




PART

THE ESSENCE OF SOFTWARE ENGINEERING

We live at an exciting time in the history of computer and network technologies where software has become a dominant aspect of our everyday life. Wherever you look and wherever you turn, software is there. It is in almost everything you use and affects most everything you do. Software is in many things such as microwaves, ATMs, smart TVs, machines running vehicles, and factories, as well as being utilized in all types of organizations.

Although software provides many opportunities for improving many aspects of our society, it presents many challenges as well. One of them is development, deployment, and sustainment of high-quality software on a broad scale. Another is the challenge of utilizing technology advancements in new domains, for instance, intelligent homes and Smarter Cities. Here, the evolution of the mobile internet, apps, the internet of things (IoT), and the availability of big data and cloud computing, as well as the application of artificial intelligence and deep learning, are some of the latest “game-changers” with more still to come.

This book provides you with fundamental knowledge you will need for addressing the challenges faced in this era of rapid technology change. Part I will introduce you to software engineering through the lens of a kernel of fundamental concepts that have been provided by the Object Management Group’s standard called Essence 1. Essence is rapidly becoming a “lingua franca” for software engineering. The authors are convinced that this approach will provide a perspective that will be a lasting contribution to your knowledge base and prepare you to participate in teams that can develop and sustain high-quality software.



From Programming to Software Engineering

This chapter sets the scene with respect to the relationship between programming and software engineering. The important issue is that software engineering is much more than just programming. Of course, the running system created by an act of programming is an essential and rewarding ingredient of what the right system will become, and it is important that the reader is actually able to use and apply a programming language to create a program, at least a small one. But is it by no means everything. Thus, this chapter

- introduces the notion of software development and that it is more than just putting a program together;
- shows what additionally is needed beyond programming, i.e., shows the differences between programming, software development, and software engineering;
- shows the motivations for the discipline of software engineering;
- introduces some important elements of software engineering that actually show the differences between software engineering and programming, and shows how they relate to each other.

What is fascinating about this aspect of software development is that it is more than just programming. Rather, it is to learn the whole picture and as a software engineer to solve a problem or exploit an opportunity that the users may have.

As a new student, understanding what software engineering is about is not easy, because there is no way we can bring its realities and complexities into the student's world. Nevertheless, it is a student's responsibility to embark on this journey of learning and discovery into the world of software engineering.

Throughout this entire book, we will trace the journey of a young chap, named Smith, from his days in school learning about programming through to becoming

Sidebar 1.1 Programming

Programming is used here as a synonym for *implementation* and *coding*. From Wikipedia we quote: “Related tasks include testing, debugging, and maintaining the source code, These might be considered part of the programming process, but often the term *software development* is used for this larger process with the term *programming*, *implementation*, or *coding* reserved for the actual writing of source code.”

a software engineering professional and continuing his on-going learning process in this ever-changing and growing field. In a way, we are compressing time into the pages of this book. If you are a new student, you are considered to be the primary audience for this book. Smith will be your guide to the software engineering profession, to help you understand what software engineering is about. If you are already a software engineer by profession, or you teach and coach software engineering, you can reflect on your own personal journey in this exciting profession. As an experienced developer you will observe an exciting and fundamentally new way to understand and practice software engineering. Regardless of your current personal level of experience, through Smith’s experiences we will distill the essence of software engineering.

1.1 Beginning with Programming

The focus of our book is not about programming (see Sidebar 1.1), but about software engineering. However, understanding programming is an obvious place to start. Before we delve deeper into it, we should clarify the relation of programming to software development and to software engineering.

Thus we have chosen the following.

- *Programming* stands for the work related to implementation or coding of source code.
- *Software development* is the larger process which, apart from programming, includes working with requirements, design, test, etc.
- “*Software engineering* combines engineering techniques with software development practices” (from Wikipedia). Moving from development to engineering means more reliance on science and less on craft, which typically manifests itself in some form of description of a designated way of working and higher-level automation of work. This allows for repeatability and consistency from project to project. Engineering also means that teams, for example, learn as they work and continuously improve their way of work-

ing. Thus, stated in simple terms, *software engineering is bringing engineering discipline to software development*.

Going forward, when introducing software engineering we will mean the larger subject of “software development + engineering,” implicitly understood without specifically separating out the two parts. This will be so even if in many cases the discussion is more about the development aspect, because the approach we take is chosen to facilitate the other aspect—engineering. When we sometimes talk about software development we want to be specific and refer to the work: the activities or the practices we use. We will not further try to distinguish these terms, so the reader can in many cases see them as synonyms.

As a frequent user of applications like Facebook, Google, Snapchat, etc., whether on his laptop or his mobile, Smith knew that software forms a major component in these products. From this, Smith became strongly interested in programming and enrolled in a programming course where he started to understand what program code was and what coding was all about. More importantly, he knew that programming was not easy. There were many things he had to learn.

The very first thing Smith learned was how to write a program that displays a simple “Hello World” on his screen, but in this case, we have a “Hello Essence!”, as in Figure 1.1. Through that he learned about programming languages,

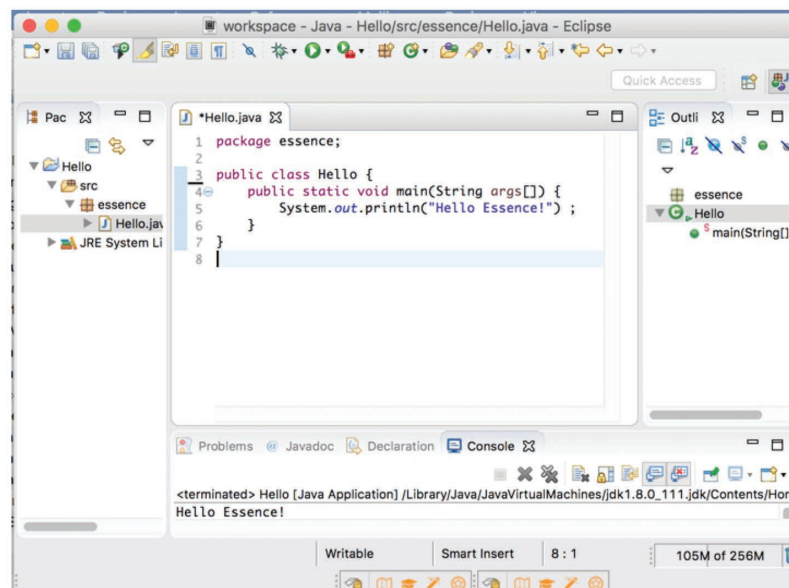


Figure 1.1 Hello Essence.

programming libraries, compilers, operating systems, processes and threads, classes, and objects. These are things in the realm of computer technology. We expect that you, through additional classes, will have learned about these things. We also expect you as a student to have some knowledge of these things as a prerequisite to reading this book. We expect that you have some knowledge of programming languages like Java and JavaScript.

1.2 Programming Is Not Software Engineering

However, Smith quickly learned that programming on its own is not software engineering. It is one thing to develop a small program, such as the “Hello Essence” program; it is a different thing to develop a commercial product.

It is true that some fantastic products such as those that gave birth to Apple, Microsoft, Facebook, Twitter, Google, and Spotify once were developed by one or a few individuals with a great vision but by just using programming as a skill. However, as the great vision has been implemented, be sure that these companies are today not relying on heroic programmers. Today, these companies have hired the top people with long experience in software engineering including great programming skills.

So, what is software engineering? Before we answer this question, we must first make it very clear that there is a remarkable difference between hacking versus professional programming. Professional programming involves clear logical thinking, beginning with the objective of the program, and refining the objective into logically constructed expressions. Indeed, the expressions are a reflection of the programmers’ thinking and analysis. Hacking on the other hand is an ad hoc trial and error to induce the desired effect. When the effect is achieved, the hacker marvels without really understanding why it worked. Professional programmers understand why and how it worked.

As such, professional programming is highly disciplined. Software engineering takes this discipline to software teams working on complex software. A typical software development endeavor involves more than one person working on a complex problem over a period of time to meet some objectives. Throughout Smith’s introductory software engineering course, he worked on several assignments, which frequently required him to work with his fellow students, and which included tasks, such as:

1. brainstorming what an event calendar app would look like;
2. writing code for a simple event calendar in a small group;
3. writing code for the event calendar app, and hosting the app on the cloud;

4. reviewing a given piece of code to find issues in it, for example bugs, and poor understandability; and
5. reviewing a fellow student's code.

Through these assignments, Smith came to several conclusions. First, there is no one true way to write code for a given problem. Writing good quality code that fellow students can understand is not easy. It often takes more than one pair of eyes to get it working and comprehensible. He learned the following.

- Testing, i.e., checking that the program behaves as intended, is not easy. There are so many paths that executing the code can follow and all have to be tested.
- Agreeing on what the application would do was challenging. Even for that simple event calendar app, Smith and his team debated quite a while before they came to a consensus on what functionality ought to be available, and how the user interface should be laid out.
- A simple application may require multiple programming languages. For example, the event calendar app would need HTML5 and JavaScript for the front end, and the Java and SQL database for the backend. Consequently, Smith found that he had to spend a significant amount of time learning and getting familiar with new programming languages and new programming frameworks. Although he endeavored to learn about all these, it was certainly not easy with the limited time that was available.
- Time management is not easy because it is hard to estimate how much time each activity will require—or when to stop fine-tuning a certain piece of code to meet time constraints of the project.

As Smith was preparing for his industry internship interview, he tried to summarize on a piece of paper, from those things he then understood, what software engineering is about, and what he had learned thus far. Smith drew what he understood many times, and he observed that he couldn't get it quite right. In the end, he settled for what is shown in Figure 1.2.

To Smith, software engineering was about taking some idea and forming a team according to the requirements. The team then transforms the requirements into a software product. To do this, the team engages in some kind of brainstorming, consensus, writing and testing code, getting to a stable structure, maintaining user satisfaction throughout, and finally delivering the software product. This requires the team to have competencies in coding, analysis, and teamwork. In addition,

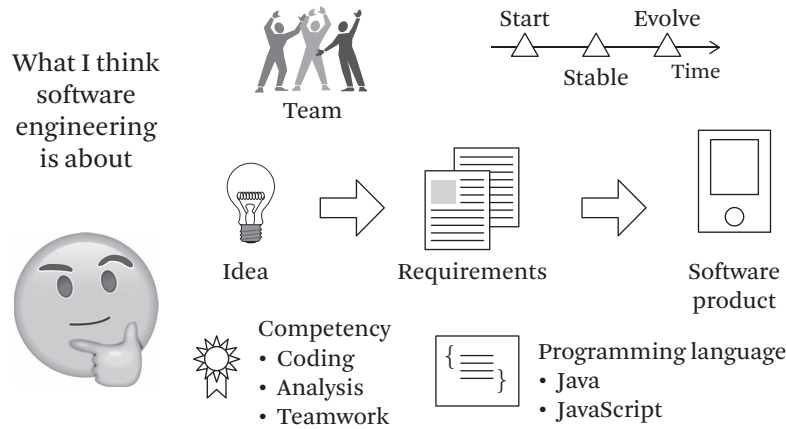


Figure 1.2 What software engineering is from the eyes of a student.

the team needs familiarity with some programming language, such as Java and JavaScript, which Smith knew. What Smith didn't yet know was that the tasks he had been given were still relatively simple tasks compared to what is typical in the software industry. Nevertheless, with this preparation, Smith marched toward his internship interview.

1.3 From Internship to Industry

With some luck, Smith managed to join the company TravelEssence as an intern trainee. Dave the interviewer saw some potential in Smith. Dave was particularly intrigued that Smith managed to draw the picture in Figure 1.2. Most students couldn't, and would get stuck if they even attempted to.

TravelEssence is a fictitious company that we will be using as an example throughout this book. TravelEssence provides online hotel booking services for travelers (see Figure 1.3). In addition, TravelEssence provides Software as a Service (SaaS) for the operation of hotels. SaaS means that the owner of the software, in this case TravelEssence, provides software as a service over the internet and the clients pay a monthly fee. Hotels can sign up and use the TravelEssence service to check-in and check-out their customers, print bills, compute taxes, etc.

Smith's stint in TravelEssence provided a whole new experience. To him, his new colleagues seemed to come from two groups: those who stated what they wanted the software to do, and those who wrote and tested the software. Figure 1.4 highlights the dramatic changes Smith experienced. While everyone seemed to speak English, they used words that he did not understand, especially the first group. As a diligent person, Smith compiled a list of some of this jargon.

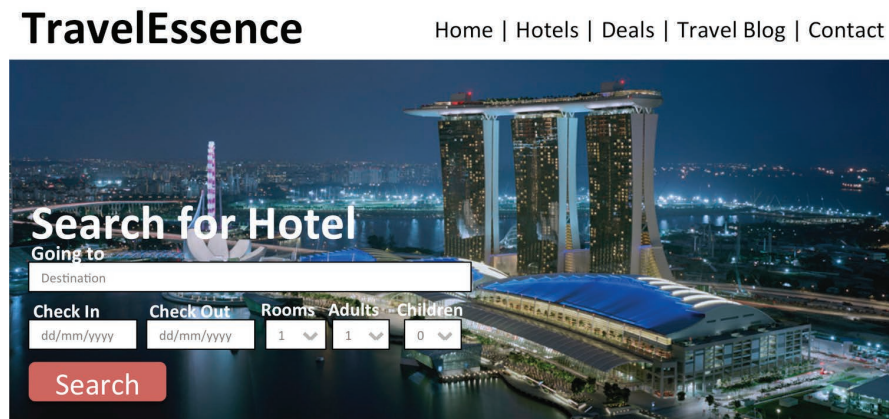


Figure 1.3 TravelEssence home page.

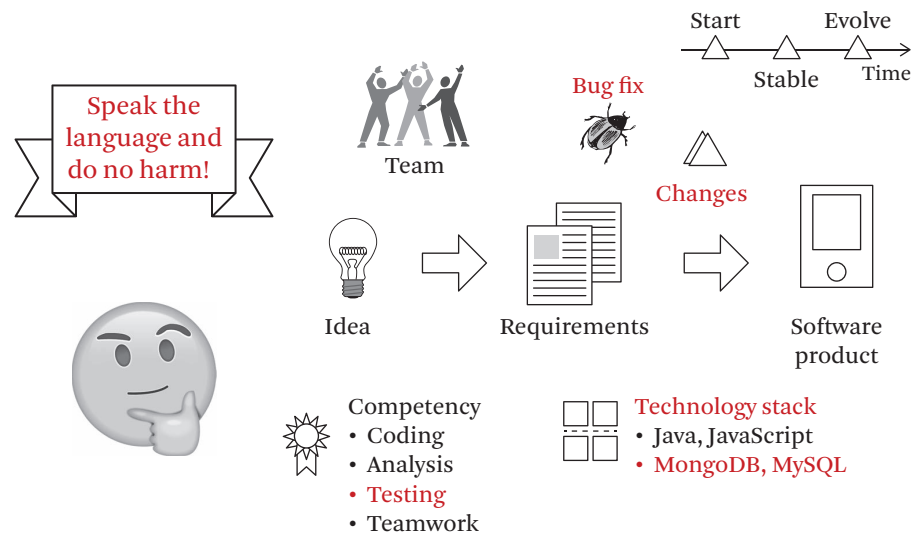


Figure 1.4 What software engineering is from the eyes of a student after internship.

Book. To sell or reserve rooms ahead of time.

No-Show. A guest who made a room reservation but did not check in.

Skipper. A guest who left with no intention of paying for the room.

PMS. Property Inventory Management System, which maintained records of items owned by the hotel such as items in each room including televisions, beds, hairdryers, etc.

POS. Point of Sale Systems (used in restaurants/outlets) that automated the sale of items and managed purchases with credit or debit cards.

It took Smith a little while to get on “speaking terms” with his new colleagues and mentors.

In his student days, Smith always wrote code from scratch, starting with an empty sheet of paper. However, at TravelEssence it was mostly about implementing enhancements to some existing code. The amount of code that Smith saw was way above the toy problems he came across as a student. His development colleagues did not trust him to make any major changes to the system. Developers in TravelEssence emphasized code reviews heavily and stressed the importance of “Do no harm” repeatedly. They would repeatedly test his understanding of terminology and their way of working. Smith felt embarrassed when he could not reply confidently. He started to understand the importance of reviewing and testing his work. After his internship, Smith attempted to summarize what he understood software engineering to be (see Figure 1.4). This was quite similar to what he thought before his internship (see Figure 1.2), but with new knowledge (indicated in red) and an emphasis on testing and doing no harm as he coded changes to the software product. Smith came to recognize the importance of knowledge in different areas, not just about the code, but also about the problem domain (in this case, about hotel management), and the technologies that were being used.

Competency not only involved analysis, coding, and teamwork, but also extensive testing to ensure that Smith did no harm. Understanding programming languages was no longer sufficient; a good working knowledge of the technology stack was critical. A technology stack is the set of software technologies, often called the building blocks, that are used to create a software product. Smith was familiar with multiple technologies that were being used including Java, JavaScript, MongoDB, and MySQL. Never mind if you do not know these specific terms.

Note: There are myriads of technology stacks available, and it is not possible for anyone to learn them all. Nevertheless, our recommendation to students is to gain familiarity with a relevant technology stack of your choice.

Smith graduated and was employed at TravelEssence. A few years later, at a get-together, Smith and his old classmates shared their newfound experiences in the real commercial world. At this occasion Smith said: “At TravelEssence even though everyone seemed to be using different terminology, and everyone did things differently, there seemed to be something common to what they were all doing.” One of his old classmates asked Smith if he could explain more, but Smith just shook his head and said, “I don’t know exactly what it is.”

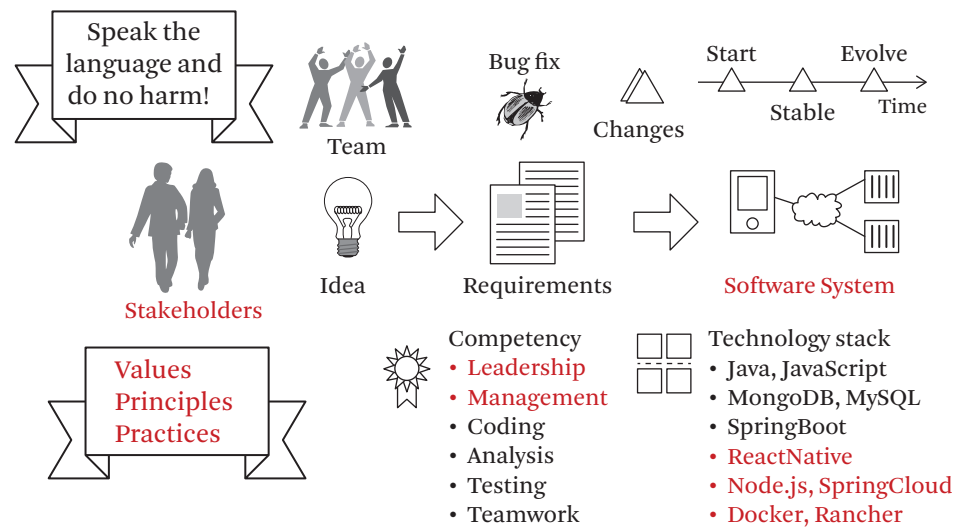


Figure 1.5 What software engineering is from the eyes of a young professional.

Some years later, Smith became a technical lead for a small group at Travel-Essence. As a technical lead he found himself continuously thinking about that discussion with his old classmates as he tried to figure out just what it was that was common about the way everyone worked at TravelEssence.

One evening the old classmates got together again. This time the discussions were a blend between technologies and people management. The old classmates were also talking more about their experiences dealing with people including their colleagues, managers, and their customers; consequently, they were talking more about the way work got done in their organizations. Managing stakeholders and their expectations became more important as they started to take on more senior positions.

After the meeting with classmates, Smith started to draw what he then thought software engineering was about (see Figure 1.5). The changes compared to Smith's internship experience are highlighted in red.

Stakeholder collaboration played an important part of Smith's work. Collaborating well involved having an agreed-on set of values, principles, and practices. These values included agreeing upon a common goal, and respecting and trusting team members, as well as being responsible and dependable. All of these values are qualities of a good and competent team player. Principles include, for instance, having frequent and regular feedback, and fixing bugs as soon as they are detected. All of

these principles identify good behaviors in a team. Practices are specific things the team will do to deliver what is expected of the team consistent with the above values and principles, as well as good quality software.

1.4 Journey into the Software Engineering Profession

Smith through his experience at TravelEssence thus far had started to appreciate the complexities involved in producing and sustaining high-quality software that meets the needs of stakeholders. He now appreciated that while programming is an important aspect, there is much more involved. It is the engineering discipline that is concerned with all aspects of the development and sustainment of software products.

Smith then reflected upon the knowledge he had attained thus far in his career. As a student with no other experience than having done some programming, it is quite difficult to understand what more is involved in software engineering. Typically, when creating a program in a course setting, the exercise starts from an idea that may have been explained in a few words: say, less than one hundred words. Based on the idea, Smith and his classmates developed a piece of software, meaning they wrote code and made sure that it worked. After the assignment they didn't need to take care of it. These assignments were small and to perform them they really did not need much engineering discipline. This situation is quite unlike what you have to do in the industry, where code written will stay around for years, passing through many hands to improve it. Here a sound approach to software engineering is a must. Otherwise, it would be impossible to collaborate and update the software with new features and bug fixes. Nevertheless, the experience in school is an important and essential beginning, even though Smith wished that it were more like the industry.

The authors of this book have all experienced, through their personal journeys, the importance of utilizing an engineering approach in providing high-quality software. Thus, we can characterize, for you, what is important in respect to software engineering.

Considering the software industry, let's put the success of Microsoft, Apple, Google, Facebook, Twitter, etc. on the side because they are so unique—relying on innovative ideas that found a vast commercial market—and programming, per se, was not the root cause of their success. In a more normal situation you will find yourself employed by a company that as part of their mission needs to develop software to support their business or to sell a product needed by potential customers. The company may be rather small or very large, and you will be part of a team. The reasons you won't be alone are many. What needs to be done is more

than what one person can do alone. If the software product is large your team will most likely not be the only one; there will be many teams that have to work in some synchronized way to achieve the objectives of your company.

As a young student having spent most of your life at school and not yet working in the industry, you may be more interested in the technologies related to software—the computer, programming languages, operating systems, etc.—and less interested in the practicalities of developing commercial software for a particular business.

However, this is going to change with this book.

First, let us consider the importance of a team. The team has a role in the company to develop some software. To do that, they need to know what the users of the software need, or in other words they need to agree on the *requirements*. In some cases, they will receive the requirements indicating that they want software that does what another piece of software does. In these cases, the team must study the other product and do something like that product or better. In other situations, someone will just tell them what to do and be with the team while they do it. In more regulated organizations, someone (or a group of people) has written a document specifying what is believed to be some or all of the requirements. Typically, people don't specify all the requirements before starting the development, but some requirements will be input to the team, so they can start doing something to show to the future users of the product. Interacting with users on intermediary results will reveal weaknesses and tell the team what they need to do next. These discussions may imply that the team has to backtrack and redo parts of what they have done and demonstrate the new results to the users. These discussions will also tell the team what more needs to be done.

Anyway, the team will in one way or the other have to understand what requirements they should use as input to the work of their team. Understanding the requirements is normally not trivial. It may take as much time or even more as it takes to program a solution. As we just stated, you will typically have to modify them and sometimes throw away some of the requirements as well as work results before the users of the software are reasonably satisfied with what they have received.

As a newcomer to software engineering but with some background in programming, you may think that working with requirements is less rewarding and less interesting than programming. Well, it is not. There is an entire discipline (*requirements engineering*) that specifies how you dig out the requirements, how you think about them to create great user experiences supported by the software, and how you

modify them to improve and sustain the software. There are requirements management tools to help you that are as interesting to work with as programming tools. There are many books and other publications on how to work with requirements, so there is a lot to learn as you advance in your career. Therefore, working with requirements is one of the things to do that is more than programming but part of software engineering.

Another thing to do that is more than programming is the *design* of the software. Design means structuring the code in such a way that it is easy to understand, easy to change to meet new requirements, easy to test, etc. You can describe your design by using elements of a programming language such as component, class, module, interface, message, etc. You can also use a visual language with symbols for such elements that have a direct correspondence in the programming language you are utilizing. In the latter case, you use a tool to draw diagrams with symbols representing, for instance, components with interfaces. In short, you express the design in a diagram form. The visual language can be quite sophisticated and allow you to not just express your design; for example, you can do quality controls using a visual language tool as well as testing the design to some extent. Doing design is as interesting and rewarding as programming and it is an important part of software engineering.

Apart from working with requirements and creating a design, there are many other things we need to do when we engineer software. We do extensive *testing* of the software; we *deploy* it on a computer so it can be executed and used. If the software we have developed is successful, we will *change* it for many years to come. In fact, most people developing software are engaged in changing existing software that has been developed, often many years ago. This means we need to deal with versions of existing software and if the software has been used at many places (even around the world) we often need to have different versions of the same original software at different locations in the world. Each version will change independent of the other existing versions. And, the complexity of the software product just continues to increase. The only way to deal with this complexity is to use tools specifically designed for its purpose: testing, deployment, version and configuration control, etc.

So, you see that software engineering is certainly much more than programming. While definitions of software engineering are always a subject of debate among professionals, the following neatly summarizes our view. *Software engineering is the application of a systematic, disciplined, and quantifiable approach to the development, testing, deployment, operation, and maintenance of software systems.*

To us, “*a systematic, disciplined, and quantifiable approach*” means it is repeatable and consistent from one project to another, with continuous improvement on the way. It means it is accompanied by some form of description of the way of working and it allows us to automate more. Software engineering includes understanding what users and other stakeholders need and transforming those needs into clear requirements that can be understood by programmers. It also includes understanding the specific technologies needed to build and to test the software. It requires teams that have the social skills to work together, so each piece of the software works with other pieces to achieve the overall goal. So, software engineering encompasses the collaboration of individuals to evolve software to achieve some goal.

Programming is very rewarding since you immediately see the impact of your work. However, as you will learn during your journey, the other activities in software engineering—requirements, design, testing, etc.—are also fascinating for similar reasons. It has been more difficult, though, to teach these other activities in a systematic and generic manner. This is due to the fact that there are so many variations of these activities and there has not been a common ground for teaching them until now as presented in this book. You will find that most students who study in the software domain have an initial desire to work with programming. However, as these people become more and more experienced they gradually move into the other areas of software engineering. This is not because programming is not important. In fact, without programming there is no product to use and sell. No, it is because they find the other areas to be more challenging; also, success in these other areas requires more experience. By essentializing software engineering as presented in this book, the full scope of the discipline will be easier to grasp and to teach.

What Should You Now Be Able to Accomplish?

After studying this chapter, you should be able to:

- explain the terms *programming*, *software development*, and *software engineering*, and how they relate to each other;
- explain the difference between professional programming and hacking;
- understand how teamwork affects the dynamics of software engineering (e.g., importance of code understandability);

- explain the importance of testing as a tool to promote safe modification of existing code;
- understand how people management blends into software engineering and why it is important to consider it;
- explain the role of requirements engineering.

In order to support your learning activities, we invite you to visit www.software-engineering-essentialized.com. There one can find additional material, exercises related to this chapter, and some questions one might encounter in an exam.

In addition to this you will find a short account of the history of software engineering in Appendix [A](#).

References

- Alpha State Card Games. 2018. <https://www.ivarjacobson.com/publications/brochure/alpha-state-card-games>. 99, 153
- S. Ambler and M. Lines. 2012. *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise*. IBM Press. 297, 339, 346
- K. Beck. 1999. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman. 203, 285, 346
- K. Beck. 2003. *Test-Driven Development by Example*. Addison Wesley. 346
- K. Bittner and I. Spence. 2003. *Use Case Modeling*. Addison-Wesley Professional, 2003. 222, 285
- G. Booch, J. Rumbaugh, and I. Jacobson. 2005. *The Unified Modeling Language User Guide*. 2nd edition. Addison-Wesley. 222, 285, 345
- F. Brooks. 1975. *The Mythical Man-Month*. Addison Wesley. 342
- M. Cohn. 2004. *User Stories Applied: For Agile Software Development*. Addison-Wesley Professional. 204, 247, 285
- E. Derby and D. Larsen. 2006. *Agile Retrospectives: Making Good Teams Great*. Pragmatic Bookshelf, Dallas, TX, and Raleigh, NC. 196, 198, 284
- E. W. Dijkstra. 1972. "The Humble Programmer." Turing Award Lecture, *CACM* 15 (10): 859–866. DOI: [10.1145/355604.361591](https://doi.org/10.1145/355604.361591). 342
- D. Graziotin and P. Abrahamsson. 2013. A web-based modeling tool for the SEMAT Essence theory of software engineering. *Journal of Open Research Software*, 1,1(e4); DOI: [10.5334/jors.ad.147](https://doi.org/10.5334/jors.ad.147), 153
- ISO/IEC/IEEE 2382. 2015. Information technology–Vocabulary. International Organization/International Electrotechnical Commission, Geneva, Switzerland. <https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:ed-1:v1:en>. 343
- ISO/IEC/IEEE 12207. 2017. https://en.wikipedia.org/wiki/ISO/IEC_12207 345
- ISO/IEC/IEEE 15288. 2002, 2008, 2015. Systems and software engineering—System life cycle processes. International Standardization Organization/International Electrotechnical Commission, 1 Rue de Varembe, CH-1211 Geneve 20, Switzerland. 345

- ISO/IEC/IEEE 24765. 2017. Systems and software engineering—Vocabulary. International Organization/International Electrotechnical Commission, Geneva, Switzerland. <https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:24765:ed-2:v1:en>. 343
- M. Jackson. 1975. *Principles of Program Design*. Academic Press. 344
- I. Jacobson. 1987. Object-oriented software development in an industrial environment. *Conference Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 87)*. DOI: [10.1145/38807.38824](https://doi.org/10.1145/38807.38824). 221, 285
- I. Jacobson and H. Lawson, editors. 2015. *Software Engineering in the Systems Context*, Systems Series, Volume 7. College Publications, London. 345
- I. Jacobson and E. Seidewitz. 2014. A new software engineering. *Communications of the ACM*, 12(10). DOI: [10.1145/2685690.2693160](https://doi.org/10.1145/2685690.2693160). 347
- I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press Addison-Wesley. 345
- I. Jacobson, I. Spence, and K. Bittner. 2011. Use-Case 2.0: The Guide to Succeeding with Use Cases. <https://www.ivarjacobson.com/publications/whitepapers/use-case-ebook>. 169, 222, 226, 233, 285
- I. Jacobson, P.-W. Ng, P. E. McMahon, I. Spence, and S. Lidman. December 2012. The essence of software engineering: The SEMAT kernel. *Communications of the ACM*, 55(12). <http://queue.acm.org/detail.cfm?id=2389616>. DOI: [10.1145/2380656.2380670](https://doi.org/10.1145/2380656.2380670). 30, 95
- I. Jacobson, P.-W. Ng, P. E. McMahon, I. Spence, and S. Lidman. 2013a. *The Essence of Software Engineering: Applying the SEMAT Kernel*. Addison-Wesley. xxvi, 30, 90, 95, 336
- I. Jacobson, I. Spence, and P.-W. Ng. (October) 2013b. Agile and SEMAT: Perfect partners. *Communications of the ACM*, 11(9). <http://queue.acm.org/detail.cfm?id=2541674>. DOI: [10.1145/2524713.2524723](https://doi.org/10.1145/2524713.2524723). 30, 96
- I. Jacobson, I. Spence, and B. Kerr. 2016. Use-Case 2.0: The hub of software development. *Communications of the ACM*, 59(5): 61–69. DOI: [10.1145/2890778](https://doi.org/10.1145/2890778). 169, 222, 226, 285
- I. Jacobson, I. Spence, and P.-W. Ng. 2017. Is there a single method for the Internet of Things? *Queue*, 15.3: 20. DOI: [10.1145/3106637](https://doi.org/10.1145/3106637). 335, 339
- P. Johnson and M. Ekstedt. 2016. The Tarpit—A general theory of software engineering. *Information and Software Technology* 70: 181–203. https://www.researchgate.net/profile/Pontus_Johnson/publication/278743539_The_Tarpit_-_A_General_Theory_of_Software_Engineering/links/55b4490008aed621de0114f5/The-Tarpit-A-General-Theory-of-Software-Engineering.pdf. DOI: [10.1016/j.infsof.2015.06.001](https://doi.org/10.1016/j.infsof.2015.06.001). 91, 92, 96
- P. Johnson, M. Ekstedt, and I. Jacobson. September 2012. Where's the theory for software engineering? *IEEE Software*, 29(5). DOI: [10.1109/MS.2012.127](https://doi.org/10.1109/MS.2012.127). 84, 87, 96
- R. Knaster and D. Leffingwell. 2017. *SAFe 4.0 Distilled: Applying the Scaled Agile Framework for Lean Software and Systems Engineering*. Addison-Wesley Professional. 297, 339
- P. Kruchten. 2003. *The Rational Unified Process: An Introduction*. 3rd edition. Addison-Wesley. 345

- C. Larman and B. Vodde. 2008. *Scaling Lean & Agile Development: Thinking and Organizational Tools for Large-Scale Scrum*. Pearson Education, Inc. 346
- C. Larman and B. Vodde. 2016. *Large-Scale Scrum: More with LeSS*. Addison-Wesley Professional. 297, 339
- D. Leffingwell. 2007. *Scaling Software Agility: Best Practices for Large Enterprises*. Addison-Wesley. 346
- P. E. McMahon. January/February 2015. A thinking framework to power software development team performance. *Crosstalk, The Journal of Defense Software Engineering*. <http://www.crosstalkonline.org/>. 96, 154
- NATO. 1968. "Software Engineering: Report on a conference sponsored by the NATO Science Committee." P. Naur and B. Randell, editors. Garmisch, Germany, October 7–11. 342
- S. Newman. 2015. *Building Microservices*. O'Reilly Media, Inc. 250, 285
- P.-W. Ng. 2013. Making software engineering education structured, relevant and engaging through gaming and simulation. *Journal of Communication and Computer* 10: 1365–1373. 99, 153
- P.-W. Ng. 2014. Theory based software engineering with the SEMAT kernel: Preliminary investigation and experiences. *Proceedings of the 3rd SEMAT Workshop on General Theories of Software Engineering*. ACM. DOI: 10.1145/2593752.2593756. 30, 96
- P.-W. Ng. 2015. Integrating software engineering theory and practice using Essence: A case study. *Science of Computer Programming*, 101: 66–78. DOI: 10.1016/j.scico.2014.11.009. 96, 152, 154
- Object Management Group. Essence—Kernel and Language for Software Engineering Methods (Essence). <http://www.omg.org/spec/Essence/1.1>. 63
- OMG Essence Specification. 2014. <http://www.omg.org/spec/Essence/Current>. 95, 346
- D. Ross. 1977. Structured Analysis (SA): A language for communicating ideas. In *IEEE Transactions on Software Engineering*, SE-3(1): 16–34. DOI: 10.1109/TSE.1977.229900. 344
- K. Schwaber and J. Sutherland. 2016. "The Scrum Guide. The Definitive Guide to Scrum: The Rules of the Game." Scrum.org.

Index

201 Principles of Software Development,
84–85

Accept a User Story activity, 207, 214–215

Acceptable state in Requirements alpha, 57,
215–217

Acceptance Criteria Captured detail level
for Test Case, 209

Acceptance Criteria Listed detail level for
Story Card, 209

Achieving the Work alpha, 215

ACM (Association for Computing
Machinery), 342

Actionability in Essence kernel, 89

Activities

Essence, 55

Essence kernel, 72–75

Microservices Lite, 255–257, 267–270

Scrum, 175–176, 178

Scrum Lite, 188–197

thing to do, 62–63

Use Case Lite, 229, 238–244

User Story Lite, 207, 211–215, 217–218

Activity spaces

Essence kernel, 68, 72–73

essentializing practices, 63–65

Adapt in Plan-Do-Check-Adapt cycle, 132

Adaptability in microservices, 252

Adapts achievement level in Development
competency, 61–62

Addressed state in Requirements alpha, 57,
80

Agile Manifesto, 335–336

Agility and Agile methods

Agile methods era, 25–26

Essence kernel relationship to, 90–91

introduction, 346

practices and methods, 335–336

All Stories Fulfilled state in Use Case alpha,
231

Alphas

Chasing the State game, 105–108

Checkpoint Construction game, 112–113

composition of practices, 279–281

customer area of concern, 160–163

development, 128–132

development journey, 146–148

Essence, 54–55

Essence kernel, 68–72, 89, 151–152

kick-starting development, 122–123

large and complex development, 311

Microservices Lite, 257–259

Objective Go game, 108–111

overview, 56

Progress Poker game, 100–104

Scrum, 175–177

Scrum Lite, 179–182

states, 55–59

sub-alphas, 124

Use Case Lite, 229–233

User Story Lite, 207–209, 215–216

Alternative practices, 278–279

Ambler, Scott, 346

Analysis competency, 76

- Analyzed state in Use-Case Slice alpha, 233
- Anomalies in development journey, 148
- Application logic, definition, 251
- Applies achievement level in Development competency, 61–62
- Architecture Selected state in Software Systems, 59, 80, 311
- Areas of concern in Essence kernel, 67–68
- Assists achievement level in Development competency, 61–62
- Association for Computing Machinery (ACM), 342
- Attainable attribute in SMART criteria, 196–197
- Automated detail level
 - Build and Deployment Script, 265
 - Test Case, 210
- AXE telecommunication switching system, 344

- Babbage, Charles, 341
- Background in practices, 34
- Backlog-Driven Development practice, 33
- Beck, Kent, 86, 346
- Booch, Grady, 345
- Bounded state in Requirements alpha, 56, 80, 215–216
- Briefly Described detail level in Use-Case Narrative work product, 235
- Brooks, Frederick P., 84, 342
- Build and Deployment Script, 264–265
- Building blocks, 10
- Bulleted Outline detail level in Use-Case Narrative work product, 235
- Bureau of the Census, 341

- Capabilities in practices, 33–34
- Capability Maturity Model Integration (CMMI), 306
- Capacity Described work product, 183
- Card games
 - Chasing the State, 105–108
 - Checkpoint Construction, 111–113
 - Objective Go, 108–111
 - overview, 97–99
 - Progress Poker, 99–105
 - reflection, 113
- Cards
 - alphas, 57–58
 - Essence, 38
 - user stories, 204
- Census, 341
- Chasing the State game, 105–108
- Check in Plan-Do-Check-Adapt cycle, 131–132
- Checking in Essence, 138–139
- Checkpoint Construction game, 111–113
- Checkpoint pattern, 78–80
- Checkpoints
 - kick-starting development, 122–123
 - kick-starting development with practices, 159–165
 - large and complex development, 310
- Cloud computing for microservices, 252
- CMMI (Capability Maturity Model Integration), 306
- COBOL programming language, 342
- Code
 - thing to work with, 54, 56
 - work product cards, 60
- Coder role pattern card, 79
- Coherent state in Requirements alpha, 57, 215–216
- Collaboration
 - importance, 11–12
 - Scrum, 165–166, 174
- Collaborations and Interfaces Defined detail level in Design Model work product, 261
- Common ground in Essence, 34–37
- Competencies
 - Essence, 55
 - Essence kernel, 68, 75–77
 - programming, 61–62
 - testing in, 10
- Compilers, 341–342
- Complete state in Microservices alpha, 258–259

- Completed PBIs Listed work product, 184
- Complex development. *See* Large and complex development
- Component methods, 22–25
- Component paradigm, 344–345
- Composition of practices
 - description, 276–282
 - Essence, 282–284
 - overview, 275–276
 - reflection, 282–283
- Conceived state in Requirements alpha, 56, 311
- Confirmation in user stories, 205
- Consensus-based games
 - Chasing the State, 105–108
 - Checkpoint Construction, 111–113
 - Objective Go, 108–111
 - Progress Poker, 99–105
 - reflection, 113
- Containers definition, 251
- Context in kick-starting development, 118–121
- Continual improvement in large and complex development, 321–323
- Continuous detail level in Build and Deployment Script, 265
- Conversation Captured detail level in Story Card, 209
- Conversations in user stories, 205
- Coordination in activity space, 74
- Culture issues, 330–331
- Customer area of concern
 - alphas, 160–163
 - competencies, 76
 - development perspective, 119–120
 - development process, 139
 - Essence kernel, 68–70
- Customer-related practices, 19
- Customers
 - description, 42–43
 - value for, 43–44
- DAD (Disciplined Agile Delivery)
 - agile scaling, 27
 - introduction, 346
 - monolithic methods, 297
 - practices from, 296
- Daily Scrum activity
 - description, 173, 178
 - diagram, 175–176
 - overview, 192–193
- Daily Standup practice in Scrum, 26, 33, 173
- Data in structured methods era, 21–22
- Data processing focus, 341
- Data stores, definition, 251
- Davis, Alan, 84–85
- Definition of Done (DoD) in Scrum, 176–177
- Demonstrable alpha state, 59, 100
- Deployment in activity space, 74
- Descriptive theory of software engineering, 87–88
- Design Model work product, 254, 257, 260–263
- Design overview, 14
- Design Patterns Identified detail level in Design Model work product, 262
- Design phase
 - iterative method, 21
 - waterfall method, 19–20
- Detail levels
 - Build and Deployment Script, 265
 - Microservice Design work product, 264
 - Story Card, 209
 - Test Case, 210
 - Use Case Lite work products, 233–236
 - Use-Case Narrative work product, 235
 - Use-Case Slice Test Case work product, 237
 - work products, 60–61
- Developers
 - Scrum, 173
 - Tarpit theory, 92
- Development
 - doing and checking, 138–139
 - kick-starting. *See* Kick-starting development; Kick-starting development with practices
 - overview, 127–132

- Development (*continued*)
 - Plan-Do-Check-Adapt cycle, 128–132
 - plans, 132–138
 - way of working, 140–142
- Development competency, 61–62, 77
- Development Complete checkpoint, 80
- Development endeavor, 79–80
- Development journey
 - anomalies, 148
 - overview, 145
 - progress and health, 146–148
 - visualizing, 145–146
- Development types
 - culture issues, 330–331
 - overview, 325
 - practice and method architectures, 326–328
 - practice libraries, 328–330
- DevOps practice, 302
- Dijkstra, E. W., 86, 342–343
- Disciplined Agile Delivery (DAD)
 - agile scaling, 27
 - introduction, 346
 - monolithic methods, 297
 - practices from, 296
- Disciplined approach in software engineering, 14–15
- Do in Plan-Do-Check-Adapt cycle, 131
- Document elements in Essence, 54
- DoD (Definition of Done) in Scrum, 176–177
- Doing alpha in PBIs, 181
- Doing in development, 138–139
- Done alpha in PBIs, 181
- Done term, definition, 99–100
- EA (enterprise architecture), 24
- Endeavor area of concern
 - competencies, 77
 - development perspective, 120–121
 - development process, 136–139
 - Essence kernel, 68, 70–71
 - kick-starting development with practices, 163–165
 - practices, 19
 - Scrum, 199
- Endeavors
 - description, 42–43
 - teams, 48–49
 - ways of working in, 49–50
 - work in, 49
- Engaging user experiences, 37–38
- Enterprise architecture (EA), 24
- Ericsson AB, 344
- Essence
 - common ground, 34–37
 - composition of practices, 282–284
 - development. *See* Development
 - development journey. *See* Development journey
 - engaging user experiences, 37–38
 - essentializing practices, 63–65
 - essentials focus, 37
 - evolution, 346
 - insights, 32
 - kick-starting development. *See* Kick-starting development
 - language, 54–61
 - large and complex development, 310–311, 322–324
 - methods and practices, 32–34
 - microservices, 252–256
 - OMG standard, 29–30
 - overview, 31
 - practices, 298–299
 - purpose, 42
 - Scrum with, 174–179
 - serious games. *See* Serious games
 - theory of software engineering, 87–91
 - Use Case Lite practice, 227–230
 - User Story Lite practice, 207–208
 - work products, 60
- Essence kernel
 - actionability, 89
 - activities, 72–75
 - alphas, 68–72
 - applying, 151–152
 - competencies, 75–77
 - extensibility, 90

- growth from, 93–94
 - observations, 151
 - organizing with, 67–69
 - overview, 67
 - patterns, 77–80
 - practicality, 88–89
 - relationship to other approaches, 90–91
 - User Story Lite practice, 215–218
 - validity, 151
- Essential Outline detail level in Use-Case
 - Narrative work product, 235
- Essentialized practices, 35–36
- Essentializing practices
 - composition of practices, 283–284
 - description, 35–36
 - Essence, 298–299
 - for libraries, 329
 - monolithic methods and fragmented practices, 296–298
 - overview, 63–65
 - reusable, 299–302
 - sources, 295–296
- Estimatable criteria in user stories, 205–206
- Evolve Microservice activity, 255, 257, 269–270
- Exchangeable packages, 345
- Explicit approaches in Scrum, 173–174
- Extensibility
 - Essence kernel, 90
 - software systems, 47
- Extension practices, 279
- Extreme Programming Explained*, 86
- Extreme Programming (XP)
 - introduction, 346
 - practices from, 296
 - user stories, 203
- Feedback in Use Case Lite practice, 239
- Find Actors and Use Cases activity, 229, 238–239
- Find User Stories activity, 207, 212
- Formed state in Teams, 311
- Fortran programming language, 342
- Foundation Established state in Way of working, 311
- Fragmented practices, 296–298
- Fulfilled alpha state, 57
- Fully Described detail level in Use-Case
 - Narrative work product, 235
- Function-data paradigm, 344
- Functionality in software systems, 46
- Functions in structured methods era, 21–22
- Future, dealing with
 - agility, 335–336
 - methods evolution, 338–339
 - methods use, 337–338
 - overview, 333–335
 - teams and methods, 337
- Games
 - Chasing the State, 105–108
 - Checkpoint Construction, 111–113
 - Objective Go, 108–111
 - overview, 97–99
 - Progress Poker, 99–105
 - reflection, 113
- General predictive theory of software engineering, 91–92
- “Go to statement considered harmful”
 - article, 86
- Goal Established state in Use Case alpha, 230
- Goals Specified work product, 183
- Gregor, Shirley, 84–85
- Hacking vs. programming, 6
- Handle favorites use-case slice, 242–243
- Happy day scenarios, 224
- Health and progress
 - development journey, 146–148
 - Essence, 54
 - Microservices Lite, 271–272
 - use-case slices, 245–246
- Hemdal, Göran, 344
- Higher-level languages, 342
- History of software and software engineering, 341–347

- Hollerith punched card equipment, 341
- Hopper, Grace Murray, 341–342
- Identification of microservices, 251–252
- Identified state
 - Microservices Lite, 258
 - user stories, 209
- Identify Microservices activity, 255, 257, 267–268
- IEEE (Institute of Electrical and Electronic Engineering), 342
- Implementation phase
 - activity space, 74
 - iterative method, 21
 - waterfall method, 19–20
- Implemented state in Use-Case Slice alpha, 233
- In Progress state in user stories, 209
- Increment elements
 - description, 177
 - work products, 183–184
- Increment Notes Described work product, 184
- Incremental development in use cases
 - slices, 226–227
- Independent criteria in user stories, 205
- Innovates achievement level in Development competency, 61–62
- Institute of Electrical and Electronic Engineering (IEEE), 342
- Interfaces Specified detail level in
 - Microservice Design work product, 264
- Internal Elements Designed detail level in
 - Microservice Design work product, 264
- Internal Structure Defined detail level in
 - Microservice Design work product, 264
- INVEST criteria for user stories, 205–206
- ISO/IEC 12207 standard, 345
- Items Ordered work product, 182
- Iterative operations
 - development, 127
 - development journey, 147
 - large and complex development, 319–321
 - lifecycle methods, 20–21
- Jackson, Michael, 344
- Jackson Structured Programming (JSP), 344
- Jacobson, Ivar
 - component paradigm, 344
 - method prison governing, 27
 - OMG, 345
 - RUP, 345
 - SEMAT, 28, 346
 - Use-Case Driven Development practice, 221
- JSP (Jackson Structured Programming), 344
- Kernel. *See* Essence kernel
- Key elements of software engineering
 - basics, 41–43
 - endeavors, 48–50
 - overview, 41
 - value for customers, 43–45
 - value through solutions, 45–48
- Kick-starting development
 - context, 118–121
 - overview, 117–118
 - scope and checkpoints, 122–123
 - things to watch, 124–126
- Kick-starting development with practices
 - context, 158–159
 - overview, 157–158
 - practices to apply, 165–167
 - scope and checkpoints, 159–165
 - things to watch, 167–169
- Kruchten, Philippe
 - method prison governing, 27
 - RUP, 345
- Language of software engineering
 - competencies, 61–62
 - essentializing practices, 63–65
 - overview, 53
 - practice example, 53–54
 - things to do, 62–63

- things to work with, 54–61
- Large and complex development
 - alphas, 311
 - common vision, 315–317
 - continual improvement, 321–323
 - Essence, 310–311, 322–324
 - iterative operations, 319–321
 - kick-starting, 309–315
 - large-scale development, 308–309
 - large-scale methods, 306–308
 - managing, 317–319
 - overview, 305–306
 - practices, 310–313
 - running, 315–322
 - scope and checkpoints, 310
 - things to watch, 313–315
- Large-scale integrated circuits, 343
- Large-Scale Scrum (LeSS)
 - agile scaling, 27
 - introduction, 346
 - monolithic methods, 297
 - practices from, 296
- Larman, Craig, 346
- Lawson, Harold “Bud,” 345
- Leadership competency, 77
- Leffingwell, Dean, 346
- LeSS (Large-Scale Scrum)
 - agile scaling, 27
 - introduction, 346
 - monolithic methods, 297
 - practices from, 296
- Levels of detail
 - Build and Deployment Script, 265
 - Microservice Design work product, 264
 - Story Card, 209
 - Test Case, 210
 - Use Case Lite work products, 233–236
 - Use-Case Narrative work product, 235
 - Use-Case Slice Test Case work product, 237
 - work products, 60–61
- Libraries for practices, 328–330
- Lifecycles, 19–21
- Lines, Mark, 346
- Lovelace, Ada, 341
- Machine instruction level, 341
- Make Evolvable activity, 255, 257, 268–269
- Management competency, 77
- Martin, Robert, 90
- Masters achievement level in Development
 - competency, 61–62
- Mayer, Bertrand, 28
- Measurable attribute in SMART criteria, 196–197
- Method prison, 27
- Methods
 - agile methods era, 25
 - component methods era, 22–25
 - consequences, 26–28
 - definition, 19
 - Essence, 32–34
 - evolution, 338–339
 - large-scale, 306–308
 - lifecycles, 19–21
 - people practices, 25–26
 - rise of, 18–19
 - structured methods era, 21–22
 - team ownership, 337
 - technical practices, 21–25
 - use focus, 337–338
- Methods war, 22, 26–27
- Meyer, Bertrand, 346
- Microprocessors, 343
- Microservice alpha, 254, 257
- Microservice Build and Deployment work
 - product, 254, 257
- Microservice Design work product, 254, 257, 263–264
- Microservice Test Case work product, 255, 257, 265–267
- Microservices, 166–169
 - description, 250–252
 - Essence, 252–256
 - overview, 249–250
- Microservices Lite practice
 - activities, 255–256, 267–270
 - alphas, 257–259

- Microservices Lite practice (*continued*)
 - Build and Deployment Script, 264–265
 - description, 256–257
 - design model, 260–263
 - impact, 270–271
 - Microservice Design work product, 263–264
 - Microservice Test Case work product, 265–267
 - overview, 253–256
 - progress and health, 271–272
 - reusable practices, 299–300
 - work products, 259–267
- Mini-computers, 343
- Mini-methods, 19
- Minimal state in Microservices Lite, 258
- Modular approaches in Scrum, 173–174
- Monolithic methods, 296–298
- Mythical Man-Month*, 84
- NATO-sponsored conference, 342
- Negotiable criteria in user stories, 205
- NZ Transport Agency, 18
- Object Management Group (OMG) standard
 - Essence, 29–30, 346
 - Essence kernel, 71
 - notation, 23–24
 - UML standard, 345
- Object-oriented programming
 - acceptance, 344–345
 - components in, 23
- Objective Go game, 108–111
- On the Criteria to Be Used in Decomposing Systems into Modules*, 86
- Operational alpha state, 59
- Opportunity
 - alpha state card, 72
 - customer area of concern, 69, 71
 - development context, 158
 - development endeavors, 42–43
 - development perspective, 119–120
 - development plans, 133–134
 - large and complex development, 311–312
 - scope and checkpoints, 161–162
 - value for customers, 43–44
- Outlined detail level in Build and Deployment Script, 265
- Pair programming teams, 26
- Paradigm shifts, 22–23
- Paradigmatic theories, 85
- Paths in use cases slices, 228
- Patterns
 - Essence kernel, 68, 77–80
 - essentializing practices, 63–65
 - Scrum, 178–179, 184–186
- PBIs. *See* Product Backlog Items (PBIs)
- People practices, 25–26
- Performance in software systems, 47
- Perlis, Alan, 92
- PLA (product-line architecture), 24
- Plan-Do-Check-Adapt cycle, 128–132
- Planned alpha in sprints, 179
- Plans
 - development, 132–138
 - Plan-Do-Check-Adapt cycle, 128–131
 - Scrum Lite, 188–192
- POs (product owners)
 - description, 178
 - pattern cards, 184–185
 - Scrum, 172–173, 175
- Possibilities in activity space, 73
- Post-development phase in development endeavor, 79–80
- Practicality in Essence kernel, 88–89
- Practice separation in Essence kernel, 90
- Practices
 - agile methods era, 25
 - background, 34
 - capabilities, 33–34
 - common ground, 34–37
 - component methods era, 22–25
 - composition of. *See* Composition of practices
 - consequences, 26–28
 - definition, 174
 - Essence, 32–34, 298–299

- fragmented practices, 296–298
- kick-starting development with. *See* Kick-starting development with practices
- large and complex development, 310–313
- libraries, 328–330
- lifecycles, 19–21
- people, 25–26
- reusable, 299–302
- rise of, 18–19
- Scrum, 173–174, 177, 198–199
- sources, 295–296
- structured methods era, 21–22
- technical, 21–25
- types, 19
- Pre-development phase in development endeavor, 79–80
- Precision in Scrum, 200–202
- Preparation in activity space, 74
- Prepare a Use-Case Slice activity, 229, 242–243
- Prepare a user story activity, 207, 212–213
- Prepared state
 - Use-Case Slice alpha, 232–233
 - Work alpha, 215
- Priorities in Scrum, 172
- Problems in kick-starting development, 118
- Product Backlog Items (PBIs)
 - alphas, 181
 - description, 172, 177
 - example, 176
 - identifying, 173
 - Scrum, 168
- Product Backlog practice, 302
- Product Backlog work product
 - activity cards, 190–192
 - description, 177
 - Scrum Lite, 182–184
- Product-line architecture (PLA), 24
- Product Management practice, 302
- Product owners (POs)
 - description, 178
 - pattern cards, 184–185
 - Scrum, 172–173, 175
- Product Ownership practice, 301
- Product Retrospective practice, 302
- Product Sprint practice, 302
- Program backlog management, 318
- Program practices, 302–303
- Programming, defined, 4
- Programming and software engineering
 - differences, 6–8
 - intern view, 8–10
 - overview, 3–4
 - professional view, 10–12
 - programming, 4–6
 - software engineering, 12–15
- Progress and health
 - activity space, 74–75
 - development journey, 146–148
 - Essence, 54
 - Microservices Lite practice, 271–272
 - use-case slices, 245–246
- Progress Poker game
 - benefits, 102
 - example, 103–105
 - overview, 99–102
- Progressing
 - use-case slices, 232–233
 - use cases, 230–232
- Provided interface, UML notation for, 261
- Quality in software systems, 47–48
- Quantifiable approach in software engineering, 14–15
- Rapidly Deployable state in Microservices Lite practice, 258
- Rational Unified Process (RUP)
 - development of, 24, 345
 - large-scale development, 306
 - monolithic methods, 297
- Reaching out in scaling, 293
- Ready for Development checkpoint, 80
- Ready for Development state in User Story, 209
- Ready requirement, 80

- Ready state
 - PBIs, 181
 - Software Systems, 59
- Recognized state for Stakeholders, 311
- Relevant attribute in SMART criteria, 196–197
- Reliability in software systems, 47
- Required Behavior Defined detail level in
 - Microservice Design work product, 264
- Required interface, UML notation for, 261
- Requirements
 - activity space, 74
 - alpha state card, 72
 - alphas, 56–58
 - development context, 158
 - development perspective, 120
 - development plans, 134–135
 - large and complex development, 311–312
 - Ready for Development checkpoint, 80
 - scope and checkpoints, 161–162
 - solution area of concern, 70
 - in solutions, 42–43, 45–46
 - thing to work with, 54–56
 - User Story Lite practice, 225, 227, 230
- Requirements alpha
 - Progress Poker game, 100–101
 - User Story Lite practice, 215–217
- Requirements engineering, 13–14
- Requirements phase
 - iterative method, 21
 - waterfall method, 19–20
- Retired alpha state, 59
- Retrospective practice in Scrum, 33
- Reusable practices, 19, 299–302
- Reviewed alpha in sprints, 179–180
- Roles in Scrum Lite, 184–186
- Roles pattern, 77–78
- Ross, Douglas, 344
- Royce, Walker, 345
- Rumbaugh, James, 345
- RUP (Rational Unified Process)
 - development of, 24, 345
 - large-scale development, 306
 - monolithic methods, 297
- SA/SD (Structured Analysis/Structured Design), 21
- SaaS (Software as a Service), 8
- SADT (Structured Analysis and Design Technique)
 - description, 21–22
 - development of, 344
- Scaled Agile Framework (SAFe)
 - agile scaling, 27
 - introduction, 346
 - monolithic methods, 297
 - practices from, 296
- Scaled Professional Scrum (SPS)
 - agile scaling, 27
 - introduction, 346
 - practices from, 296
- Scaling
 - challenges, 289–291
 - dimensions of, 291–294
 - large and complex development. *See* Large and complex development
 - overview, 289
 - reaching out, 293
 - scaling up, 292–293
 - zooming in, 291–292
- Scenario Chosen detail level in Use-Case
 - Slice Test Case work product, 237
- Scenarios in use cases slices, 228
- Scheduled alpha in sprints, 179
- Schwaber, Ken, 346
- Scope
 - kick-starting development, 122–123
 - kick-starting development with practices, 159–165
 - large and complex development, 310
- Scoped state in Use-Case Slice alpha, 232
- Scripted detail level in Test Case, 210
- Scripted or Automated detail level in Use-Case
 - Case Slice Test Case work product, 237
- Scrum
 - collaboration, 165–166, 174

- components, 33
- composite practices, 306–307
- description, 168
- with Essence, 174–179
- fragmented practices, 297
- introduction, 346
- overview, 171–173
- practices, 173–174, 198–199, 296
- precision, 200–202
- reflections, 198–202
- Scrum Lite
 - activities, 188–197
 - alphas, 179–182
 - overview, 174–177
 - planning, 188–192
 - roles, 184–186
 - usage, 187–188
 - work products, 182–184
- Scrum Masters
 - description, 173, 178–179
 - large and complex development, 321–322
 - pattern cards, 184–186
 - patterns, 175
- Scrum of Scrums meetings, 320
- Scrum Teams
 - description, 179
 - Essence, 175
 - pattern cards, 185–186
- SDL (Specification and Description Language), 344
- Self-organizing teams, 26
- SEMAT (Software Engineering Method And Theory)
 - description, 28–29
 - founding, 346
- Serious games
 - Chasing the State, 105–108
 - Checkpoint Construction, 111–113
 - Objective Go, 108–111
 - overview, 97–99
 - Progress Poker, 99–105
 - reflection, 113
- Service-oriented architecture (SOA), 24
- Simplest Story Fulfilled state in Use Case
 - alpha, 231
- Simula 67 language, 23
- Slice the Use Cases activity
 - description, 229
 - working with, 241–242
- Slicing use cases, 226–227
- Small attribute
 - SMART criteria, 196–197
 - user stories, 206
- Smalltalk language, 23
- SMART criteria, 196–197
- “So that” clauses in user stories, 206
- SOA (service-oriented architecture), 24
- Social issues, 330–331
- Software as a Service (SaaS), 8
- Software crisis, 18, 343
- Software development, defined, 4
- Software Engineering Method And Theory (SEMAT)
 - description, 28–29
 - founding, 346
- Software engineering overview
 - challenges, 17–18
 - defined, 4–5, 14–15
 - history, 341–347
 - key elements. *See* Key elements of software engineering
 - language. *See* Language of software engineering
 - methods and practices, 18–28
 - OMG standard, 29–30
 - and programming. *See* Programming and software engineering
 - SEMAT initiative, 28–29
 - Tarpit theory, 92
 - theory, 84–87
- Software Life Cycle Processes, 345
- Software Systems
 - alpha cards, 58, 72
 - Demonstrable alpha state card, 100
 - development context, 158
 - development perspective, 120
 - development plans, 135–136

- Software Systems (*continued*)
 - large and complex development, 311–312
 - Objective Go game, 109–111
 - scope and checkpoints, 161, 162
 - solutions, 42–43, 45–48, 70
 - thing to work with, 54–56
- Soley, Richard, 28, 346
- Solution area of concern
 - competencies, 76–77
 - development perspective, 120
 - development process, 139
 - Essence kernel, 68–70
 - kick-starting development with practices, 161
- Solution-related practices, 19
- Solutions
 - description, 42–43
 - value through, 45–48
- Specification and Description Language (SDL), 344
- Splitting User Stories activity, 207, 213–214
- Sprint Backlog
 - activity cards, 190–191
 - description, 177
 - PBIs, 172
 - work products, 183
- Sprint Planning activity
 - activity cards, 188–192
 - description, 178
- Sprint Retrospective activity
 - activity cards, 195–196
 - Scrum, 178
- Sprint Review activity
 - activity cards, 193–195
 - description, 172, 178
- Sprints
 - alphas, 179–181
 - description, 177
 - Scrum, 172–173
- SPS (Scaled Professional Scrum)
 - agile scaling, 27
 - introduction, 346
 - practices from, 296
- Stakeholder alpha in Chasing the State game, 105–107
- Stakeholder Representation competency, 76
- Stakeholders
 - activity space, 74
 - alpha state card, 72
 - customer area of concern, 69, 71
 - as customers, 42–43
 - development context, 158
 - development perspective, 119
 - development plans, 133
 - large and complex development, 311–312
 - Objective Go game, 108–111
 - scope and checkpoints, 159–160
 - value for, 44–45
- Started state in Work alpha, 311
- States in alphas, 55–59
- Stored program computers, 341
- Story Card work product, 207, 209–210
- Story practice, 166
- Story Structure Understood state in Use Case alpha, 231
- Structure and Approach Described detail level in Design Model work product, 260
- Structured Analysis and Design Technique (SADT)
 - description, 21–22
 - development of, 344
- Structured Analysis/Structured Design (SA/SD), 21
- Structured detail level in Use-Case Model work product, 235
- Structured methods era, 21–22
- Student Pairs pattern card, 78
- Sub-alphas, 124
- Subsystems in UML notation, 261
- Sufficient Stories Fulfilled state in Use Case alpha, 231
- Support in activity space, 74–75
- Sutherland, Jeff, 346
- SWEBOK, 84–85
- System Boundary Established detail level in Use-Case Model work product, 234
- Systematic approach in software engineering, 14–15

- Tarpit theory, 91–92
- TD (test-driven development) in Essence, 36
- TDD (Test-Driven Development) in Extreme Programming, 346
- Team Backlog practice, 301
- Team Retrospective practice
 - description, 301
 - large and complex development, 321–322
- Team Sprint practice, 301
- Teams
 - activity space, 74–75
 - agile, 26
 - alpha state card, 72
 - development perspective, 120
 - development plans, 136–137
 - endeavor area of concern, 42–43, 48–49, 70–71
 - Essence, 36
 - large and complex development, 311–312
 - methods ownership, 337
 - need for, 12–13
 - scope and checkpoints, 163–164
- Technical practices, 21–25
- Technology stacks, 10
- Test a Use-Case Slice activity
 - description, 229
 - working with, 243–244
- Test Automated detail level in Microservice
 - Test Case work product, 267
- Test Case work product, 207, 209–210
- Test Dependencies Managed detail level in Microservice Test Case work product, 266
- Test-driven development (TD) in Essence, 36
- Test-Driven Development (TDD) in Extreme Programming, 346
- Test Scenarios Chosen detail level in Microservice Test Case work product, 266
- Testable attribute
 - SMART criteria, 196–197
 - user stories, 206
- Testing
 - activity space, 74
 - waterfall method phase, 19–20
- Testing competency, 10, 77
- Theory
 - arguments, 85–87
 - Essence, 87–91
 - general predictive theory, 91–92
 - growth from, 93–94
 - overview, 83–84
 - software engineering, 84–87
 - uses, 87
- Things to do
 - activities, 62–63
 - backlogs, 49
 - composition, 279
 - Essence kernel, 72–75
- Things to watch
 - kick-starting development, 124–126
 - kick-starting development with practices, 167–169
 - large and complex development, 313–315
- Things to work with
 - alpha states, 56–59
 - alphas, 56
 - Essence kernel, 69–72, 89
 - overview, 54–56
 - Use Case Lite practice, 230–234
 - work products, 59–61
- To Do alpha in PBIs, 181
- Turing tar-pit, 92
- UCDD (Use-Case Driven Development)
 - practice, 221–222
- Unified Modeling Language (UML) standard
 - development of, 24
 - introduction, 345
 - Microservices Lite practice, 260–261
 - primer, 260
 - use cases, 222–223
- Unified Process prison, 27
- Unified Process (UP), 24, 345
- Univac I computer, 341
- University of Wisconsin, 18
- UP (Unified Process), 24, 345
- Usable alpha state, 59
- Use Case alpha, 229–231

- Use-Case diagrams, 24
- Use-Case Driven Development (UCDD)
 - practice, 221–222
- Use Case Lite practice
 - activities, 238–244
 - alphas, 229–233
 - Essence, 227–230
 - impact, 244–245
 - kick-starting, 237–240
 - overview, 221–222
 - reusable practices, 299–300
 - use-case slices progress and health, 245–246
 - use cases description, 222–226
 - use cases slicing, 226–227
 - user stories vs. use cases, 246–248
 - work products, 233–236
 - working with, 240–244
- Use-Case Model work product, 227, 229, 234–235
- Use-Case Narrative work product, 227, 229, 235–236
- Use-case narratives, 224–225
- Use case practices, 166, 168–169
- Use-Case Slice alpha, 229, 232–233
- Use-Case Slice Test Case work product, 227, 229, 236–237
- Use-case slices
 - process, 226–227
 - progress and health, 245–246
- Use Cases
 - introduction, 345
 - practices from, 296
- User experiences in Essence, 37–38
- User interface, definition, 251
- User stories
 - description, 204–207
 - Scrum teams, 166
- User Stories practice
 - description, 168–169
 - vs. use cases, 246–248
- User Story alpha in User Story Lite practice, 207–208
- User Story for Extreme Programming, 346
- User Story Lite practice
 - activities, 211–215
 - alphas, 207–209
 - Essence, 207–208
 - Essence kernel, 215–218
 - impact, 216–218
 - overview, 203
 - usage, 211
 - user story description, 204–207
 - work products, 209–210
- Validity in Essence kernel, 151
- Valuable criteria in user stories, 205
- Value
 - for customers, 43–45
 - through solutions, 45–48
- Value Established detail level in Use-Case Model work product, 234
- Value Established state in Opportunity, 311
- Value Expressed detail level in Story Card, 209
- Variables Identified detail level in Use-Case Slice Test Case work product, 237
- Variables Set detail level in Use-Case Slice Test Case work product, 237
- Verification phase
 - iterative method, 21
 - waterfall method, 19–20
- Verified state
 - Use-Case Slice alpha, 233
 - user stories, 209
- Vodde, Bas, 346
- von Neumann, John, 341
- Waterfall method
 - description, 19–20
 - development of, 344
- Way of working
 - adapting, 140–141
 - alpha state card, 72
 - development context, 158
 - development perspective, 120–121
 - development plans, 138

- endeavor area of concern, 42–43, 49–50, 71
- Essence kernel, 141–142
- large and complex development, 311–312
- scope and checkpoints, 163, 165
- “Where’s the Theory for Software Engineering?” paper, 84
- Work activity
 - alpha state card, 72
 - development context, 158
 - development perspective, 120
 - development plans, 136–137
 - endeavor area of concern, 42–43, 49, 71
 - large and complex development, 311–312
 - scope and checkpoints, 163–164
- Work alpha, 215–216
- Work Forecast Described work product, 183
- Work products
 - Essence, 54–55
 - Microservices Lite practice, 259–267
 - overview, 59–61
 - Scrum, 175, 177
 - Scrum Lite, 182–184
 - Use Case Lite practice, 229, 233–236
 - User Story Lite practice, 207, 209–210
- Write Code activity cards, 62–63
- XP (Extreme Programming)
 - introduction, 346
 - practices from, 296
 - user stories, 203
- Zooming in in scaling, 291–292

Author Biographies

Ivar Jacobson



Dr. Ivar Jacobson received his Ph.D. in computer science from KTH Royal Institute of Technology, was awarded the Gustaf Dalén medal from Chalmers in 2003, and was made an honorary doctor at San Martin de Porres University, Peru, in 2009. Ivar has both an academic and an industry career. He has authored ten books, published more than a hundred papers, and is a frequent keynote speaker at conferences around the world.

Ivar Jacobson is a key founder of components and component architecture, work that was adopted by Ericsson and resulted in the greatest commercial success story ever in the history of Sweden (and it still is). He is the creator of use cases and Objectory—which, after the acquisition of Rational Software around 2000, resulted in the Rational Unified Process, a popular method. He is also one of the three original developers of the Unified Modeling Language. But all this is history. His most recently founded company, Ivar Jacobson International, has been focused since 2004 on using methods and tools in a smart, superlight, and agile way. Ivar is also a founder and leader of a worldwide network, SEMAT, whose mission is to revolutionize software development based on a kernel of software engineering. This kernel has been realized as a formal standard called Essence, which is the key idea described in this book.

Harold “Bud” Lawson



Professor Emeritus Dr. Harold “Bud” Lawson (The Institute of Technology at Linköping University) has been active in the computing and systems arena since 1958 and has broad international experience in private and public organizations as well as academic environments. Bud contributed to several pioneering efforts in hardware and software technologies. He has held professorial appointments at several universities in the USA, Europe, and the Far East. A Fellow of the ACM, IEEE, and INCOSE, he was also head of the Swedish delegation to ISO/IEC JTC1 SC7 WG7 from 1996 to 2004 and the elected architect of the ISO/IEC 15288 standard. In 2000, he received the prestigious IEEE Computer Pioneer Charles Babbage medal award for his 1964 invention of the pointer variable concept for programming languages. He has also been a leader in systems engineering. In 2016, he was recognized as a Systems Engineering Pioneer by INCOSE. He has published several books and was the coordinating editor of the “Systems Series” published by College Publications, UK.

Tragically, Harold Lawson passed away after battling an illness for almost a year, just weeks before the publication of this book.

Pan-Wei Ng



Dr. Pan-Wei Ng has been helping software teams and organizations such as Samsung, Sony, and Huawei since 2000, coaching them in the areas of software development, architecture, agile, lean, DevOps, innovation, digital, Beyond Budgetings, and Agile People. Pan-Wei firmly believes that there is no one-size-fits-all, and helps organizations find a way of working that suits them best. This is why he is so excited about Essence and has been working with it through SEMAT since their inception in 2006, back when Essence was a mere

idea. He has contributed several key concepts to the development of Essence.

Pan-Wei coauthored two books with Dr. Ivar Jacobson and frequently shares his views in conferences. He currently works for DBS Singapore, and is also an adjunct lecturer in the National University of Singapore.

Paul E. McMahon



Paul E. McMahon has been active in the software engineering field since 1973 after receiving his master's degree in mathematics from the State University of New York at Binghamton (now Binghamton University). Paul began his career as a software developer, spending the first twenty-five years working in the US Department of Defense modeling and simulation domain. Since 1997, as an independent consultant/coach (<http://pemsystems.com>), Paul helps organiza-

tions and teams using a hands-on practical approach focusing on agility and performance.

Paul has taught software engineering at Binghamton University, conducted workshops on software engineering and management, and has published more than 50 articles and 5 books. Paul is a frequent speaker at industry conferences. He is also a Senior Consulting Partner at Software Quality Center. Paul has been a leader in the SEMAT initiative since its initial meeting in Zurich.

Michael Goedicke



Prof. Dr. Michael Goedicke is head of the working group Specification of Software Systems at the University of Duisburg-Essen. He is vice president of the GI (German National Association for Computer Science), chair of the Technical Assembly of the IFIP (International Federation for Information Processing), and longtime member and steering committee chair of the IEEE/ACM conference series Automated Software Engineering. His research interests include, among others, software engineering methods, technical specification and realization of software systems, and software architecture and modeling.

He is also known for his work in views and viewpoints in software engineering and has quite a track record in software architecture. He has been involved in SEMAT activities nearly from the start, and assisted in the standardization process of Essence—especially the language track.