 Limited Quality Control

The lack of automation and hence the absence of process control make it difficult to control product quality. Whereas in traditional engineering disciplines, the production process is a systematic, repeatable process, one can control quality analytically by certifying the process or empirically by statistical observation. Because the production of software proceeds through a creative process, neither approach is feasible, since the process is neither systematic nor repeatable. This shifts the control of product quality to product controls, such as static analysis, or dynamic program testing.

TABLE 1.2 Lifecycle cost distribution: development versus testing

	Software engineering, %	Other engineering, %
Development	≈50	>99
Testing	≈50	<1

1.7.4 Unbalanced Lifecycle Costs

In most other engineering disciplines, products are produced in large volume and are generally assumed to behave as expected; in software engineering, due to the foregoing discussion, such an assumption is unfounded, and the only way to ensure the quality of a software product is to subject that product to extensive analysis. This turns out to be an expensive proposition, in practice, and the source of another massive paradox in software engineering economics. Whereas testing (and more generally, verification and quality assurance) takes up a small percentage of the production cost of any engineering artifact, it accounts for a large percentage of the lifecycle cost of a software product. As a practical example, consider that the development of Windows Server 2003 (©Microsoft Corp.) was carried out by a team of 4400 software engineers, of whom 2000 formed the development team and a staggering 2400 formed the test team. More generally, testing accounts for around 50% of lifecycle costs, which is much higher than traditional manufacturing industries (where the likelihood of a defective product is so low as to make any significant amount of testing wasteful) (Table 1.2).

Good software engineering practice dictates that more effort ought to be spent on up-front specification and design activities and that such up-front investment enhances product quality and lessens the need for massive investment in *a posteriori* testing. While these practices appear to be promising, they have not been used sufficiently widely to make a tangible impact; so that software testing remains a major cost factor in software lifecycles.

1.7.5 Unbalanced Maintenance Costs

It is common to distinguish in software maintenance between several types of maintenance activity; the two most important types (in terms of cost) are as follows:

- Corrective maintenance, which aims to remove software faults
- Adaptive maintenance, which aims to adapt the software product to evolving requirements

Empirical studies show that adaptive maintenance accounts for the vast majority of maintenance costs. This contrasts with other engineering disciplines, where there is virtually no adaptive maintenance to speak of: it is not possible for a car buyer to return to the dealership to make her/his car more powerful, add seats to it, or make it more fuel-efficient. Hence, it is possible to distinguish between software products

TABLE 1.3 Maintenance cost distribution: corrective versus adaptive

	Software engineering, %	Other engineering, %
Corrective	≈20	>99
Adaptive	≈80	<1

TABLE 1.4 Corrective maintenance cost distribution: design versus wear and tear

	Software engineering, %	Other engineering, %
Design	≈100	1
Wear and tear	≈0	99

and other engineering products by the distribution of maintenance, as shown in Table 1.3.

While it is not realistic to expect a car dealership to change a car to meet different specifications, it is certainly their responsibility to repair if it no longer meets its original specifications. Another distinguishing feature arises when one considers corrective maintenance: Whereas in software products corrective maintenance consists in changing the design or implementation of the product, in other engineering disciplines products need (corrective) maintenance due to wear and tear (Table 1.4).

The only cases where a maintenance action on a brick-and-mortar product (e.g., a car) is of type *design* are cases where a manufacturer makes a product recall; these are sufficiently rare that they are usually newsworthy and are broadly advertised in public forums.

1.8 CHAPTER SUMMARY

This chapter introduces the discipline of software engineering with all its specific characteristics and paradoxes, contrasts it with more traditional engineering disciplines, and elucidates the role that software testing plays within this discipline.

1.9 BIBLIOGRAPHIC NOTES

For more information on the COCOMO cost model, consult (Boehm, 1981) or (Boehm et al., 2000); for more information on the JPL report on the evolution of avionics and space flight software, consult (Jet Propulsion Laboratory, 2009); for more information on the classification of software products into broad families of applications, consult (Somerville, 2004).