Brief Contents

Detailed Contents

Part III Making Software

Part IV Testing, Maintaining, and Modifying Software Systems

Part V Latest Trends of Software Development

CHAPTER

1

What is Software Engineering?

Learning Objectives

In this chapter, we begin by exploring some of the foundations of software engineering. Specifically, we consider:

- Various definitions of software engineering
- Characteristics of software
- Problems confronting software engineering
- Its response to the problems
- Challenges with the response and the grand challenge
- What it is like to be a software engineer

1.1 MOTIVATION

Most textbooks on software engineering start with a picture of gloom. Copious references are made to the 'software crisis' (see Chapter 2), with indications that the crisis has not ended yet, and insinuations that it may never end. The monumental cost of software failure is highlighted with facts and figures. Perhaps all of this is meant to emphasize the difficulties of software engineering and the onus on an aspiring software engineer. When I read such books as a student (or at least *started* reading), the first few pages of the first chapter had a rather depressing effect. Given the gory details of the crisis software engineering seemed to be perpetually in, I was not sure I wanted to risk my happiness getting sucked into that vortex of missed deadlines, unhappy customers, and other vicissitudes. But in spite of those ominous openings, fortunately, I ended up being a software engineer. After more than a decade of studying, researching, and practicing software engineering, when I come across similar books now, I find their overtures both odious and misplaced.

Odious, since it is both in bad taste and pedagogically sterile to introduce a student to a discipline by reciting all the privations of the past. It is very important

to challenge the student, but it does no good to present the discipline in a foreboding light. While recognizing that many of the facts reflecting on the difficulties of software engineering are true, their introduction in the first few pages is still misplaced. Software engineering, as we shall see, is a very young discipline. Many of its monumental failures are very much in recent memory. On the other hand, no one is old enough to remember exactly how many bridges fell (at least the falling of the London bridge is canonized in nursery rhyme!) or how many trains tumbled (well, they still do), before the engineering behind these artefacts stabilized. Every engineering work is a trial-and-error game, as so brilliantly argued in books like [Petroski 1992], and software is no exception. To the prepared mind, failures are great learning aids, but to beginners, they are hardly inspiring.

Like any other human endeavour that is *alive*, software engineering is a work in progress. If I have the privilege of ushering bright, young minds (I am talking about you, the reader) into the field, I prefer to do so by outlining the challenges we face in building beautiful, flexible, and resilient software. Yet, at the same time underlining that you will be equal to those challenges in your lifetime with software engineering. This book is a journey to get you started with the best equipment we have now, so that you can fully utilize better equipment that comes to you in future. This chapter begins our journey.

1.2 DEFINITION OF SOFTWARE ENGINEERING

When asked to define his subject, one mathematician reportedly said, mathematics is what is done by mathematicians; and mathematicians are those who do mathematics [Hamming 1997]. This is surely a joke, and the humour perhaps lies in trying to define something in terms of itself. But this anecdote also highlights how difficult it is to define anything, even as established and important as mathematics. Defining software engineering poses more problems, at least quantitatively. First of all, when compared to ‘mathematics’, ‘software engineering’ is two words versus one. Moreover, both ‘software’ and ‘engineering’ are so-called *operative* words. There is no consensus on what ‘engineering’ means, and even less unanimity on what we mean by ‘software’. Thus trying to make sense of software engineering by tying the definition of ‘software’ with that of ‘engineering’ is likely to create even more rancour. Instead of taking on such a task ourselves, let us see how others have tried to define software engineering.

- According to Boehm, software engineering involves the application of science and mathematics through which the facilities of computer equipment are made useful to human beings via computer programs, procedures, and associated documentation.

- Pfleeger identifies software engineering with the utilization of tools, techniques, procedures, and paradigms toward quality improvement of the software product.
- Naur and Randall see software engineering in terms of establishing and using sound engineering principles to obtain economically effective and reliable software that can work efficiently on real machines.
- According to Freeman and Von Staa, software engineering involves the organized application of methods, tools, and knowledge towards fulfilling stated technical, economic, and human goals for a software-intensive system. Interestingly, in recent literature [Booch 2006], 'software-intensive systems' is being increasingly used to denote what we customarily call 'software systems'. The new nomenclature highlights that to be successful, software has to successfully integrate within a larger framework of technological, commercial, and human concerns.
- Kacmar says simply applying engineering principles to designing and constructing computer software can be termed software engineering.
- To Schach, software engineering is the discipline that aims at producing fault-free software, to be delivered on time and within budget, which satisfies the user's needs.
- Whitmire describes software engineering as a 'slippery' term, and says for some it is something that can only be applied to a large project, while to others it is just a 'figment of collective imaginations'. He gives a *working definition* as, 'Software engineering is the science and art of designing and building, with economy and elegance, software systems and applications so they can fill the uses to which they may be subjected' [Whitmire 1997].

Now, what is the essence that ties these definitions together?

1.3 CHARACTERISTICS OF SOFTWARE

All the definitions in the previous section together make up our *current* understanding of software engineering, which may not necessarily be complete. Software engineering is very much a work in progress, due to its relative youth, as well as the very nature of software. We will consider these topics in more depth in Chapters 2 and 3. However, it is appropriate now to mention the set of software characteristics Brooks identified decades ago [Brooks 1995]; and whose depth and relevance we are still discovering.

- Software is inherently complex.
- Software must be made to conform to existing interfaces.
- Software is constantly subject to change.
- Software is invisible and unvisualizable.

Exhibit 1.1 What's in a Name?

Shakespeare, in the romantic classic *Romeo and Juliet*, has the hero say, 'What's in a name? That which we call a rose by any other name would smell as sweet'. This oft-quoted phrase is taken to mean that names do not matter, substance does. But for software engineering, in the beginning at least, names did matter.

The phrase 'software engineering' was first used in a public discourse at a NATO Science Committee sponsored conference, held at Garmisch, Germany, from 7th to 11th October, 1968 [Bauer et al. 1968]. The conference was a visionary exercise, seeking as it did to bring together experts from the industry, academia, and user communities to chart out the course of software development for the future. Discussions were organized in the areas of Design, production, and service of software. The conference proceedings, now publicly available [Bauer et al. 1968] illuminate how much has changed with software engineering till date, with newer tools and technologies; as well as how little has changed, in terms of basic concerns and expectations. While deliberating on the 'nature' of software engineering, the importance of feedback was highlighted many times during the conference [Bauer et al. 1968]. This feedback aspect assumes much importance in the light of what we discuss later in this chapter.

There have been many conclaves on software engineering ever since, but the 1968 NATO conference gave software engineering a name, in the most literal sense.

We will not get into the detailed discussion of each of the above characteristics at this time; let us wait till they unravel themselves as we get deeper into the book. We will remark in passing, however, that although the above may not capture *all* that is important about software, it certainly touches upon the essence of software as a unique artefact of human ingenuity and utility. The problems that software engineering addresses draw largely from these characteristics of software.

1.4 PROBLEMS CONFRONTED BY SOFTWARE ENGINEERING

Every engineering task starts off in response to some pressing problem. Civil or structural engineering served the need to have shelter; mechanical engineering addressed the need for locomotion; electrical engineering catered to growing energy demands; and chemical engineering unlocked the hidden potential of matter. What is, if any, the corresponding 'mission' for software engineering?

Software engineering confronts the problems of *change* and *complexity*.

1.4.1 Problem of Change

When a bridge, a house, or a car is built and given to us, we try to use it, love it, or hate it, continue using it, or move on to a new bridge, house, or car. When a

software system is given to us, we try to use it, love it, or hate it, and want the same system to work the way we want it to. Typically, we do not know the way we want it to work, before we start using it.

The very nature of software—its *plasticity*—makes it amenable to a continuous cycle of change. It seems rather easy to accomplish. After all, tweaking one statement in a software program can radically alter the program's behaviour. But such tweaking—little by itself, but considerable in conjunction—can end up changing the intent of the program's Design in fundamental ways. It is absurd to expect a car to fly or float. But very often a software system built for one context is expected to function in drastically different contexts, with the same grace and efficiency. These expectations can be traced to our wide *cognitive gap* [Datta 2007] with the use of software. Decades and centuries of using cars and bridges respectively, and millennia of using houses has ingrained in our minds what cars, houses, and bridges can and cannot do. Accordingly, we tune our expectations as well as environmental factors to set the context for these systems to function. In comparison, the use of software amongst a large community of lay users has just begun. Our understanding of how and to what extent software can serve our needs is yet not complete. As a result, the problem of change for software comes primarily from changing user expectations, and also from changes in the environment—technological and social.

1.4.2 Problem of Complexity

Complexity is a complex word and there is no one definition to cover its ken; even reaching a definition is an onerous task [Nicolis and Prigogine 1989], [Waldrop 1992]. But we need to care about it in life as well as in software engineering as complexity arises out of simplicity, at times suddenly and surreptitiously. Think of a simple computer program of five lines of code. It is straightforward; by carefully reviewing each line, we can hope to have complete knowledge of the program's structure and behaviour. Now what if, a *loop* is introduced in the program—a simple construct that executes a set of statements repetitively, until a condition holds. The number of execution paths through the program has significantly increased now, and it has become far more difficult to know for sure what happens in each step when the program runs. (As we have illustrated in Chapter 17, for any non-trivial software system, an impractical amount of time and effort is needed to test each and every path of the program's execution.) This example is just a watered down instance of the *combinatorial* complexity software systems customarily face.

Then there are even more involved issues such as complexity of the problem domain, complexity in the interaction of the various forces—technological, commercial, political—that a software system has to balance to be successful.

We have made a case for software's complexity in Chapter 12 and will not go into the details here. However, one must note that a common feature of complex systems is that they are greater than the sum of their parts. Anyone who has done a class project to build a piece of software stretching across several files can appreciate the sense of this statement: A piece of software is made of individual files, but it delivers something that merely bunching the files together will not achieve. Now scale-up to a real world system—with hundreds, if not thousands of files; and thousands, if not millions of interfaces between them; perhaps simple by themselves, but certainly complex when functioning together. And this is just one, relatively less significant, facet of software complexity. Given that change and complexity are facts of life, what does software engineering do about them?

1.5 THE SOFTWARE ENGINEERING RESPONSE

The software engineering response to complexity and change comes in two parts: breaking down the problem into smaller, more manageable 'chunks' to confront complexity, and setting regular checkpoints during the process of building a software system to address the effects of change. The breaking down results in something we will call *workflows* and the checkpointing leads to *phases*; together they constitute the *software development life cycle* or the SDLC. The SDLC lies at the heart of software engineering and we take it up in right earnest later in the book (Chapters 4 and 14). We will now briefly discuss how the problems of complexity and change are addressed.

Workflows represent sets of activities starting from understanding what users want from a software system (Requirements), to translating the language of the problem into the language of the solution (Analysis), to expressing the solution constructs in the language of development (Design), to building the system using programming resources (Implementation), and finally, verifying whether the system matches the stated Requirements (Testing). Phases, on the other hand, are focused towards monitoring and managing change. During *Inception* we ask, what do the users want from the system? During *Elaboration*, we are interested in knowing if the system is feasible. Next comes the question: How do we build the system? This is the concern of *Construction*. Finally, during *Transition*, we enquire, how do we transfer the system from the developer domain to the user domain? In a particular development life cycle, we may not know the answers to these questions when we ask them. But based on our experience and understanding, we have an expectation of what the answers are likely to be. When expectations are not met, it serves as a reality check: A change, not budgeted for, must have occurred. This makes us aware of the need to find out what changed and what that change might affect.

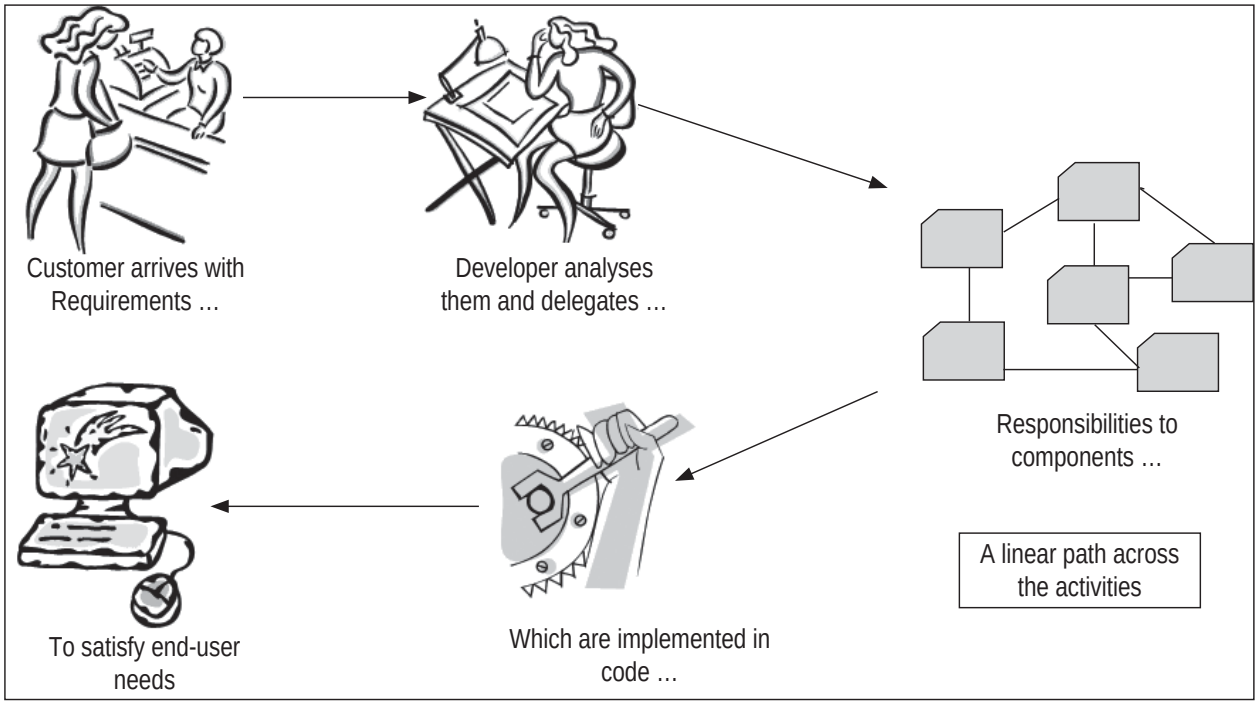

Figure 1.1 Software development, ideally

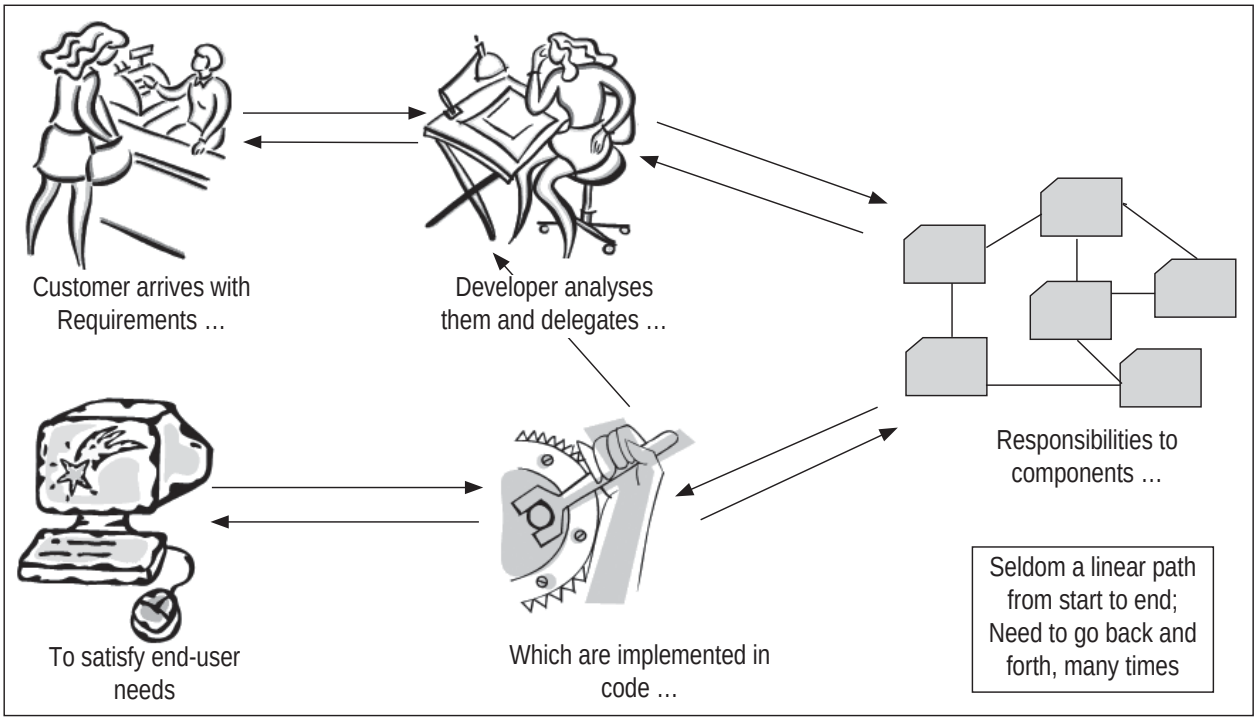

Figure 1.2 Software development in reality

On the face of it, software engineering's response to the problems of change and complexity—seems cogent. But certain challenges come with it.

1.6 CHALLENGES WITH THE RESPONSE

As outlined above, an element of *linearity* is implicit in software engineering's response to the problems of change and complexity. Customers come with Requirements, which are Analysed, followed by the Design of the system, its Implementation and Testing. The right questions are asked at each point; and Inception, Elaboration, Construction, and Transition seem to follow one another in harmony.

But reality is much messier. Answers are seldom ready when questions are asked; at the very least, customers and users change their minds all the time, and technology and business environments change. Thus, in the real-world of software development, it becomes imperative to go back and forth across workflows and phases several times, driven by a variety of reasons. Figures 1.1 and 1.2 highlight the differences between the ideal and real world of software engineering. Life is inherently non-linear, and software engineering is no exception. But just as in life, in software engineering too, we build our case on assumptions of linearity. And then hope to tackle non-linearity, on a case-to-case basis.

So the key challenge with software engineering's response boils down to being able to monitor, control, and utilize the many *feedback* paths that exist in the real-world software development life cycle (SDLC). 'Feedback is one of the most fundamental techniques of engineering. In the simplest of terms, feedback is a mechanism for controlling an activity by regulating the input based on the output' [Datta 2007]. Figure 1.3 illustrates a simple feedback mechanism. *Processor 1* is the primary processor of information; the *Comparator* compares the actual output from *Processor 1* with the expected output, and depending on the results, feeds back information to the optional *Processor 2*, whose output is added to the initial input by the *Adder* and fed into *Processor 1*. A system without a feedback does not have its input conditioned by the output. The simple act of closing the loop (taking the output of the comparator and adding it to the input, via the optional *Processor 2*) can have a profound effect on system behaviour.

In our discussions throughout this book, we shall see how important a role feedback plays in software engineering. Feedback exists at many levels, practical as well as perceptual. An exception handler is a simple example of a feedback loop. It monitors the execution of a piece of code and takes appropriate action if the outcome is not as expected. On the other hand, modifying a system based on user response is also an example of feedback. In software engineering, often the difficulty lies in integrating the various forms and levels of feedback into a consistent and repeatable development model. This is the central challenge with the software engineering's responses to the problems of change and complexity.

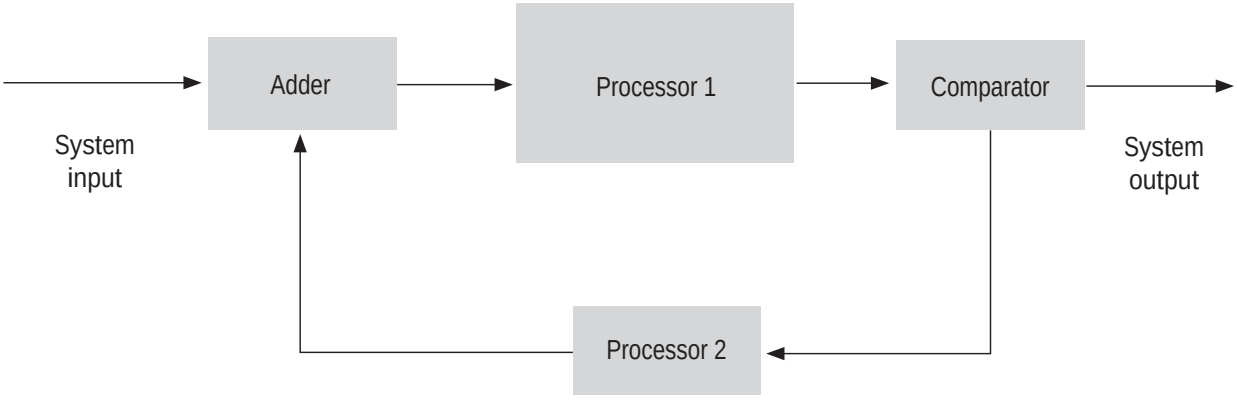

Figure 1.3 A simple feedback mechanism

1.7 GRAND CHALLENGE

A few years back, pioneering computer scientist, inventor of the *quicksort* sorting algorithm, and Turing award winner, C.A.R. 'Tony' Hoare, outlined a set of *grand challenges* for computing research [Hoare and Milner 2005]. According to Hoare, a typical grand challenge is like proving Fermat's last theorem (already accomplished), putting a man on moon (already accomplished), finding a cure for cancer in 10 years (not yet accomplished), and mapping the human genome (already accomplished). A grand challenge project typically lasts around 15 years, has world-wide participation, presents clear criteria for evaluating success, and offers a path-breaking advance in basic science and engineering.

One of the grand challenges Hoare identified for the next 15–20 years is 'Dependable Systems Evolution'. This aims to address the dependability of programs running in homes, offices, cars, planes, and rockets by developing tools and technologies to have the computer guarantee the integrity, safety, and correctness of its own programs. This guarantee should remain in place even as the programs evolve to deliver better service or meet new needs. The project is expected to illuminate the 'logical foundations of computer science and its application to software engineering' [Hoare and Milner 2005]. Note how the challenge to build dependable software systems involves *change* (better service, new needs) and *complexity* (multiple domains of operations, from homes to rockets): These are the same problems for software engineering we have discussed earlier in the chapter. The fact that someone of Hoare's erudition and experience identifies the problem of dependable software evolution as a 'grand challenge', points to how important it is for the world, and how difficult it is to solve.

Who will meet the grand challenge of Dependable Systems Evolution?

I am sure it will be you; the bright, young minds who study software engineering today and will research and practice it tomorrow.

1.8 WHAT IT IS LIKE TO BE A SOFTWARE ENGINEER?

From what we have discussed so far, software engineering seems exciting, but serious business, with its problems of change and complexity, its response, and the challenges with the response. Indeed, there is software engineering in nearly every aspect of our lives today. And thus it is no surprise, we as software engineers encounter the most intriguing and important problems. This book takes you on a journey of discovery; how software engineering is doing today, as well as how you will do it even better, tomorrow. It is an exciting journey. For you to embark on a spirit of fun and adventure, let us see what it is like to be a software engineer.

1.8.1 Knowing across Domains

As we discussed briefly earlier, and will discuss in detail in Chapter 3, software is unique amongst engineering artefacts in a number of ways. A bridge, a building, or a bicycle has some value per se. They help us cross a chasm, give shelter from the elements, or let us go from one place to another. On the other hand, a piece of software usually provides a *service* to an existing enterprise, helping get it done faster, better, and cheaper. Of course, additionally there are also truly game-changing uses of software. They let us do something that was never imagined before, like deciphering the human genome. But these lie in the realm of research for quite some time before getting integrated into the so-called mainstream of software engineering.

Over the course of a typical software engineering career, even when working on the usual customer-oriented projects, there is much scope for acquiring knowledge across a wide spectrum of application domains and industries. The domains of finance, travel, health care, entertainment, etc., are each different, with its own vocabulary, quirks, and challenges, as well as rewards. Some solutions work well for a particular domain, while wholly new ones have to be devised for others. Software engineers thus need to stay updated with working knowledge across various domains, in addition to refining their core skill of making software. This is not easy, but it provides for intellectual stimulation and variety not readily available in many other professions. It is enlightening to know of different industries, and their unique user expectations and functioning. Successful software engineers use this as an opportunity for hastening professional maturity.

1.8.2 Teaming across Cultures

Software engineering is very much a global enterprise now, and this trend will only grow in the future (see Chapter 21). This offers unique opportunities for interactions across cultures, even for those at the entry level. Indeed, to become a successful software engineer, it is becoming essential to acclimatize oneself quickly

and easily to a variety of work environments. This includes an understanding of social, political, regional, and cultural sensitivities, in addition to technical and communication skills.

Being a team player is an important criterion for success in the engineering profession—after all, no serious engineering product comes out of the head or hand of a single individual. For software engineers, merely being a team player is not sufficient. The most successful software teams, the truly 'jelled' ones [Humphrey 1999] have the key ingredient of diversity. Most often, we need to work closely with people who are very different—in language, culture, background, and skill—yet united in a common professional purpose. Learning how to embrace and thrive in such environments is hardly something textbooks or class lectures can teach you. Your own attitude and temperament are your best—and often only—teachers.

1.8.3 Innovating across Technologies

As we mentioned before, software engineering is a young discipline; almost infant when compared to some of the conventional engineering disciplines. While this has its disadvantages—we are still groping for laws and first principles—it also makes our field fertile for innovation. With little discipline, focus, and imagination, every software engineer can innovate.

The burgeoning open source paradigm (see Chapter 22), has made it easy to interact with the software community at large. And this interaction fuels innovation. Ingredients for innovation are now available to every practicing software engineer as basic professional tools. Web access and efficient computing facilities, are necessary for software engineering innovation; but they are not sufficient. What remains is a key element: the willingness to think outside the box. Even amidst the grind of day-to-day work, with looming deadlines and delivery pressures, it is not impossible to think a little deeply on the most pressing issues at hand; why is a particular task taking so much time, how can performance of a component be improved, is there a general solution to a particular problem? Such 'lateral' thinking will alleviate the tension and ennui of everyday work. Also, sooner or later, it will lead to some innovation not only satisfying by itself but also offering valuable career boost. Hamming's description of how he developed his pioneering work on error-correcting codes, while doing his routine work is very inspiring [Hamming 1997] for today's software engineers looking to innovate. The facilities a software engineer has today even for routine work were unthinkable a few years ago. It is our onus—to society, to our profession, and most importantly, to ourselves—to utilize these facilities to their fullest.

Summary and Take-Aways

This chapter begins the book's journey of discovering what software engineering is, what its major challenges are, and how tomorrow's software engineers—the readers of this book—can stand up to those challenges and get way beyond them. Our discussion can be summarized as follows:

- There is no single universally accepted definition of software engineering; the essence lies in synthesizing the various definitions.
- Change and complexity are the two major problems confronting software engineering.
- To address the problem of complexity, software development is broken down into workflows, each of which addresses a specific concern in the development process.
- To address the problem of change, phases of software development monitors changes and their effects during the development process.
- Workflows and phases together constitute the software development life cycle; which lies at the heart of software engineering.
- The key challenge of software engineering is to be able to monitor, control, and utilize the many feedback paths that exist in real-world software development.
- Hoare has identified the evolution of dependable software systems, carrying with it the guarantee of acceptable behaviour across a wide variety of operating conditions, as one of the grand challenges of computing in the next 15–20 years.
- Knowing across domains, teaming across cultures, and innovating across technologies are the key elements of a software engineer's experience.

Where to Look for More

Although software engineering is a young discipline, a body of informative and insightful writing has already been accumulated. The website http://tinyurl.com/100sebooks lists the so-called 'Top 100 Best Software Engineering Books, Ever'. While this may not be the definitive list—few of my own favourites are not featured—it does identify some very good books. Additionally, the author discusses the metrics he used in ranking the books, which may be generally helpful in choosing a good book.

EXERCISES

Review Questions

Review Questions test your understanding of the key concepts presented in this chapter.

1. Which of the following is not included in Whitmire's working definition of software engineering?
   (a) Economy
   (b) Use of a software system
   (c) Art
   (d) Performance

2. According to Brooks, which of the following is a characteristic of software?
   (a) Complexity
   (b) Changeability
   (c) Invisibility
   (d) All of the above

3. Software systems need to encounter the problem of change primarily because
   (a) users do not initially know what they want from software
   (b) user needs are complex
   (c) there is combinatorial complexity in software
   (d) of all of the above

4. Which of the following is not a concern associated with a workflow?
   (a) Testing
   (b) Feasibility study
   (c) Analysis
   (d) Implementation

5. Which of the following is a concern associated with a phase?
   (a) Testing
   (b) Feasibility study
   (c) Analysis
   (d) Implementation

Reflective Questions

Reflective Questions require you to think deeply about some of the ideas and come up with your own interpretations and answers.

1. Comment on the following statement in the light of the various definitions of software engineering: 'No matter how we define it, the most important component of software engineering is computer programming.'

2. Among the various definitions of software engineering given in this chapter, which one do you think comes closest to software engineering as you see it? Justify your answer.

3. Out of the four characteristics of software mentioned by Brooks (few decades ago), which one do you think is most relevant to software as it is perceived and used today, and which one the least? Support your choices with reasons.

4. Are invisibility and unvisualizability the same characteristic of software? If not, why? Explain with examples.

5. In this chapter, we have identified two major problems that software engineering needs to address. Can you correlate them with the characteristics of software Brooks identified?

6. Do you think the so-called cognitive gap mentioned in the context of software vis-à-vis other engineering disciplines is valid? Give reasons for your answer.

7. Change and complexity are the two major problems confronting software. Are these two related? Can the response to one serve the other?

8. How do you think workflows and phases are related? Are they aligned or orthogonal to one another?

9. Feedback is everywhere. Give an example of feedback and discuss the benefits it offers in that particular case and how the situation would have been without feedback.
10. How do you think we should approach the grand challenge of Dependable Systems Evolution? Is there a particular way that Hoare suggests? What do you think of his suggested path?
11. In this chapter, we have outlined some aspects of what it is like to be a software engineer. Which aspect attracts you most? Which one do you think is most boring?

REFERENCES

Bauer, F.L., Bolliet, L., and Helms, H.J. (1968), NATO Software Engineering Conference 1968, http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF, last accessed on Nov 8, 2009.

Booch, G. (2006), 'The Accidental Architecture', *IEEE Softw.*, 23(3): 9–11.

Brooks, F.P. (1995), *The Mythical Man-Month: Essays on Software Engineering*, 20th Anniversary Edition, Addison-Wesley.

Datta, S. (2007), *Metrics-Driven Enterprise Software Development: Effectively Meeting Evolving Business Needs*, J. Ross Publishing.

Hamming, R.R. (1997), *Art of Doing Science and Engineering: Learning to Learn*, CRC.

Hoare, T. and Milner, R. (2005), Grand challenges for computing research, *Comput. J.*, 48(1): 49–52.

Humphrey, W.S. (1999), *Introduction to the Team Software Process*, SEI Series in Software Engineering.

Nicolis, G. and Prigogine, I. (1989), *Exploring Complexity: An Introduction*, W.H. Freeman and Company.

Petroski, H. (1992), *To Engineer Is Human: The Role of Failure in Successful Design*, Vintage.

Saxe, J.G. (1850), 'The Blind Men and the Elephant', http://bygosh.com/Features/092001/blindmen.htm, last accessed on Nov 8, 2009.

Waldrop, M.M. (1992), *Complexity: The Emerging Science at the Edge of Order and Chaos*, Simon and Schuster.

Whitmire, S.A. (1997), *Object-Oriented Design Measurement*, Wiley Computer Pub.