An Introduction to Software Engineering

Software engineering is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use. In this chapter, we will explain the following:

- the definition of computer science and software engineering and how the two are different
- software engineering is similar to other engineering disciplines and what that means for software engineers
- the unique challenges of software engineering
- software development models and processes and their component parts, software development practices

> *Software systems are perhaps the most intricate and complex . . . of the things humanity makes.* *– Fred Brooks (Brooks, 1995)*

As a discipline, software engineering has progressed very far in a very short period of time, particularly when compared to classical engineering field (like civil or electrical engineering). In the early days of computing, not much more than 50 years ago, computerized systems were quite small. Most of the programming was done by scientists trying to solve specific, relatively small mathematical problems. Errors in those systems generally had only "annoying" consequences to the mathematician who was trying to find "the answer." Today we often build monstrous systems, in terms of size and complexity. What is also notable is the progression in the past 50 years of the visibility of the software from mainly scientists and software developers to the general public of all ages. "Today, software is working both explicitly and behind the scenes in virtually all aspects of our lives, including the critical systems that affect our health and well-being." (Pfleeger, 1998)

Despite our rapid progress, the software industry is considered by many to be in a crisis. Some 40 years ago, the term "Software Crisis" emerged to describe the software industry's inability to provide customers with high quality products on schedule. "The average software development project overshoots its schedule by half; larger projects generally do worse. And, some three quarters of all large systems are "operating failures" that either do not function as intended or are not used at all." (Gibbs, 1994) While the industry can celebrate that software touches nearly all aspects of our daily lives, we can all relate to software availability dates (such as computer games) as moving targets and to computers crashing or locking up. We have many challenges we need to deal with as we continue to progress into a more mature engineering field, one that predictably produces high-quality products.

1 The Engineering of Software

Up until this point in your academic career, you have likely focused on being a computer scientist. Consider the definition of computer science offered by CSAB,

the organization that accredits Computer Science programs in the United States (CSAB, 1997):

> *Computer science* is a *discipline that involves the understanding and design of computers and computational processes. In its most general form it is concerned with the understanding of information transfer and transformation. Particular interest is placed on making processes efficient and endowing them with some form of intelligence.*

It is likely that your main focus thus far has been to get the computer to do what you want it to do, as efficiently as possible. There are definitely other issues to consider. There are many, many definitions of the term engineering. One that we feel captures the essence has been proposed by Robert Baber (emphasis added) (Baber, 1997):

> *. . . the systematic and regular application of scientific and mathematical knowledge to the design, construction, and operation of machines, systems, and so on of practical use and, hence, of economic value. Particular characteristic of engineers is that they take seriously their responsibility for correctness, suitability, and safety of the results of their efforts. In this regard they consider themselves to be responsible to their customer (including their employers where relevant), to the users of their machines and systems, and to the public at large.*

Computer science is one of the disciplines that provide a theory basis for the profession of software engineering. (Some others are psychology, economics, and management.) There are two important issues beyond "getting the computer to do what you want, as efficiently as possible" when transitioning to software engineering. The issues underlined in the above definition of engineering are further discussed below:

- *Practical use, economic value.* Engineers need to produce products that customers actually want and are willing to pay real money for. These products need to help people do the things they need to do. Listening to the customer is of prime importance. Engineers also need to produce these products the customer wants as economically as possible. The best product in the world won't sell if it's too expensive. And, if we develop products using inappropriate practices and processes, our products will be too expensive. As engineers, we need to determine the content and build the best product value to our customers.

- *Responsibility for correctness, suitability, and safety.* Engineers are ethically obligated to ensure their programs are correct and suitable for their customers. In fact, there is a software engineering code of ethics (ACM/IEEE-CS Joint Task Force on Software Engineering, 1999) that we are responsible for adhering to. In some instances, our programs have safety critical implications, where people might die if a program has errors. In other cases, whole businesses could be at risk if a program is not correct. We are sure that you

have always tried to get your programs to be correct and suitable in the past. The new dimension now is that you must always consider your responsibility and obligation to your customer. The work you do could impact their safety, their business . . . and their well being!

- *Regular application of scientific and mathematical knowledge*. As was said, in our field we are just beginning to build such knowledge that is common in other engineering fields.

2 Software Development

Software engineering is *the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software* (IEEE, 1990). The "systematic, disciplined, quantifiable approach" is often termed a software process model (in the general sense) or a software development process (in the specific sense). Specific software development processes consist of a particular set of software development practices which are often performed by the software engineer in a predetermined order. Software development practices, models, and methodologies will be introduced in the next two subsections.

2.1 Software Development Practices

Engineers adopt a systematic and organized approach to their work. As you learn software engineering, you should be exposed to many specific practices (or techniques) for developing software. By *software development practice* we refer to *a requirement employed to prescribe a disciplined, uniform approach to the software development process* (IEEE, 1990), in other words, a well-defined activity that contributes toward the satisfaction of the project goals; generally the output of one practice becomes the input of another practice. First, we provide one list of software development practices (but this list may vary depending upon the process and its associated terminology):

- Requirements engineering
- System analysis
- High-level design/architecture
- Low-level design
- Coding
- Integration
- Design and code reviews
- Testing
- Maintenance
- Project management
- Configuration management

Most disciplines come to recognize some practices as *best practices*. A *best practice* is a *practice that, through experience and research, has proven to reliably lead to a desired result and is considered to be prudent and advisable to do in a variety of contexts*. Over time, we accumulate information on whether new practices are good or not. This information might be just stories of people succeeding with the practice, generally called

anecdotal or qualitative evidence. Ideally, someone has done a controlled experiment that shows that a new practice is better than some other practice. This is called empirical or quantitative evidence. For example, before the damages of smoking were quantitatively assessed, there were physicians who recommended their patients not to smoke because there was some sort of evidence that the smoke was bad. Ultimately, structured empirical analysis backed up these physicians advice.

Those of you familiar with music will understand the concept of an etude. An etude is a musical composition written solely to improve technique. At the XP Universe conference in 2001, Kent Beck, the originator of Extreme Programming (XP) (Beck, 2000) likened learning software best practices to etudes in music. When he was learning to play a musical instrument, he was given etudes – short scores of music – to play over and over and over again. He said these short scores were not pleasing to the ear. The purpose of learning to play them was to really engrain in him how to play that kind of a combination of notes. Then later, when that sort of combination of notes appears in the midst of a larger beautiful composition, the notes will just flow off his fingers. Learning each etude was fairly painful, but the practice led to beautiful music.

So, how does this relate to software development? As you study software engineering, you will learn about many software development practices. You'll learn each individually, and (hopefully) you will "play" them over and over again. You will come to understand and appreciate when a certain practice is very rigorous and probably good for safety-critical software while a similar practice is not so painstaking and perhaps better for small projects. Engineering is all about selecting the most appropriate method for a set of circumstances – the right tool for the job. The goal is that when you are faced with a project, you will understand what types of practices are appropriate for that kind of project. You will then be able to include these practices into a suitable process just as an etude in incorporated into a classical score. Then, you will be making beautiful software!

2.2 Software Process Models and Methodologies

In simplistic terms, if you string an appropriate set of specific software practices together and this set accomplishes all the fundamental activities listed in 3.1, you create a software development process. A *software development process* is t*he process by which user needs are translated into a software product. The process involves translating user needs into software requirements, transforming the software requirements into design, implementing the design in code, testing the code, and sometimes installing and checking out the software for operational use. Note: these activities might overlap or be performed iteratively.*

A software process model is a *simplified, abstracted description of a software development process*. The primary purpose of a software process model is to determine the order of stages involved in software development and to establish the transition criteria for progressing from one stage to the next (Boehm, May 1988). Because of the simplification, several software development methodologies may share one process model – the differentiation is in the details of the process itself. Software methodologists incorporate the general characteristics of software development **models** into specific

software development **processes** that adhere to the spirit of these models. While software development models have general characteristics, such as "having strong documentation and traceability mandates across requirements, design and code" (Boehm and Turner, 2003), software development methodologies have *specific practices* that need to be followed, such as code inspection.

Of recent, process models have begun to be characterized as plan-driven or agile (Boehm, 2002). The *plan-driven models* have an implicit assumption that a good deal of information about requirements can be obtained up front and that information is fairly stable. As a result, creating a plan for the project to follow is advisable. A long-standing tenet of software engineering is that the longer a defect remains in a product, the more expensive it is to remove it. (Boehm, 1981; Humphrey, 1995) An overriding philosophy of plan-driven software models is that the cost of product development can be minimized by creating detailed plans and by constructing and inspecting architecture and design documents. As a result of these activities, there will be significant cost savings because defects will be removed or prevented. Plan-driven models can be summarized as "Do it right the first time." These models are very appropriate for projects in which there is not a great deal of requirements and/or technology changes anticipated throughout the development cycle. Plan-driven models are also considered more suitable for safety- and mission-critical systems because of their emphasis on defect prevention and elimination. (Boehm, 2002) Some examples of plan-driven methodologies are the Personal Software Process (Humphrey, 1995), the Rational Unified Process (Jacobson, Booch et al., 1999), and Cleanroom Software Engineering (Mills, Linger et al., 1987).

Alternately, *agile models* are considered to be better suited for projects in which a great deal of change is anticipated. (Boehm, 2002) Because of the inevitable change, creating a detailed plan would not be worthwhile because it will only change. Spending significant amounts of time creating and inspecting an architecture and detailed design for the whole project is similarly not advisable; it will only change as well. The methodologies of the agile model focus on spending a limited amount of time on planning and requirements gathering early in the process and much more time planning and gathering requirements for small iterations throughout the entire lifecycle of the project. Some examples of agile methodologies are the Extreme Programming (XP) (Beck, 2000), Scrum (Schwaber and Beedle, 2002), Crystal (Cockburn, 2001), FDD (Coad, LeFebvre et al., 1999), and DSDM (Stapleton, 1997).

However, there need not be a dichotomy between the two models; hybrid models that have both agile and plan-driven characteristics have been used successfully in many projects.

3 Software Engineering Challenges

There are some unique and pressing issues to deal with in the software industry. Several of these are now discussed:

- *Tractable Medium.* We are engineers, yet what we engineer is a logical and tractable, not physical medium. The constraints of physical medium can serve to

simplify alternatives. For example, in a house design you can't put a kitchen and a bathroom in the same place; batteries have standard voltages. Frederick Brooks, notable software engineer and author of the legendary book *The Mythical Man Month*, expresses an analogy,

> *The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination.* (Brooks, 1995)

This tractability has its own pros and cons. On the positive side, as programmers we have the ultimate creative environment. We can create grandiose programs chock full of beautiful algorithms and impressive user interfaces. And we can completely change this functionality or the look of the interface in mere seconds and have a new creation! Conversely, because we are only dealing with "thought-stuff," our profession has a limited scientific and/or mathematical basis. In other fields, the scientific and mathematic basis of physical, intractable mediums constrain the solution to a problem -- only certain materials can withstand the weight of a car, only certain paints can take the intensity of the UV rays on the top of a mountain, etc. With software, the sky's the limit!

Quite often programmers are also asked to fix hardware product problems because people think that it is cheaper to fix the problems in the (tractable) software than it is to re-design and re-manufacture physical parts. This presents software engineers with the need to design and coding changes, often at the last minute.

The software industry has been trying to formulate a sort of scientific/mathematical basis for itself. Formal notations have been proposed to specify a program; mathematical proofs have been defined using these formal notations. The software community is also establishing analysis and design patterns. (Gamma, 1995; Fowler, 1997) These patterns are general solutions to recurring analysis and design problems; the patterns are proposed, proven and documented by experts in the field. Engineers can become familiar with these general solutions and learn to apply them appropriately in the systems and programs under development.

- *Changing requirements*. Adapting for hardware changes is only one source of requirements churn for software engineers. Unfortunately, requirements changes come from many sources. It is often very hard for customers to express exactly what they want in a product (software is only thought-stuff for them too!). They often don't know what they want until they see *some* of what they've asked for. Requirements analysts may not understand the product domain as completely as they need to early in the product lifecycle. As a result, the analysts might not know the right questions to ask the customer to elicit all their requirements. Lastly, the product domain can be constantly changing during the course of a product development cycle. New technology becomes available. Competitors release new products that have features that weren't thought of. Innovators think of wonderful new ideas that will make the product more competitive.

- *Schedule Optimism*. Software engineers are an optimistic crew. In most organizations, it is the software engineers who estimate how long it will take to develop a product. No matter how many times we've taken longer than we thought in the past, we still believe "Next time, things will go more smoothly. We know so much more now." As a result, we often end up committing to a date we have no business committing to, giving the software industry a "never on time" reputation.

- *Schedule Pressure*. We often make these aggressive commitments because of the intensity of the people asking us for commitment. It seems that every product is late before it's even started, every feature is critical or the business will fold. Products need to be created and updated at a constant, rapid pace lest competitors take over the business.

4. Summary

There are some keys ideas to remember as you begin your study of software engineering. These ideas are summarized in Table 1.

Table 1: Key Ideas for Software Engineering

	Computer science is concerned with getting the computer to do what you want it to do, as efficiently as possible.
	Software engineers use their computer science skills to create products of practical use and economic value. Software engineers are ethically responsible for the correctness, suitability, and safety of their projects. When possible, software engineers apply scientific and mathematical knowledge to their work.
	A software development process is a process by which user needs are translated into a software product. Software development processes are comprised of specific software development practices.
	A software process model is a generalized abstraction of a family of software development processes.
	Plan-driven processes are best for projects with a low degree of change or those with critical safety and security needs.
	Software engineering is especially challenging because software is a tractable medium, requirements often change, and competitive pressures cause schedule pressure.

Glossary of Chapter Terms

Word	Definition	Source
best practice	a software development practice that, through experience and research, has proven to reliably lead to a desired result	

	and is considered to be prudent and advisable to do in a variety of contexts	
computer science	A discipline that involves the understanding and design of computers and computational processes. In its most general form it is concerned with the understanding of information transfer and transformation. Particular interest is placed on making processes efficient and endowing them with some form of intelligence	(CSAB, 1997)
engineering	the systematic and regular application of scientific and mathematical knowledge to the design, construction, and operation of machines, systems, and so on of practical use and, hence, of economic value. Particular characteristic of engineers is that they take seriously their responsibility for correctness, suitability, and safety of the results of their efforts. In this regard they consider themselves to be responsible to their customer (including their employers where relevant), to the users of their machines and systems, and to the public at large.	(Baber, 1997)
Software development practice (or technique)	a disciplined, uniform approach to the software development process	(IEEE, 1990)
Software development process (or methodology)	The process by which user needs are translated into a software product. The process involves translating user needs into software requirements, transforming the software requirements into design, implementing the design in code, testing the code, and sometimes installing and checking out the software for operational use. Note: these activities might overlap or be performed iteratively.	(IEEE, 1990)
Software process model	simplified, abstracted description of a software development process	
software engineering	the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software	(IEEE, 1990)

References

ACM/IEEE-CS Joint Task Force on Software Engineering, "Software Engineering Code of Ethics and Professional Practice."

Baber, R. L. (1997). "Comparison of Electrical "Engineering" of Heaviside's Times and Software "Engineering" of our Times." IEEE Annals of the History of Computing **19**(4): 5-17.

Beck, K. (2000). Extreme Programming Explained: Embrace Change. Reading, Massachusetts, Addison-Wesley.
Boehm, B. (2002). "Get Ready for Agile Methods, with Care." IEEE Computer **35**(1): 64-69.
Boehm, B. (May 1988). "A Spiral Model for Software Development and Enhancement." Computer **21**(5): 61-72.
Boehm, B. and R. Turner (2003). Balancing Agility and Discipline: A Guide for the Perplexed. Boston, MA, Addison Wesley.
Boehm, B. W. (1981). Software Engineering Economics. Englewood Cliffs, NJ, Prentice-Hall, Inc.
Brooks, F. P. (1995). The Mythical Man-Month, Anniversary Edition, Addison-Wesley Publishing Company.
Coad, P., E. LeFebvre, et al. (1999). Java Modeling in Color with UML, Prentice Hall.
Cockburn, A. (2001). Agile Software Development. Reading, Massachusetts, Addison Wesley Longman.
CSAB (1997). "Defining the Computing Sciences Professions." http://www.csab.org/comp_sci_profession.html.
Fowler, M. (1997). Analysis Patterns: Reusable Object Models. Menlo Park, CA, Addison Wesley Longman, Inc.
Gamma, E. H., Richard; Johnson, Ralph; and Vlissides, John (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Massachusetts, Addison-Wesley Publishing Company.
Gibbs, W. W. (1994). Software's Chronic Crisis. Scientific American**:** 86-95.
Humphrey, W. S. (1995). A Discipline for Software Engineering. Reading, MA, Addison Wesley.
IEEE (1990). IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology.
Jacobson, I., G. Booch, et al. (1999). The Unified Software Development Process. Reading, Massachusetts, Addison-Wesley.
Mills, H. D., R. C. Linger, et al. (1987). "Box Structured Information Systems." IBM Systems Journal **26**(4): 395-413.
Pfleeger, S. L. (1998). Software Engineering: Theory and Practice. Upper Saddle River, NJ, Prentice Hall.
Schwaber, K. and M. Beedle (2002). Agile Software Development with SCRUM, Prentice-Hall.
Stapleton, J. (1997). DSDM: The Method in Practice, Addison Wesley Longman.

Chapter Questions

1. Describe the difference between a software process and a software process model.
2. What are the challenges of today's SE? How do software engineers respond to these challenges?
3. Software requirements change is inevitable. However, the requirements of some software are not so volatile. Give three examples of such software. What are the characteristics of this kind of software?

4. For a commercial shrink-wrapped software product, what are the important goals the software developers seek to achieve? List at least 5 items, and rank them in order.
5. Search the web, and find three software process models. Give some description for each model.
6. Based on Baber's definition about engineering and your personal experience, do you think software is engineering? Why? How is software different from other kind of engineering?
7. As a software professional, we must take our ethical responsibility. ACM (Association of Computer Machinery) and IEEE (Institute of Electrical And Electronic Engineers) have produced a code of ethics and professional practice. Find it on the web, and describe in your word what ethical responsibilities we should take.
8. For more than 30 years, software engineers have been thinking how to improve the process of software development. Today, we can find an army of software processes, and new ones are being created. If you were a manager in a software consulting company, would you adapt new software practices? If you would, what would be the motivation? If not, what would be the concerns?
9. Why, in your opinion, are software engineers often over-optimistic?