Chapter 2
Software Engineering

Abstract Software engineering is an engineering discipline that is focused on all aspects concerning the development of software-based systems. This chapter begins with an explanation of the contributions of software engineering to the issues related to requirements, discussing the possibility of adopting their methods on projects of other engineering disciplines. The chapter also characterises the software engineering, identifying and describing the fifteen knowledge areas of the SWEBOK guide. Additionally, the most relevant characteristics associated with the software are discussed. Finally, some of the most popular development process models are presented.

2.1 Contributions for Requirements Engineering

The advances and progresses that were made during the last 40 years in the software and information systems domain are, whatever the perspective, extraordinary. Despite some negative propaganda (for instance, the eternal software crisis or the non-event that was the *year 2000 bug* (Y2K bug), there is a huge number of success cases that changed human daily habits, in a significative or even drastic way. Nowadays, software is present not only in traditional personal computers or in highly-sophisticated supercomputers, used for scientific purposes or for executing governmental activities, but also in mobile devices, like cellular phones or tablets, or in devices responsible for routing in computer networks.

The existence of a knowledge area related to software engineering is due to the growing complexity of the developed systems and to economic factors that prompt software producers to try to optimise the processes. Therefore, they can gain a competitive advantage to be better positioned in the markets where they operate. All these issues have influenced the evolution of this area, which started to take shape at the end of the 1960 decade. There are of course problems associated with software development and many software projects are not ready on time and cost much more than was initially planned. It is due to the fact that those problems can occur that one needs to adopt an engineering approach.

J.M. Fernandes and R.J. Machado, *Requirements in Engineering Projects*,
Lecture Notes in Management and Industrial Engineering,
DOI 10.1007/978-3-319-18597-2_2

Software engineering is a discipline composed of theories, methods, approaches, and tools needed to construct software systems. Software engineering, like the other engineering branches, needs to base its activity on *quality*. This implies that everything that is related to engineering, including obviously the developed systems, must possess quality, in order to satisfy the client. In a simple way, quality is the compliance with the requirements. The main objective of software engineering is to develop software systems with quality, fulfilling the budget, meeting the deadlines, and satisfying the real needs of clients by solving their problems.

> "The function of good software is to make the complex appear to be simple."
> *Grady Booch (1955–), software engineer*

To cope with the growing complexity and diversity of real-world problems and to changes in requirements during the development process, the construction of systems in the software and information systems domain needs to adopt engineering approaches, to improve the quality of those systems. This does not imply that the system is for sure optimal, but that normally it possesses a good quality and is produced in a controlled way and limited by the available budget.

In this context, over the last years, the software engineering discipline has accumulated an extensive scientific body of knowledge related to the requirements and their problems. This reality results from the need to control the enormous uncertainty and the high number of risks normally associated with this domain, due namely to the intangible nature of the information technologies. Operationally, the success of projects in the software and information systems domain requires an equilibrium between the capacity of reading the surrounding reality (environment) and the ability to socio-technically act upon it. These demands refer to the relationship with the stakeholders and to the technical decisions associated with the project management.

Based on this track taken by the software engineering discipline, the dimension and importance of the scientific community that is focused on the subject of requirements has grown tremendously. This community adopted the 'requirements engineering' concept, understood as the set of activities that in the context of developing a system allows one to elicit, negotiate, and to document the functionalities and restrictions of that system.

In fact, the process of characterising a building, automobile, boat, or house contains different aspects than those that are relevant in the process of defining the requirements of a system in the software and information systems domain. However, it also includes many similar issues. The methods and the techniques discussed in this book result from the contributions made by the requirements engineering scientific community. In most cases, those methods and techniques are not exclusive for the software and information systems domain and can thus be applied in any engineering project, whatever branch or field.

2.2 Characterisation of the Discipline

Software engineering, as a discipline, can be defined in different ways. In a simple way, one can say that it corresponds to the application of the engineering principles to the software development process. This line of reasoning was used by Fritz Bauer, when in 1968, he defined it as being the utilisation of the basic engineering principles to obtain, in a economically-viable way, reliable software that runs efficiently in real computers.

Software engineering is focused on all aspects related to the development and utilisation of software systems, from the initial steps of specification until maintenance. There are two software engineering aspects that deserve to be highlighted. On the one hand, engineers must be able to manage the development and industrialisation of useful and efficient system. On the other hand, software engineering is not just centred in the technical aspects associated with the development, but includes also activities related to managing the development process.

> "Leonardo Da Vinci combined art and science and aesthetics and engineering. That kind of unity is needed once again."
>
> *Ben Shneiderman (1947–), computer scientist*

One can also define software engineering as a computer science field that tackles the construction of software systems whose complexity requires a team of software engineers to build it (Ghezzi et al. 1991, p. 1). Here, the emphasis is put on the system complexity and, as happens in all engineering branches and fields, whenever the scale of the systems is changed, different problems and challenges arise. Some software systems have a long lifecycle, which explains the reason why they go through several versions in order to make sure that they adapt to new realities. The systems developed within the scope of software engineering are also used by different types of users and may include gigantic volumes of information.

Small software programs, written and maintained by a single person (the so-called heroic programmer), are not generally considered sufficiently complex to demand the use of an engineering approach. It is relevant here to distinguish computer programming from software engineering, since unfortunately there are still many persons, some professionally connected to computing, that consider both activities to be the same. A *programmer* writes complete software programs. It is very difficult, even literally impossible, for a single programmer to dominate all the facets of a software system, since its complexity largely exceeds the human mind capacities (Booch et al. 2007, p. 8). It is very likely, according to Laplante (2007, p. 4), that a software engineer spends less than 10 % of his time in programming activities, using the remaining time for the other activities of the software engineering process. Regardless of the number being totally aligned with the reality, the relevant thing to understand here is its order of magnitude.

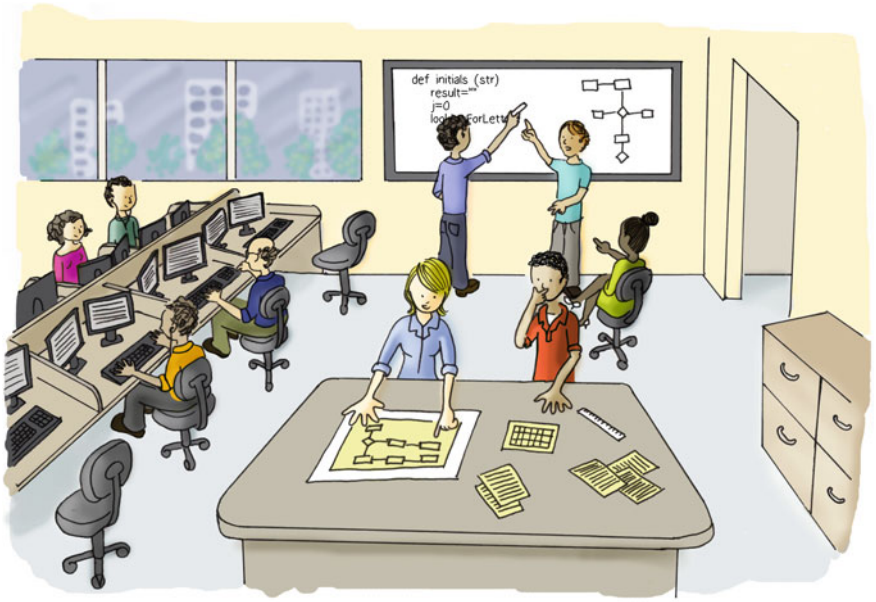

Illustration 2.1 A typical software engineering environment, with programmers, software architects, analysts, testers, and clients

The differentiation between software engineers and programmers is intimately related to the professional liability and accountability for acts of public trust in technologic interventions. It has been particularly difficult to convince the society and the professionals themselves that, in contexts of great complexity, the engagement of software engineers in the implementation phase is not justifiable to execute programming tasks. Software engineers should instead technically coordinate the work of the programmers. Metaphorically, software engineering is related to programming in the same way as civil engineering is associated with civil construction.

Software engineering includes also a specific management component that does not exist in programming. In a small project, with one or two programmers, the relevant aspects have essentially a technologic nature. In a longer project with a team composed of many members, it is indispensable to conduct management efforts, to plan and control the activities of the various professionals with differentiating roles, such as analysts, architects, programmers, and testers (Illustration 2.1).

In this book, **software engineering** is defined as the application of a systematic, disciplined and quantifiable approach in the context of the planning, development and exploration of software systems, that is, it is the application of engineering to the software domain. It is under this definition that appears the SWEBOK (software engineering body of knowledge) guide (Bourque et al. 1999; Abran et al. 2004), as an important reference to characterise the software engineering discipline. This guide is the result of an initiative jointly promoted by the IEEE Computer Society

Table 2.1 KAs of the SWEBOK

1	**S**oftware requirements	9	Software engineering models and methods
2	Software design	10	Software quality
3	Software construction	11	Software engineering professional practice
4	Software testing	12	Software engineering economics
5	Software maintenance	13	Computing foundations
6	Software configuration management	14	Mathematics foundations
7	Software engineering management	15	Engineering foundations
8	Software engineering process		

(IEEE-CS) and the Association for Computing Machinery (ACM). The SWEBOK was created to address the following objectives:

- to promote in the scientific community the existence of a coherent view of software engineering;
- to clarify how software engineering is related to other disciplines, such as computer science, computer engineering, project management, and mathematics;
- to characterise the scope and thematic contents of the software engineering discipline;
- to ease the access to the contents of the software engineering body of knowledge;
- to provide a reference for the definition of *curricula* and professional certifications in the software engineering discipline.

The SWEBOK guide structures the software engineering corpus according to the KAs shown in Table 2.1. KAs 11–15 were only introduced in third version of the guide (IEEE 2014).

Next, all the KAs are very briefly presented, to provide a generic idea of the subjects covered by software engineering. Thus, it is possible to relate requirements engineering (the main topic of this book) with the other activities carried out in the context of software engineering.

Most topics detailed in this book fall within the scope of KA1 (software requirements). This KA handles the elicitation, analysis, documentation, validation, and maintenance of software requirements. Requirements are considered as properties that the systems (still in project) may manifest later after development. The software requirements express the necessities and the restrictions that are put to a software system and that must be taken into account throughout its development. This KA is recognised as having primary importance for the software industry, due to the impact that its activities promote on stabilising and managing all the development process.

Software design (KA2) is the process where the architecture, components, interfaces, and other system (or its components) characteristics are defined. From the process perspective and within the scope of the software development lifecycle, software design is the activity in which the software requirements are handled with the purpose of producing a description of the internal structure and organisation of the

system. From the product perspective, the final result of the process should describe the system architecture (i.e., how it can be decomposed and organised into components), the interfaces between the components, and the components with a level of detail that permits their construction.

Software construction (KA3) represents the fundamental act associated with software engineering. It consists in implementing software in accordance with the requirements and that works correctly, through a combination of coding, validation, and testing activities. The implementation of software is intimately related to design, since the former must transform into code the architectures conceived and described by the latter. This transformation tends to be more and more automatic, since there are tasks perfectly repetitive and mechanistic. Thus, it is in the software implementation phase that the utilisation of tools is more critical, to free the engineers from the less creative and error-prone activities.

Software testing (KA4) constitutes a mandatory part of the software development. Simultaneously, it is an activity in which one evaluates the software system quality and enhances it through the identification of the defects and potential problems. Testing includes, for instance, the dynamic verification of the software system behaviour in comparison with the expected one, using a finite set of test cases, especially chosen to cover the most critical situations.

Software maintenance (KA5) consists in introducing changes in the software system, after it was deployed and brought into operation, in order to (1) improve the system, (2) correct defects, and (3) adapt the system to a new context. The maintenance phase copes with the defects, the technological modifications, and the user requirements evolution. It is recommended to prepare maintenance in advance during the development phases, to ease the tasks that compose it. Although software maintenance is an area of software engineering, it has received a smaller attention by the scientific community when compared, for example, with design and construction.

Maintenance can be reactive, when the intervention is dictated by defects observed in the system, or proactive, whenever the intervention is performed before detecting the defects. In another dimension, maintenance can be oriented towards correction, which means that one attempts to detect and to repair the defects, or oriented towards improvement, with the objective of enhancing the system to accommodate new requirements or contexts of utilisation. The IEEE 14764 standard employs these two dimensions, as illustrated in Table 2.2, to divide maintenance into four categories: preventive, corrective, perfective, and adaptive.

> "Maintenance typically consumes about 40 to 80 % of software costs. Therefore, it is probably the most important life cycle phase of software."
>
> *Robert L. Glass (1932–), software engineer*

Change is inevitable when developing systems, due to new business conditions, modifications on the necessities of the clients and users, reorganisations of the development teams, and financial and time restrictions in the projects. Software configu-

Table 2.2 Maintenance categories

	Correction	Improvement
Proactive	Preventive	Perfective
Reactive	Corrective	Adaptive

ration management (KA6) aims to identify the configuration of the software system, in distinct moments of the lifecycle, to systematically control the changes of configuration and to maintain the integrity and traceability of the software system. This activity can be part of a more extensive process that aims to manage the software quality. The configuration management process of a given system over time can also be designated as version/release management. Software configuration management refers to the activities of control and monitoring that start at the same time as the project does and that terminate only when the system is no longer used.

Software engineering management (KA7) corresponds to software management activities (planning, coordination, measurement, monitoring, control, and communication), to guarantee that the software systems are engineered in a systematic, disciplined and measurable way. This KA is considerably distinct from the management practiced in engineering processes of other branches, due to the specificities of software and its process, like the intangible and abstract nature of the software artifacts and the very high rate of technological update that is required in the software industry.

The software engineering process (KA8) can be seen in two distinct perspectives. In the first one, the process is considered as a set of directives that guide how the professionals should organise and execute their activities over the project, for acquiring, developing and maintaining software systems. In the second perspective, one intends to evaluate and improve the software engineering process itself. Since the first perspective is already largely handled in the scope of other KAs, it is mainly within the second perspective that this KA contributes to. This explains why it is also designated, in a more restrictive way, but that simultaneously better characterises its focus, as engineering of the software process.

The use of software engineering models and methods (KA9) is fundamental to allow software systems to be engineered in a systematic, disciplined, quantifiable, and efficient way. Taking into consideration the abstract nature of software systems, the models constitute an indispensable tool when taking decisions in all the development process phases. The methods allow software models and other artefacts to be created and manipulated throughout the system lifecycle.

Quality must constitute a permanent concern of the engineers, since one expects engineering systems to possess high quality. The quality is related to the conformity of the system under development with the requirements. Software quality (KA10) is an activity that spreads all the software process and that requires the treatment of non-functional aspects like, for example, usability, efficiency, portability, reliability, testability, and reusability. The quality in the software must be seen as a transversal concern to all the software process.

> "Quality is never an accident. It is always the result of intelligent effort."
> *John Ruskin (1819–1900), social thinker*

The software engineering professional practice (KA11) has a highly multidisciplinary nature and is focused on topics related to professionalism, ethics, law, group dynamics, psychology, multiculturalism, communication, writing, presentation.

Software engineering economics (KA12) gathers contents of economic nature related to software systems in a business perspective. This KA includes topics like economics fundamentals (finance, accounting, inflation, time-value of money), life-cycle economics (portfolio, price, investment decisions), risk and uncertainty, and economic analysis methods (return on investment, cost-benefit analysis, break-even analysis).

KA13–KA15 are related to concepts and foundations of three disciplines that are critical to the success of the software engineer: computing, mathematics, and engineering.

2.3 Software

To understand the full scope associated with the software engineering discipline, it is convenient to figure out what is software, if viewed as an artefact or set of artefacts that result from the engineering process. In this section, the most relevant characteristics of software are presented.

2.3.1 Definition of Software

The term 'software' is relatively new and, according to Campbell-Kelly (1995), was used for the first time in 1959. Cusumano (2004, p. 91) says that Applied Data Research, a company founded in 1959, was the first one selling a software product separated from the hardware. At the beginning of the popularisation of computers (1950 decade), it was common practice to sell hardware and software as a unique system. Software was at that time viewed as part of the hardware and was designated as 'program code'. The emancipation of the software has its origin related to a relevant fact, from an historical point of view: the decision of american justice to demand IBM to distinguish, in its accounting documents, hardware and software, providing separate prices for each one (Buxmann et al. 2013, p. 4), something that has become reality since the 1970s.

Surprisingly, defining the concept associated with the 'software' term is not easy. A first attempt to define software can be made by a process of elimination. In a classic perspective, a computer consists of hardware and software and it only works

in a useful way if there is a fruitful and symbiotic combination of those two parts. In computers that follow a classical architecture, the hardware, by itself, is not capable of realising useful tasks from the user point of view. In fact, it is necessary to provide some of the hardware components with a list of instructions (in the form of a program) that defines the task to be accomplished. Equally, software to be executed needs a hardware support. Galler (1962) proposes that everything that, in the users perspective, composes a computer, except the hardware, is the software. Galler's definition is elegant due to its simplicity and has additionally the advantage of putting the users as a central element. Additionally, it induces the need to define the concept of hardware, which is not especially difficult to formulate, due to its tangible nature. The **hardware** of a computer is composed of electronic and electromechanical components, including, namely, the processor, the memory, and the input/output devices. The hardware of a computer refers to the material components (integrated circuits, printed circuit boards, cables, power supplies, plugs, and connectors) and not to algorithms or instructions.

> "Hardware and software are logically equivalent. Any operation performed by software can also be built directly into the hardware and any instruction executed by the hardware can also be simulated in software."
>
> *Brian Randell (1936–), computer scientist*

A definition of software, elaborated without a process of elimination, is however necessary. According to Blundell (2008, p. 4), **software** refers generically to the *programs*, which include the instructions that are executed by the computer (more specifically the hardware), as well as the *data* that are operated by those instructions. For Ceruzzi (1998, p. 80), software is simply the set of instructions that direct a computer to do a specific task. Alternatively, Tanenbaum (2006, p. 8) defines software of a computer system as being the *algorithms* (instructions that describe in detail how to accomplish something) and their respective representations, namely the *programs*. A given program can be materialised in different physical means (punched card, magnetic tape, floppy disk, compact disc, etc.), but its essence resides in the set of instructions that constitute it and not in the support where it is stored. The software is the set of programs, procedures and rules (and occasionally documentation), related to the operation of a system that aims to acquire, store, and process the data. Software is the abstract element that, together with the hardware, constitutes the automatised part of a real-world system, implementing a stimulus-answer mechanism, with the objective of satisfying the needs of an entity external to the system.

An intermediate form between hardware and software is the so-called **firmware**, which consists of software embedded in electronic devices during their manufacturing. Firmware is used when it is expected, for example, that programs are never (or rarely) changed or when programs cannot fail in case of lack of power supply. Thus, firmware is typically stored in non-volatile memory. In some processors, the operation is controlled by a *microprogram*, which is a form of firmware.

It is important to understand that the most classic view about software, as being a program that executes in a personal computer, is nowadays far away from being the only one. Software is an integral and fundamental part of the so-called *embedded systems*, which are computer-based systems integrated in equipments (typically electromechanical). In an embedded system, the main actions are performed by the hardware, but software plays a major role. Embedded systems are developed for a specific purpose and the communication with the outside world occurs through sensors and actuators. Embedded systems normally run continuously and in real-time. The software that exists in those system is called **embedded software**, because it is integrated in a system that cannot be classified as being of software. For this reason, embedded software is not sold in an independent way (Kittlaus and Clough 2009, p. 6). The end users generally do not associate to this software type the same characteristics of the most traditional software. Instead, the users perceive the software as a set of functions that is provided by the system.

A modern automobile has today a significant percentage of its engineering effort related to the production of software. Pretschner et al. (2007) indicate that the BMW 7 series models implement 270 functionalities which the automobile users can interact with and that the software, in total, occupies around 65 Mb (65×10^6 bytes) of code in binary language (i.e., the program codified in the language that can be directly executed by the hardware). These authors predicted that, in 2010, the software of a top-of-the-range automobile could reach the 1 Gb (10^9 bytes) figure, but according to Ebert and Jones (2009) this fact occurred one year earlier. Cellular phones are nowadays equipped with much more software than the one that could be found some years ago in computers of large organisations or corporations. According to figures indicated by Ebert and Jones (2009), a top-quality mobile phone can posses 1,000,000,000 (10^9) lines of binary code, the same being true of aerial navigation systems or with software to control spacial missions. Even less sophisticated devices, like washing machines, low-end mobile phones, or *pacemakers*, have approximately 1,000,000 (10^6) lines of binary code. In factories, there are so many pieces of equipment and processes controlled by tailor-made software systems. The electrical energy that arrives at homes depends, in large measure, on software that controls its management and distribution. Due to the fact that the world is becoming more digital at various levels, the examples are almost endless, due to the omnipresence, sometimes unnoticed, of systems with software in the modern societies.

> "The future lies in designing and selling computers that people don't realise are computers at all."
>
> *Adam Osborne (1939–2003), computer engineer*

The first characteristic of a software system, that distinguishes it from other engineering systems, is its intangible nature. A software system is not a concrete and material entity, that is, it has no physical existence, contrarily to what happens with most systems from the other engineering branches (civil, mechanical, naval, chemistry, electrical). The software is not restricted by the materials properties, nor ruled

by the laws of physics. Ghezzi et al. (1991, p. 17) indicate that software is soft, since it is relatively easy to change it. The softness of software, which is explicitly reflected in its name, results from its condition of intangible system. This characteristic has been, however, the cause of some of the problems associated with software development, since the changes imposed are often made without a careful analysis in terms of schedule, cost, quality, and impact.

Due to its intangible nature, software is developed, but not fabricated or constructed in the classical meaning of the term. In this book, a company that develops software and where many software engineers work is designated as a **software producer**. Often one uses 'software supplier', as a synonym, although this designation may have a more commercial connotation, since the one that supplies (i.e., sells, rents, lends, offers) something is not always the one that fabricates it.

Due to the intangible nature of software, in reality producing the first replica of a software system implies high costs, but the subsequent replicas are produced at much lower (in some cases, insignificant) costs. Additionally, copying software is an extremely easy operation, from a technical point of view, and that does not introduce loss of quality, since in the digital world one can consider that there are no differences between the original and the replicas. Due to these facts, the cost of software is essentially determined by the cost of the human resources necessary to develop it.

> "Software is like entropy: It is difficult to grasp, weighs nothing, and obeys the Second Law of Thermodynamics; i.e., it always increases."
>
> *Norman R. Augustine (1935–), aeronautical engineer*

A second characteristic of software is related to the fact that supposedly it does not wear out, in the sense that it does not lose its qualities over time. Although software does not wear out, in the physical sense of the term, it exhibits an enormous deterioration or degradation, derived essentially from the alterations that are introduced with the aim of maintaining its usefulness. Actually, the incorporation of new functionalities implies, almost inevitably, the introduction of defects in the software, which means that it eventually loses quality during its lifecycle.

2.3.2 Software Systems and Products

As already indicated, a system, in the context of engineering, is an identifiable and coherent set of components that cohesively interact to achieve a given objective. This definition permits that almost everything that exists in the universe can be seen as a system, which turns out to be true, since it is difficult to imagine something that could not potentially be viewed in a systemic perspective. Maybe an electron, if considered as an elemental particle (without components), could not be viewed as a system. However, for the engineer, what matters is not knowing whether something

is a system or not, but instead if that thing is viewed by him as a system (Thomé 1993). This happens because the engineer has an interest in studying the properties of that entity as a system. It is the engineer that defines the frontier of the system with the environment, which makes the system definition not intrinsic to it, but rather dependent on the particular purposes and intentions of the engineer in each situation. As a consequence, the components that in a specific context constitute a given system may be just a subsystem of a wider system in a different context. The term 'system' is used here in a comprehensive way, including concepts like structure, machine, product, process or service. Terminological variants such as apparel, appliance, artefact, equipment, gadget, installation, instrument, object and organisation, may also be used for designing systems.

Here, two criteria are proposed to classify software systems (software-dominated or software-intensive systems): 'what is sold' and 'number of copies' (Xu and Brinkkemper 2007). Crossing these two criteria gives origin to the types of software systems shown in Fig. 2.1.

Whenever the final customer buys a given appliance that includes software, this is normally designated as *embedded software*. This term is used, either for a unique appliance (for instance, a satellite or a spaceship), or for devices produced in large numbers (for example, television sets or mobile phones). These devices correspond to heterogeneous systems (that include parts with different technologies), rarely being referred to as 'software products', even in the cases where the software parts correspond to the most important technological dimension of the system. Gadgets like digital cameras, smart phones, or printers are generically called 'consumer electronics products', in spite of including significant portions of software. The comparison factor is here the complexity of the technological effort. In the case of heterogeneous systems with software when viewed as products, the devaluation of the software technology results from the typical focus of the end users on the mechanical or electronic parts that exist in those tangible products, rather than on the software parts. Based on a mathematical analogy in relation to the product designation, in a heterogeneous system, the software corresponds to the neutral element and never to the absorbing one.

Whenever the software system uses exclusively software technologies (i.e., something that can be designated as a pure homogeneous software system), then there are

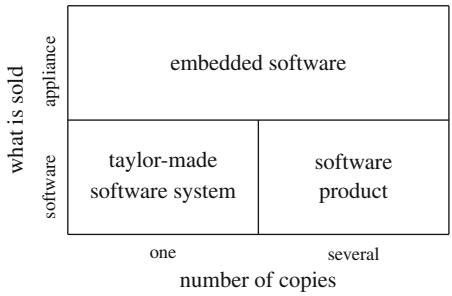

Fig. 2.1 Classification of software systems

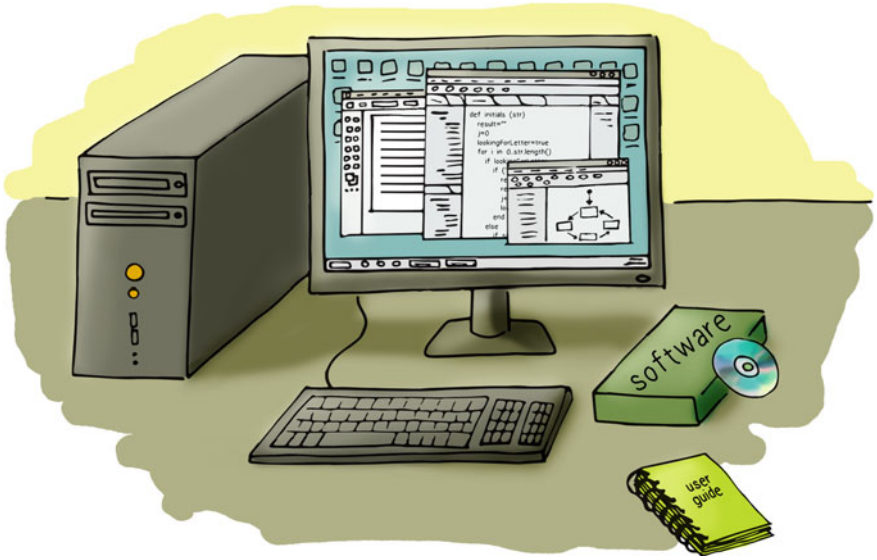

Illustration 2.2 A software product

two different types of systems. If the system is developed by request of a given client for satisfying his own necessities and expectations, then it is referred to as **tailor-made software system** (also called custom software system or bespoke software system). In this case, the principal objective is to satisfy the particular and specific needs of that client, without caring if it is equally useful for other clients.

If a software system is produced to be commercialised for, or made available to, the public in general, then it is designated as a software product, also called mass-market product. Generically, a **product** is a combination of (material and immaterial) goods and services that the supplier combines, in accordance with his commercial interests, to transfer established rights to the client (Kittlaus and Clough 2009, p. 6).

> "I already am a product."
>
> *Lady Gaga (1986–), singer*

According to this perspective, a **software product** refers to a homogeneous software product, being composed of the following three elements (Illustration 2.2);

- **programs**, whose instructions when executed offer the functionalities of the product;
- **data structures** that permit programs to access the necessary information for their execution;
- **documentation** that explains how to install, use, and maintain the programs.

Here, one defines a (computer) **program** as a text, written in a symbolic language capable of being interpreted by a computer, and composed of a set of operations that operate on the data and that obey the controls that stipulate the execution moments. This perspective of product is essentially technological, since it considers that the persons are not part of it, contrarily to what happens in the information systems with a socio-technical nature, in which are included the persons, as performers of parts of the organisational processes.

In business contexts, the development of a software product aims to obtain the highest possible number of customers, in the scope of the market segment that is identified for its commercialisation. The software producer has frequently the objective of selling massively, to maximise the respective market share and the economic incomes. Hence, a software product is developed in conformance with the common denominator of the necessities of the different users. If there is a tiny set of users that have a specific need, it is very likely that that need will not be included in the product. In any case, an underlying difficulty to develop a software product results precisely from not knowing in advance (i.e., during development time) who will use it.

> "Before new products can be sold successfully to the mass market, they have to be sold to early adopters."
>
> *Eric Ries (1979–), entrepreneur*

One can also classify software products in relation to the proximity that they have in relation to the hardware. Generically, two types of software products[1] can be considered: (1) *system software* responsible for managing the hardware resources of the computer; (2) the *software applications* that perform tasks that are useful for the end users.

A **system software** is composed of programs that interact in an intense and direct way with the computer hardware. Thus, it is normal that system software is not explicitly used by the end users. This type of software includes the operating systems, utilities to monitor the resources (to analyse, configure, and optimise the computer), *device drivers* and network software (web servers, email servers, network routers).

A **software application** (or software app) is a software product developed to support the realisation of the individual tasks of the persons and the execution of the organisational processes (government, industry, commerce, services). These applications to be executed use a computer (hardware) and a system software, for instance, an operating system. Therefore, one can say that, from the users point of view, they are (well) above the hardware level. These applications are essentially seen as productivity tools, that is, tools to enhance the human or organisational capa-

[1] Middleware could be considered as a third type, but in this book, a simpler solution was adopted. According to this first criterion, middleware is basically a generic designation used to refer to the software that executes between the system software and the software applications. The objective is to promote an easy development of distributed applications, since middleware serves to transfer informations and data among programs.

bilities. The software applications help people in various tasks, like editing texts, preparing budgets, storing and searching information, drawing graphics and tables, performing calculations and an infinity of other things. Nowadays, software applications are no longer limited by hardware-related aspects, but instead by the human imagination and by cost restrictions and user habits (Cusumano 2004, pp. 280–281).

> "The aim of marketing is to know and understand the customer so well the product or service fits him and sells itself."
>
> *Peter F. Drucker (1909–2005), management consultant*

There are some aspects that distinguish the tailor-made software systems from the software products, namely those related to the origin of the necessity that led to their development. That necessity may come from an individual person or from the market (Wieringa 1996, p. 34). In tailor-made software systems, the origin is clear, since development occurs after an explicit manifestation of the necessity, made by the potential client, to support the realisation of individual tasks or the execution of organisational processes. The development effort related to software products can be initiated, regardless of an explicit manifestation of the necessity by the potential clients. It is common that the necessity is identified by marketing experts, based on market studies and analysis of consumption trends. For tailor-made software systems, only one unique instance is usually made available, whilst for software products it is necessary to produce various installations for exploration by different clients. The software industry is a business area with a high return on investment, where making a unique copy of a software product or several thousands costs roughly the same (Cusumano 2004, p. 15). This reality allows the investments in software to have the potential to result in substantial profits, as also happens in the film, music, and pharmaceutic industries. However, the development of a software product has normally a relatively high fixed cost that is not recoverable, if the product is not a commercial success.

In some cases, it is difficult to classify a software system. For example, ERP (*enterprise resource planning*) software products, like the SAP ERP or the Microsoft Dynamics NAV, are generic, but can be configured to respond to specific requirements of a given client. Despite the referred terminology, the differences between a software system, a product and an application are often not significative, which means that the three terms are practically used as synonyms. A tailor-made software system can be transformed into a software product, through the generalisation of the functionalities it offers. The contrary is also true, that is, a software product can be adapted to satisfy the particular requirements of a given client. For this reason, in this book, the three terms are, in many situations, used for representing software artefacts, with high complexity and whose development requires approaches, methods, and tools from the software engineering sphere.

2.3.3 Domains

Engineering systems are developed due to the existence of some necessity of the stakeholders that must be satisfied. The area in which the system is explored is designated as domain. It is necessary to precisely characterise what is its meaning, since this word has different meanings, as a function of the context where it is used. Generically, a domain can be considered as a business area, collection of problems, or knowledge area with its own terminology. Within this book, a **domain** is an area of human knowledge or activity that is characterised by possessing a set of concepts and terms that the respective players know.

Examples of domains are telecommunications, transports, health, agriculture, industry, retail, banking, insurance, education, entertainment, cinema, and theatre. Domains can involve the physical world (for instance, a library involves the manipulation of books) or can be intangible (for example, schedule management). Generally, the domains have no relation with computers, although there are exceptions (for example, hardware commerce or source code management). The domains where the software technology can be present are only limited by the human imagination.

It is not obligatory that, before initiating the development of a system in a given domain, the requirements engineer has any knowledge about that domain. Obviously, it is desirable that she is at least comfortable with some of the basic concepts of the domain, so that he can speak in an comprehensible way with the system stakeholders. Over the project, the requirements engineer must increase the knowledge that he possesses about the domain, even as a mechanism to make sure that the development team takes into account the client's perspective. Some software producers are specialised in a given domain, in order to gain a competitive advantage with respect to their competitors. Although this approach reduces the potential market, the solid knowledge on the respective domain permits more specialised systems to be developed, thus offering a better answer to the users needs.

In relation to a given system, it is common to refer to two distinct domains: the problem domain and the solution domain. The *problem domain* is the context where one feels the necessities that need to be satisfied by the system to be developed. For instance, in the case of a restaurant, the problem domain includes the elements that characterise it: clients, cooks, tables, chairs, towels, cutlery, crockery, meals, etc. If the problem domain is an airline company, then one can see, for example, airplanes, pilots, stewards, passengers, suitcases, and tickets. The persons have technical or business problems that can be solved with the engineers contribution. The aim of the requirements engineers consists in understanding what are the problems of those persons, in their language and culture, so that one can construct systems that satisfy the necessities of those persons (Leffingwell and Widrig 2000, p. 19). The *solution domain* refers to the activities that are executed and the artefacts that are handled and constructed to solve the problem.

Davis (1990, pp. 29–32) classifies the software systems domain along five orthogonal axes:

1. difficulty of the problem class;
2. temporal relationship between data and processing;
3. number of tasks to be executed simultaneously;
4. relative difficulty of the data, interaction, and control aspects of the problem;
5. determinism level.

> "If there is no solution to the problem then don't waste time worrying about it. If there is a solution to the problem then don't waste time worrying about it."
> *Dalai Lama XIV (1935–)*

The problem class is about how the real problem (i.e., the problem felt by the stakeholders in the context of the problem domain) can be framed into the conceptual problem that covers it and that may have been previously studied and analysed by experts. For example, the real problem of a logistics company that aims to define routes that minimise the delivery times and fuel expenses can be framed into a more conceptual problem, like the travelling salesman or Chinese postman problems. The difficulty of the problem class can be divided into two groups. The *difficult problems* are those that were never solved or that do not have any satisfactory solution. The *not-difficult problems* are all the others, that is, those that have been previously resolved in a reasonable way.

With respect to the temporal relations that exist between the availability of the input data and its processing, there are also two classes. In *static applications*, all the inputs must be available before the application processes them. In *dynamic applications*, the input data arrive continuously during the processing, therefore having an effect on the results. Compilers are typically static applications, while interactive systems are examples of dynamic applications.

A third alternative of classification is related to the number of tasks that the system can simultaneously handle. *Sequential applications* manipulate a single task at a time, while *parallel applications*, from the users' perspective, must be capable of processing several tasks in simultaneous. The most difficult aspect related to the externally observable behaviour of the system to address constitutes another classification axis. It includes three cumulative dimensions that can be considered: data, interaction, and control. In a *data-centred application*, the type, organisation, and persistency of the data that support that application are the most critical aspects to consider. In *interactive applications*, the most difficult aspect to handle is how the environment and the system interact by exchanging and presenting information. In applications with a strong *algorithmic component* or *decision taking*, the primary aspect is the relation that is established between the system inputs and outputs, forcing or permitting the control levels caused by the exchange of information with the environment. These three alternatives are not mutually exclusive, since it is possible to observe significative manifestations of these three types of behaviour in the same system.

A last axis of classification refers to the predictability of the system outputs as a response to the inputs. In *deterministic systems*, one expects the same results to be produced for the same set of inputs. For example, a scientific calculator must always provide the same result for the same inputs. *Non-deterministic systems* provide answers that are not absolutely clear. It is possible that different outputs can be accepted as valid. For example, a software application to play chess can, in each move, opt for any of the various valid alternatives. There will be certainly some moves that are better than others, but that evaluation has not a unique answer, in the generality of the cases.

2.4 Models for the Development Process

The software process can be executed in different ways and according to different approaches. A *process model* represents a development process and indicates the form in which it must be organised. The process models aim to help the engineers in establishing the relation among the activities and the techniques that are part of the development process. Whenever the development process is modelled, one can reap the benefits that result from the systematisation and identification of the best practices, to allow systems development in an efficient, reliable, and predictable way. With the development process systematisation, through the definition of the respective model, one tries to reach the following objectives:

- to clearly identify the activities that must be followed to develop a system;
- to introduce consistency in the development process, ensuring that the systems are developed according to the same methodological principles;
- to provide control points to evaluate the obtained results and to verify the observance of the deadlines and the resources needs;
- to stimulate a bigger reuse of components, during the design and implementation phases, to increase the productivity of the development teams.

The difficulties faced by the software development teams and, more specifically, their managers lies in defining processes that promote the utilisation of management mechanisms that keep the projects under control. The challenge here is not blocking the necessary creativity and flexibility that is required so that the system at hand can adapt to changes both in technology and the users needs.

The next subsections present and characterise the most fundamental process models. To completely describe a process, several facets should be considered: the activities (set of tasks that must be executed to develop the system), the artefacts (results of the activities), and the roles (responsibilities of the persons engaged in the process). In the majority of the cases, this book presents, in a graphical form, only the activities, leaving for the textual part the discussion of the other facets whenever necessary.

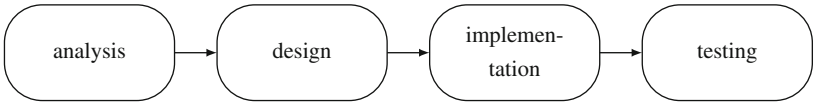

Fig. 2.2 The waterfall process model

2.4.1 Waterfall

The oldest software development process model is designated as the *waterfall model*. As Fig. 2.2 depicts, it is composed of a sequence of phases, namely analysis, design, implementation, and testing. The use of the 'waterfall' word aims to evince the irreversibility whenever one progresses from one phase to the next one, as well as the risk associated with the process execution. The most relevant characteristic of this process model is the strong tendency for the development to follow a top-down approach (from the most abstract to the most concrete) and, in a high-level perspective, the strictly-sequential progression between consecutive phases (Yourdon 1988, pp. 45–47).

During the *analysis* phase, the functioning of the system is specified, through the identification of the various requirements that must be considered. The document that contains the specification serves as a basis for the next phases, so, ideally, one should use implementation-independent notations and allow all stakeholders to clearly understand (i.e., without any ambiguities) what are the intended functionality.

Once the document that specifies the system under development is accepted, the *design* phase, which consists in transforming a specification into an architecture, begins. According to Bosch (2000, p. 230), the most complex activity during software development is precisely the transformation the requirements into an architecture. Generically, this phase is divided into two steps. The first one, designated as architectural design, describes how the system is constituted and is, in many cases, one of the most creative tasks in all the development process (Stevens et al. 1998, p. 88). In this first step, an architecture must be established, by identifying the system components and possible restrictions in their behaviour. This architecture determines the internal system structure, defined based on the entities that compose it and in the relations among them. Whenever the architecture is defined, the second step, designated as detailed design, establishes in detail the components, in order to include enough information to allow its implementation. Sometimes, one considers the existence of a step, called the mechanistic design, that relates the decisions taken in the architectural design and in the detailed design, through the detailed study of the mechanisms that provide the architecture with the expected behaviours for the system.

In the design phase, the principal objective is to structure (i.e., to define the architecture of) the system at hand. For example, an object-oriented design includes the object-oriented decomposition process, using an appropriate notation to describe all the (logical or physicals and static or dynamic) aspects related to the system (Booch et al. 2007, p. 42). In a system conceived according to the object-oriented

paradigm, the respective structure is dictated by the objects that compose it and the relations that are established among them.

The principal difference between the analysis and the design phases is that while the former produces an abstract model that mimics the fundamental aspects of the existing needs in the problem area, the latter creates a model that specifies the components that structure a particular system solution. In other words, the analysis phase defines the system functionality (what to do), while the design phase stipulates the architecture (how to do) that the system must present so that the expected behaviour is obtained.

Despite the many methodological and technological advances that have occurred in the last years, the software is often developed in an handicraft way. The software industry is still far from reaching a point where there is, in fact, an entire catalogue of software components able to be easily and directly integrated in the systems. In software engineering, the development of all the parts of a system is not only generally accepted but also often encouraged, with the argument that this is the only way to build the system with the intended quality. This argument is called the not invented here (NIH) syndrome Lidwell et al. (2010, pp. 168–169). Such scenario would be, nowadays, unthinkable in other industries. Fortunately, there are many mechanisms, such the component libraries, frameworks, application programming interfaces (APIs), design patterns, which provide very significant advances in this area. Web applications constitute an important niche, where the use of reusable components, like web services, is quite common.

The *implementation* (codification or programming) phase transforms the models defined in the design phase in executable code. This transformation involves the definition of the internal mechanisms so that each component can satisfy its specification and the implementation of those mechanisms with the chosen programming language (Zave 1984). The implementation phase is considered by many authors, for instance, Hatley and Pirbhai (1987, p. 10), Rumbaugh et al. (1991, p. 278) and Whytock (1993), as a a purely mechanical, simple, and direct task, after the most intellectual and creative work has been performed in the analysis and design phases. The implementation phase is thus a serious candidate to be automated, if one can rely on tools that permit us to indicate how the final code can be generated from the specifications obtained in the previous phases. However, reality has shown that it is not always so easy to pursuit with this phase. Object-oriented programming allows systems to be implemented, organised as collections of objects. Each object is an instance of a class and each class is a member of a structure where there are hierarchical relationships.

While in the development of non-complex systems, the effort devoted to the analysis and design phases can be residual (when compared with the effort in the implementation phase), for complex systems the effort devoted to issues related to analysis and design is of vital importance. Actually, the popularisation of tools, which automatically produce code from specifications, allows one to say that the point where “the specification is the implementation” is about to be reached. The essence of developing systems becomes thus focused on decisions related to the analysis and design phases.

Even when the engineers carefully follow development processes and approaches, the artefacts that result from development activities may still contain defects. To avoid these possible defects, the systems must be tested. *Testing* has two main objectives: (1) to show that the system under development does what it is expected to do, and (2) to allow defects in the system to be discovered.

> "If debugging is the process of removing software bugs, then programming must be the process of putting them in."
>
> *Edsger W. Dijkstra (1930–2002), computer scientist*

The term 'testing' encompasses a very large set of activities that go from testing a small piece of code by the programmer (unit testing) up to the validation, by the client, of a software system (acceptance testing). To distinguish some of these activities, the community adopts the terms 'verification' and 'validation'. By *verification* one means the process through which it is ensured that the system was built in accordance with the requirements and the specifications. The verification can be achieved through dynamic approaches, in which the run-time behaviour of the system is checked, or static ones, where one analyses and inspects any system-related artefact or document (Ghezzi et al. 1991, p. 260). The aim of *validation* is to make sure that the system satisfies the necessities and expectations of the users and the clients.

The testing phase was traditionally executed at the end of the development process. The program code was totally written, before executing any function testing. However, this vision needed to be changed, as soon as it was clear that testing is more that just debugging code. Software testing, if well performed, can result in huge economical benefits. Nowadays, testing complex software takes around 40 % of the development costs (Ebert and Jones 2009; Sommerville 2010, p. 6), which demonstrates the growing importance that it has in the software engineering context. Actually, the success of software testing depends on the planning of its execution and its effective realisation in the initial development phases. If quality of a software system is strongly dependent on the quality of the adopted development process, similarly, the quality and effectiveness of testing are largely determined by the quality of the testing process (Kit 1995, p. 3). Software testing has its own lifecycle, which is realised at distinct levels: it starts at the same time as the requirements elicitation and, from that point on, follows in parallel with the development process. In other words, for each phase or activity of the development process, there is an associated testing activity, as Fig. 2.3 depicts (Robinson 1992, p. 3). This figure, which represents the V process model, can be viewed as a refinement of the waterfall model shown in Fig. 2.2.

The first level of testing (unit testing or unitary testing) takes place as the diverse components (units) are implemented. The objective is to verify if each component in isolation works as expected, using the decisions taken in the detailed design as reference. When the components pass the unit tests, the following step is the *component integration testing*, whose purpose consists in guaranteeing that the interfaces between the components have the behaviour estimated during the architectural

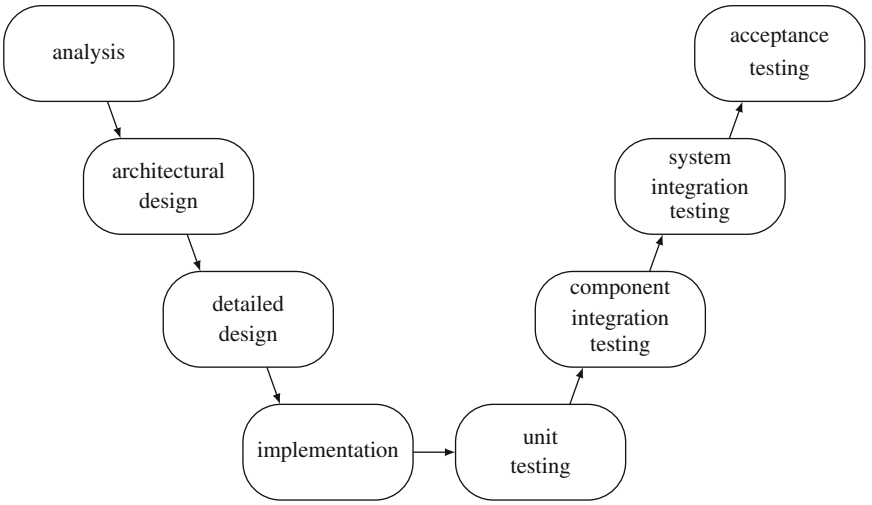

Fig. 2.3 The V process model

design. Next, the *system integration testing* permits one to verify if the software system, as a whole, satisfies the requirements indicated in the analysis phase. The two types of integration tests indicated can be very demanding activities in terms of resources and time, especially for critical systems. Finally, the *acceptance testing* is executed jointly with the end users that validate the operation of the system with respect to their expectations, in the light of what has been contracted. Some testing activities can be automated, partially or even totally, and that possibility depends on the notations used in the analysis and design phases.

All these phases (analysis, design, implementation, and testing) are related among them and none should be neglected during the development of a given system. However, the division among these phases is not always so explicit as it was indicated up to now. For example, in the object-oriented methodologies, there is normally an overlap in the tasks covered in the analysis and design phases, since the separation between those two phases is more theoretical than real, being very difficult to define the respective frontier (Booch et al. 2007, p. 131). This fact can be interpreted as a disadvantage, but a positive perspective is also possible, meaning that the transition between those phases is made in a natural and soft way, when an object-oriented decomposition is followed.

The waterfall model, due to its conceptual simplicity is still one of the most referred software development processes, despite the various problems that it presents and the many alternatives that were proposed. The waterfall model is considered too inflexible and produces satisfactory results only when the requirements are clear and the chances of being changed too low. To develop, for instance, a compiler, based on a grammar completely defined and which is not likely to be changed, this model seems to be perfectly adequate (Ghezzi et al. 1991, p. 374).

However, the waterfall model, is based on documentation as a criterion to decide when a phase is finished and the next one can start, which requires complete documents to be written. This fact is extremely negative, in projects where the ideas and the necessities of the stakeholders are not yet totally clear, since it forces all the requirements to be decided very early. This fact probably introduces errors that will be spread to the design and implementation phases. Unfortunately, those errors are only detected when the implementation is finalised, usually quite after the specification of the requirements.

The processes that follow the waterfall model tend to be accomplished according to a plan in which generically the requirements are specified completely, so that one can subsequently design, construct, and test the system. The waterfall model, despite allowing the process to be executed in various iterations, is essentially characterised by following a process with a significant level of bureaucracy and ceremony.

In practice, the order in which the phases are executed is difficult to capture precisely, because it is not necessary that a phase finishes totally for the next one to be initiated. This fact is contrary to what the waterfall model theoretically proposes. Usually, the results that are obtained in a given phase are used for correcting the results of the previous phases. Thus, the processes should be iterative in practice, since problems identified in more advanced phases of the project force the previous phases to be revisited.

2.4.2 Incremental and Iterative

In development contexts, where the market is extremely accelerated and vibrant, with new opportunities arising at very fast rhythms, the waterfall process is very inadequate. The *incremental and iterative model* is based on the characteristics of the waterfall model, introducing however iterations to permit an incremental development. As Fig. 2.4 illustrates, this process model applies linear sequences of development in a phased way. Each linear sequence produces as result a functional increment of the system. For example, the construction of a text editor could proceed through the following iterations:

- in the 1st iteration, one develops the functions to manipulate files (open, save, close, print) and the basic functionalities for editing text (insert, delete, select);
- in the 2nd iteration, one includes more advanced edition capabilities (search and replace text, fonts, bold, italics);
- the 3rd iteration allows the inclusion of figures, tables, and graphics in the documents, by providing new commands that manipulate those elements;
- in the 4th iteration, new functionalities that permit the use of thesaurus and automatic spellers are added.

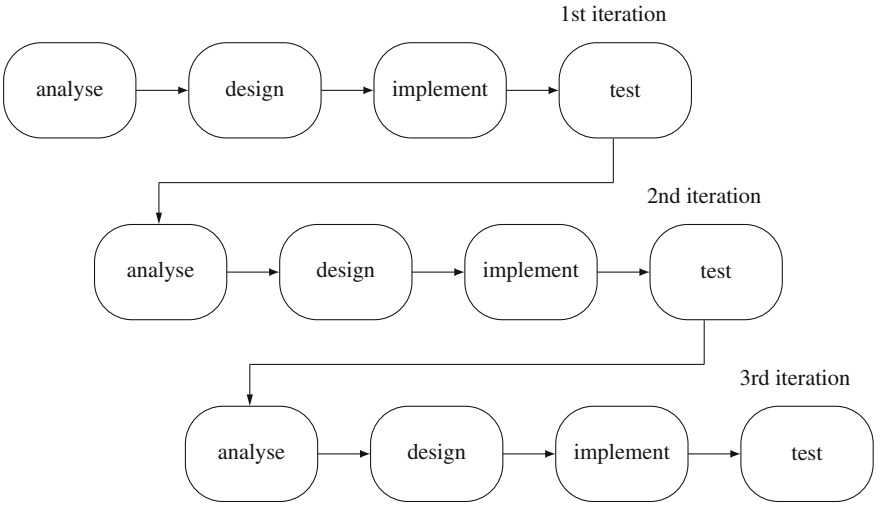

Fig. 2.4 The incremental and iterative process model

> "You should use iterative development only on projects that you want to succeed."
>
> *Martin Fowler (1963–), software engineer*

The incremental and iterative model is based on the idea that it is easier to create a simple artefact than a complex one and also that it is simpler to modify an existing artefact than to create a new one from scratch. Hence, instead of trying to completely construct the system in a unique cycle, the attention at the beginning is deliberately focused on incorporating the most critical and important aspects (or those that are more clearly mastered). Afterwards, new aspects can be progressively incorporated, until the system is complete.

The incremental and iterative process is, in its essence, repetitive, i.e., the functionalities are gradually added and improved until the system is fully handled. Each increment represents a part of the system functionality.

Agile methods for developing software can be framed within this type of process model. They appeared, in the 1990 decade, as an alternative to the traditional methods, dismantling many of the assumptions associated with the latter. The accelerated dynamics in various domains and businesses make impossible, from a practical point of view, obtaining a set of stable requirements, since they are subject to constant and unexpected changes. Agile methods seek to position themselves as an adequate alternative for these highly unstable scenarios. For that purpose, they follow four fundamental principles, established by Beck et al. (2001), which, despite recognising value and utility to the items on the right, give prevalence to those on the left:

individuals and interactions	vs.	processes and tools
working software	vs.	comprehensive documentation
customer collaboration	vs.	contract negotiation
responding to change	vs.	following a plan

Agile methods aim essentially to minimise the risk associated with software development, defining for such very short development cycles, called iterations (or *sprints*). They are incremental and iterative development methods, in which the cycles last between one and four weeks. An agile software project aims to produce, at the end of each iteration, a new version/release of the system, that is, a version that is executable, that works correctly and that provides value to the client. Hence, each iteration includes a cycle with all the tasks necessary to concretise the inclusion of the new functionalities: requirements engineering, design, implementation, testing, and documentation.

> "What is the difference between method and device? A method is a device which you used twice."
>
> *George Pólya (1887–1985), mathematician*

At the end of each iteration, the team reevaluates the requirements and can introduce alterations. This practice permits that set of requirements to be changed, through the inclusion of new requirements and the elimination or change of requirements previously identified. Additionally, the priority of each requirement can be changed. This reevaluation of the requirements implies that they can be effectively changed, during the project, thus increasing the utility and value of the system for the stakeholders.

Agile methods lie in the real-time and, if possible, face-to-face collaboration among the development team, the clients and the users. This collaboration allows the team to discuss the project scope, analyse and prioritise the requirements, and decide the options to be taken. Thus, it is not so critical the existence of written documents to support the development tasks. Actually, most agile software projects, in comparison with other approaches, produces less voluminous documentation.

2.4.3 Transformational

Although they are not yet a generic solution for all types of systems, mathematical methods (typically designated formal methods) offer the very attractive perspective of generating software with no defects. The use of formal methods presupposes a *transformational process* that assumes the existence of tools that automatically convert a formal specification into a software program that satisfies that specification. This process implies that the changes are reflected in the specification, thus eliminating the problem of getting *spaghetti code*, i.e., code that gradually becomes poorly structured, as a consequence of being successively modified.

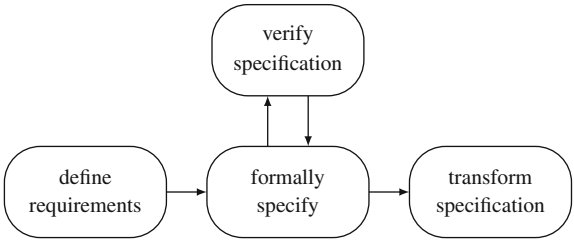

Fig. 2.5 The transformational process model

Figure 2.5 illustrates the transformational process model that is composed of four main tasks. In the first task, requirements are elicited, with a set of techniques that are considered adequate. Based on the requirements, a formal specification is created and is progressively developed, until an executable version is obtained. This characteristic means, in this context, that the specification can be processed, from which results a program that can be executed. An executable specification is more that the mere static model that defines the system behaviour and should allow the verification of the behavioural properties. Some of the benefits observed in the utilisation of executable specifications, in several large-scale projects, are described by Harel (1992). If some non-conformity is detected during the verification, the specification should be modified, in order to eliminate that problem, and verified again. This cycle repeats the necessary number of times until the specification is in accordance with the requirements.

> "It is easier to change the specification to fit the program than vice versa."
> *Alan J. Perlis (1922–1990), computer scientist*

The specification is used to obtain, through automatic transformation mechanisms, parts of the final program with the efficiency levels specified in the requirements. The process of transforming the specification is controlled by the software engineer, which thus can vary the characteristics of the program, taking into account, for instance, the non-functional requirements. This process model becomes efficient and useful, if there is an environment that provides tools to automatically support the various activities, especially those to transform the specifications into the program.

The specification changes are considered as a part of the process. Contrarily, in a waterfall process, changes are seen as reparations, since their occurrence is not considered beforehand. From here it results that changes are made under big pressure, normally at the end of development, implying often that the changes are made modifying directly the code, without reflecting them in the specification. Hence, the specification and the implementation diverge one from the other, making any future changes more difficult to accomplish. This situation does not occur in a transformational process, since the process executed to develop the software system and the respective decisions (intermediate steps, based on mathematical proofs) for each

transformation are registered, which makes it possible to restart, from an intermediate point, the transformation of the specification into an implementation.

Despite the growing maturity that the formal methods have acquired, their utilisation is not yet disseminate in a pervasive way, being its application restricted to well-identified areas. Among the disadvantages of their utilisation, one can find the long development time, the difficulty in using the specifications as a medium of communication with the clients, and the need to resort to specialists to manipulate the mathematical specifications. This last argument should not be taken literally, since some formal methods employ concepts familiar to any engineer. More recently, the model-based development approach resort to the transformational principles of the formal methods, but it adopts models closer to the implementation languages (than the strictly-mathematical approaches), as a way to make viable the transformational approach in large-scale real projects.

2.4.4 Spiral

The *spiral process model* (Boehm 1988) is based on a risk-driven approach and not on documents or the code. In this context, a **risk** is a potentially adverse circumstance that can have negative or perverse effects in the development process and in the final quality of the system. A risk is a measure of the uncertainty of achieving an objective or meeting requirements. The spiral model centres its action in the identification and elimination of problems with high risks.

The various tasks are organised in cycles, as Fig. 2.6 documents. Each cycle of the spiral is constituted of four main tasks, being each one represented by a quadrant of the diagram. The radius of the spiral represents the progress in the process and the angular dimension indicates the accumulated cost in the process.

In the first task of the cycle, one identifies the objectives (performance, functionality, easiness of modification, etc.) for the system under development, with respect to the quality levels to be achieved. It is also relevant to identify the alternative means of implementation (develop A or B, buy, reuse, etc.) and the restrictions that must

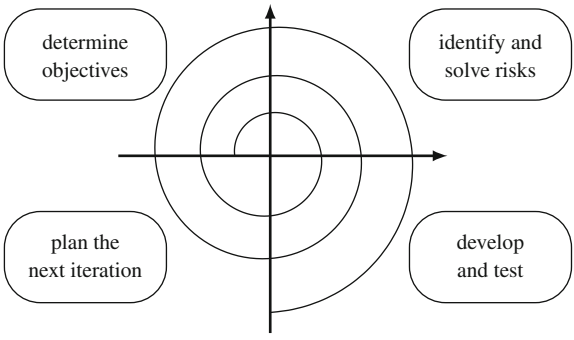

Fig. 2.6 The spiral model

be assumed to materialise one of those alternatives. In the second task of the cycle, one evaluates the alternatives previously identified with respect to the objectives and restrictions, which frequently implies the identification of uncertain situations that represent potential sources of risk. To proceed with this identification, one can resort to different techniques, like prototyping, benchmarking, simulation or questionnaires. During the third task, one should develop and verify the system for the next cycle, based again on a risk-oriented strategy. In the fourth and last task of the cycle, the obtained results are reviewed and the next spiral cycle, if that is the case, is planned.

> "The basic problem of software development is risk."
>
> *Kent Beck (1961–), software engineer*

Whenever the requirements of a system are reasonably well known, a waterfall process can be followed, which means that only one spiral cycle is fulfilled. For systems whose requirements are less clear, several cycles may be necessary to achieve the intended results, from which results an incremental and iterative process.

As special cases, the spiral model includes the process models presented in this section. It allows the choice of the best combination and composition of those process models for each situation in which it is applied. Therefore, the spiral model can be seen as a meta-model, i.e., a reference model to instantiate different process models.

2.5 Summary

Software engineering is an engineering discipline focused on all the aspects related to the development and production of software. The software engineer, as any engineer, is responsible for the development process and production of artefacts that are to be used by third-parties. Software engineering addresses the development and production of software systems made by teams constituted by several professionals. This implies that software engineering is not just centred in the technical aspects associated with software development, but includes also process management activities. Software engineering brought to the computer science field the junction of the development processes and methods with economic and management issues. These questions are indispensable for the professionals that actuate in the industry with roles and responsibilities that go behind the mere computer programming.

The software engineering body of knowledge is structured into 15 KAs according to the SWEBOK guide. Software engineering is a scientific area with an extremely important economic and social relevance and that has allowed people to observe many of their daily tasks solved in a faster, more comfortable, and more economic way. This success benefits from the capacity of software engineering to adapt to the

enormous demands that nowadays their professionals are subject to. They have to handle all the software production process issues, but simultaneously possess the technological competencies and sensibility for the necessities and expectations of the users. The software engineering essence lies upon the capacity to put in practice the most appropriate sets of cooperative, coordinated and concurrent technological and methodological approaches for each software development reality.

Software is the principal artefact that results form the software engineering process. The chapter presents the most relevant software characteristics and defines and presents the various types of software systems. The software systems are developed to satisfy the stakeholders needs. The domain of a software system can be a business area, a collection of problems, or a knowledge area.

The chapter ends with a presentation and characterisation of the most popular process models used to guide the development teams in the organisation of their tasks. There is no unique development process model that is adequate for all the projects, which means that it is very important to choose the one that best adjusts to each particular context.

Further Reading

The literature in software engineering is quite extensive. There are some books that try to make a complete coverage of all the discipline. To start with, there are some works considered as true classics, like the encyclopedia-like books written, over the past years, by Sommerville (2010) and Pressman (2009). These books have go through several editions (always with updates in relation to the previous version), in the case of Pressman every 5 years approximately, which demonstrates the highly accelerated dynamics of this discipline. Another interesting book, written by Pfleeger and Atlee (2009), focuses on modelling and agile methods. Another landmark book in this area is the work authored by Ghezzi et al. (1991), which, despite its generalist nature, is quite concise and easy reading.

The history of software is also very interesting. Two recommended sources are Ceruzzi (1998, Chap. 3) and Campbell-Kelly (2003). Broy and Denert (2002) edited a book with a compilation of some of the scientific articles that, since the 1950s, changed somehow software engineering. This book constitutes a compilation of undeniable rigour and historical value, even though one can contend that "the" software engineering pioneers are not necessarily just the authors included in the book. An extremely interesting essay about the differences between software engineering and other more traditional engineering fields can be found in Beizer (2000).

Software engineering education is addressed by various authors, for example, Parnas (1990, 1999), Shaw (1990, 2009), Osterweil (2007). It is also relevant to read the *curricula* recommendations for software engineering (ACM/IEEE-CS 2004, Pyster 2009).

In the field of the software process, it is recommended that one reads the works about the software development process improvement, like, for instance, (Humphrey

2005). The book written by Krutchen (2003) presents the unified process. With respect to agile methods, it is suggested that one reads books related to eXtreme Programming (XP) (Beck 2000) and Scrum (Schwaber and Beedle 2001).

Exercises

Exercise 2.1 Indicate the 15 KAs that constitute the software engineering body of knowledge, according to the SWEBOK guide.

Exercise 2.2 Identify which elements are included in a software product.

Exercise 2.3 A *mobile application* is a software application, developed to run on a smartphone or other handheld device. Identify the most important characteristics of a mobile application, according to Salmre (2005, Chap. 2).

Exercise 2.4 (Naveda and Seidman 2006, pp. 23–24) While developing a software application, two similar defects were detected: one during the requirements phase and another one during the implementation phase. Which of the following sentences is more likely to be true?

1. The most expensive defect to repair is the one detected in the requirements phase.
2. The most expensive defect to repair is the one detected in the implementation phase.
3. The repair cost of the two defects tends to be similar.
4. There is no relation between the phase in which a defect is detected and the repair cost.

Exercise 2.5 Point out some advantages that result from using an incremental and iterative process in comparison with a sequential process that follows the waterfall model.

http://www.springer.com/978-3-319-18596-5

Requirements in Engineering Projects

Fernandes, J.M.; Machado, R.J.

2016, XVII, 225 p. 60 illus., Hardcover
ISBN: 978-3-319-18596-5