Introduction to Software Engineering

This is a book about software engineering, so we begin by defining our topic, first by defining "software," and then "engineering." Then we will talk about why software engineering is both hard and important, its status as a discipline and a profession, and how successful software development is after almost fifty years of progress in software engineering. We finish up buy surveying in more detail the concerns that software engineers must address in their work.

What is Software Engineering?

The word "software" is relatively easy to define, but the context matters. To the hardware, software is a sequence of instructions to be executed by a processor. To the developer, software is a collection of human-readable statements (in one or more languages) that can be converted into a sequence of machine-executable instructions. Of course, human-readable statements are converted into instructions executable by a processor, so these two perspectives are strongly related. We will use the term "software" to refer to whichever of these two senses is appropriate in the context, so **software** is a sequence of instructions that can be executed on a computer, or a collection of statements that can be converted into such a sequence.

The word "program" is slightly more difficult to define now than it used to be. In the early days, the word "program" referred both to a file (or deck of cards) containing the human-readable statements necessary to perform one or more specific tasks, and to the file that contained the corresponding sequence of machine-executable instructions. Now, things are much more complicated both because some programs are compiled while others are interpreted, and because both the human-readable and machine-executable instructions are almost always in multiple files. Even worse, mostly for marketing and advertising reasons, a variety of other words have been used over the years like "utility," "tool," "script," and most recently, "app." It is almost impossible to find universally accepted definitions for any of these terms, though most people would agree that they are software and many would agree that they are programs. We will use the word **program** as a generic term for any piece of software that can run on its own.

Besides programs, software can be organized in a variety of ways. One important distinction is between programs and sub-programs and libraries. A sub-program is a collection of instructions or statements that implement an algorithm for accomplishing a specific task or tasks. A **library** is a group of related sub-programs for accomplishing a specific collection of (usually related) tasks. Programs are made up of sub-programs often combined with (parts of) libraries.

People interested in obtaining software typically want more than just programs, sub-programs, or libraries; usually they want everything that they need to solve problems with software. We call such things software products. A **software product** is one or more programs, sub-programs, or libraries, along with data and supporting materials and services, that a client can use to solve problems or achieve goals. For example, MS Word is a software product, as is the software that runs in a car to control its carburetor.

Software products are generally classified as either bespoke or generic. **Bespoke software products** are developed (usually under contract) for a specific customer. For example, a car manufacturer might contract with a software development company for the software that controls the carburetors in its cars. **Generic software products** are developed (usually speculatively) and then sold in a market (either a mass market or a niche market). The consumer products that Microsoft makes and sells are examples of generic software products.

Engineering is the application of (scientific) theories, methods and tools to the specification, design, creation, verification and validation, deployment, operation, and maintenance of products.

For example, civil engineers apply principles of chemistry and physics and methods and tools developed over thousands of years to build and maintain roads, bridges, parks, and other large public structures.

Combining our characterizations of software and engineering, we have that software engineering is the application of engineering discipline to building and software. But customers purchase software products, not just software. Furthermore, software engineering as a discipline tends to emphasize building and maintaining products more than the science behind its techniques. So our final definition is that **software engineering** is the technical and managerial discipline concerned with the systematic and disciplined development, operation, and maintenance of high quality software products delivered on time, at minimum cost. Software engineering applies scientific principles from computer science, management science, psychology, and economics to achieve its goals.

Software engineering has two main areas of concern: managerial concerns and technical concerns.

> *Managerial Concerns*—Managerial concerns are about organizing and controlling the job of building a software product. Such concerns include project cost and time estimation; project scheduling and tracking; team organization and management; risk management; and product quality tracking and improvement.
>
> *Technical Concerns*—Technical concerns are about the tools and techniques we use to determine what product to build, how to build it, and then actually building it. Such concerns include software requirements elicitation and representation techniques and tools; software design techniques and tools; programming language tools and environments; coding standards and practices; defect prevention, detection, and removal tools and techniques; version and configuration management; documentation tools and techniques; and maintenance and debugging tools and techniques.

We will look at these concerns in more detail at the end of this chapter.

Making Software Products

Suppose you get a job working for a small construction company as their first software specialist. You will be expected to build and maintain a web site, and to write programs to help out with various tasks in running the business. Of course, the company already has software for doing inventory, taxes, payroll, and so forth, but occasionally some tasks will arise for which there is no commercial product and you will be asked to develop some software. Because you are the only software developer in the company, you won't just be writing code; you will have to do all the work to develop an entire software product. This means that besides developing the software, you will have to write its documentation, set up databases or configuration files, train the users, and so forth. You will also have to work with management to provide estimates of the time and cost for developing products, report on your progress, explain delays or extra costs, and so forth.

Here are some of the questions that you will have answer to develop software products in your new job.

- What exactly should the new program do? If several people have ideas for the program, you can bet that their ideas will be different. How should these different ideas be reconciled?
- How will the program fit into the company's current ways of operating?
- How exactly should the program interact with its users?
- What parts should the program have and how should they interact? What properties must the program and its parts have? What data structures and algorithms should be used?

- What languages and systems should be used to implement the program?
- What standards and practices should be used in writing the code?
- Does the program do what you expect it to do (that is, does it have bugs)? Does the product satisfy the needs and desires of your coworkers? How will you determine answers to these questions?
- How much effort will be required to build the product? What knowledge and skills are needed? How long will it take you to build it? What other resources will be needed?
- What sorts of documentation or other user support products and services should be part of the product? How will you train your coworkers to use the new program?
- Will the product need to be modified, ported to other platforms, improved, and fixed over time? If so, how will bug reports and change requests be collected? How should the redevelopment effort be different from an initial development effort?
- During development, how will you tell whether you are going to finish on time and within budget? What will you do if things are not going well?

These are questions that must be answered when building small products. Besides these, other issues arise when large products are built. The following list summarizes these issues.

- Specifications for large programs or collections of programs are extensive (thousands or tens of thousands of requirements) and complicated (related to each other in many ways that may be hard to understand).
- Designs for large programs are big (thousands of classes and tens or hundreds of thousands of methods) and complicated (classes related to one another in many ways that may be hard to understand).
- Code for large programs is enormous (millions of lines). For example, the Linux kernel has over 13 million lines of code.
- Testing large programs is extremely difficult because there are so many possible execution paths.
- The larger a programs grows, and the more people are involved in creating it, the harder it is to estimate how much effort will be required to complete it, and to determine whether adequate progress is being made.
- As the number of people involved in a project increases, the communication paths between them increase quadratically, making communication and coordination much more difficult.
- Estimating effort, tracking progress, and communicating and coordinating, all require large efforts in big projects, so extra people (managers and support personnel) must be hired to do all this extra work, which increases the size and cost of the project even more.

Software engineering attempts to deal with the issues involved in building software products of all sizes. As these lists make clear, these issues are challenging, so software engineering is hard. Studying software engineering will make you aware of these issues, and give you the skills and tools you need to resolve them (or at least to have a good chance of resolving them). If building software is important (and the size of the software industry and the pervasiveness of software in out lives suggest that it is), then software engineering is important too.

Software Engineering as a Discipline

Though most people don't think about it much, disciplines have a tendency to proliferate. Most commonly, a discipline will split into new or more disciplines when the specialists in that discipline can no longer agree on the core knowledge and skills in the discipline.

Electronic computers were first invented and came into practical use during the second world war, when they were used to help decipher encoded German military messages. Early computers were hugely expensive and not very capable, and were mostly used to solve scientific and engineering problems. Programming them was relatively easy compared to the challenge of getting the hardware to work, and the people who worked on computers early on concentrated mostly on the hardware. But as computers grew in speed, reliability, and storage capacity through the 1950s, they began to be used for much more demanding applications, and issues about data structures and algorithms began to be very important. The people focused on the hardware began to disagree with the people focused on the software about the core of the discipline, and so the field split into computer engineering, a sub-field of electronics and electrical engineering, and computer science, which is heavily influenced by mathematics and concentrates on discrete mathematics, algorithmic thinking, and numerical methods.

Computers continued to be applied in more and more challenging domains, and by the late 1960s and 1970s practical problems of building massive programs with large teams of developers came to the fore as many software development projects were failing, or finishing late and over budget. Some computer scientists advocated the inclusion of material drawn from project management, psychology, human factors, and other disciplines in computer science, and there was once again disagreement over the core content of the field. This gave rise to the emergence of software engineering as a distinct discipline.

The birth of software engineering is conventionally dated to a NATO conference convened in Garmisch, Germany in 1968 to consider the many problems faced by large software projects. The term *software crisis* was coined at this conference to describe the inability of software developers to complete many projects successfully. The conference was entitled "software engineering," a new term chosen to suggest that software developers needed to establish the same theoretical foundations and practical techniques as traditional engineers, a provocative idea at the time.

Software engineering and computer science remain closely affiliated, and most software engineering courses are taught in computer science departments, though there are a few software engineering department and degree programs around the world. Software engineering is still an emerging discipline.

Software Engineering as a Profession

Most disciplines with "engineering" in their names are recognized professions with legal status and rigorous certification required for practitioners. Software engineering does not have this status (an embarrassing state of affairs). However, efforts over the years have produced some of the artifacts characteristic of a recognized engineering profession, such as a code of ethics, a body of knowledge, and an accepted college curriculum. A survey of software engineering is part of the computer science and software engineering curricula; this book is the text for such a course. The material in this book is based largely on the Software Engineering Body of Knowledge (SWEBOK) [1].

The State of the Software Industry

Although software engineering aims to improve software development, and it has succeeded in doing so to some extent, developing software is difficult and many projects are not successful. In

fact, the software industry continues to have many failures, some of them quite spectacular. For example, consider the following recent findings from various groups studying the software industry.

- The Standish Group, which collects data on thousands of projects annually, found that in 2012 (a) 39% of all projects finished on time, on budget and delivered expected features and functions; (b) 43% were late, over budget, or failed to deliver all expected features and functions; (c) 18% failed outright, meaning that they were cancelled or that they delivered a product that was never used [2]. More recent findings from the Standish group are similar.
- A study of 5,400 large scale information technology (IT) projects (projects with initial budgets greater than $15M) found that (a) on average, large IT projects ran 66% over budget 33% of the time, while delivering 17% less value than predicted; (b) 17% of large IT projects went so badly wrong that they threatened the existence of the company [3].
- Interviews with 600 executives overseeing software development projects in 2010 and 2011 found that even at the start of a project 75% of them expected their projects to fail. Furthermore, 78% of respondents reported that project requirements were usually or always out of sync with business needs [4].
- A General Accounting Office study found that 413 federally funded IT projects with budgets totaling at least $25.2 billion in fiscal year 2008 were either poorly planned, poorly performing or both [5]. For example, Project Railhead, an effort to upgrade the National Counterterrorism Center's (NCTC's) terrorist watch list, failed utterly at a cost of $500M [6].

As these results show, software development project failures are quite common and very costly. On the other hand, failure rates and costs have declined steadily (though slowly) since the field began in 1968, so things are getting better. Though no one can prove it, it appears that this steady improvement is due to gradual refinement of software engineering methods and techniques, and their wider adoption and application across the software industry. But problems remain for the following reasons.

- Many software project managers have not been educated in software engineering or project management—poor management is the biggest source of project failures;
- Managers are unwilling to fund and staff projects adequately, and often demand results impossible to achieve in the time allotted—this is another management failure and amounts to an unwillingness to acknowledge reality;
- Even well-planned and well-run projects fail when they encounter technical obstacles or prove too ambitious—these are technical failures.

In short, knowing and applying good software engineering practices pays off.

More About Managerial and Technical Concerns

To begin our study of software engineering, we look in more detail at the things that software engineers must address in their work, both on the managerial and the technical sides. This brief overview of the topic introduces basic terminology and frames the more detailed discussions that follow in the later chapters.

Managerial Concerns

A **project** is a one-time effort to achieve a particular goal, subject to specific time or cost constraints. For example, building a doghouse, writing a term paper, or setting up a savings plan, are all projects. Ongoing activities, or activities that do not have any constraints, are not projects. So

running a household, maintaining a car, and the pursuit of happiness are not projects. Software is almost always built and maintained in software projects.

Management is the activity of assembling, directing, and supporting human and other resources in achieving substantial goals under time and cost constraints. Management is not necessary for trivial tasks or when time and money are not at issue, but it becomes increasingly important as goals, and the time and money needed to achieve them, become larger. Both ongoing activities and projects may need to be managed, but here we focus on managing projects.

There are several things that need to be managed in a project if it is to succeed, among them the following.

Scope—The **scope** of a project is the work to be done in the project. If scope is not controlled, then project goals may not be achievable within time and cost constraints. It is easy for scope to *creep*, that is for the work to increase gradually during the project until it gets out of hand. This happens on software development projects, for example, when developers add requirements unrequested by external stakeholders because they think it will improve the product.

Time—The time for project completion is a constraint that must be set at a reasonable level given the goals of the project, or the project will fail. Project participants must monitor progress and make adjustment to ensure that projects finish on time.

Cost—Like time, cost must be set at a reasonable level given the goals of the project, or the project will fail. Also like time, project participants must monitor costs and make adjustments to ensure that projects finish on budget.

Quality—The level of quality of a product or service may be thought of as one of the goals of the project; usually a product or service is not considered acceptable if its quality is poor. We can also think of the level of quality as an additional constraint. No matter how we think of it, however, acceptable quality is not achieved unless quality is assured throughout a project.

Resources—Resources (human and otherwise) must be obtained and be ready when needed during a project, and maintained and perhaps improved over the course of the project.

Risks—Things always go wrong. Projects are more likely to succeed if project participants anticipate risks, rule them out or reduce their likelihood as much as possible, and make plans to deal with them if they occur.

A important relationship exists between the first four of these items. The project management **iron triangle** is pictured in Figure 1.

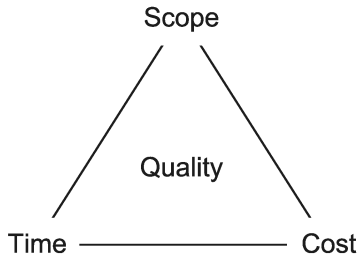

Figure 1: Project Management Iron Triangle

In a project one is attempting to achieve project goals (scope), finish by the due date (time), and finish the project on budget (cost). If scope increases (decreases), then time or cost must increase (decrease) as well. If time increases (decreases), then scope may increase (decrease) or cost may

decrease (increase). If cost increases (decreases), then scope may increase (decrease) or time may decrease (increase). Thus these three factors are strongly linked, as indicated by the lines in the triangle. It is assumed that quality will never be compromised (though it obviously can be), so it is pictured in the middle of the triangle.

Resources are closely linked with costs, for obvious reasons. Risks are a cross-cutting concern, because risks can arise with respect to scope, time, cost, quality, and resources.

Traditional and Agile Methods

Since its inception in 1968 software engineering has adopted many approaches and techniques from other disciplines, notably other engineering fields. Engineering management has typically used processes and management practices emphasizing careful planning, close oversight by managers, and thorough documentation; we call such processes and practices **traditional methods**. Since about 2000, different ideas have emerged that emphasize rapid adjustment to change, self-directed teams, and minimal documentation; these processes and practices are called **agile methods**. The field of software engineering is in the midst of a transition from traditional to agile methods. A recent large surveys found that about half of 3,000 projects used agile methods and about half used traditional methods [7]. Consequently we will discuss both because both are in use throughout the software industry.

Although there are several agile methods, the most popular is called *Scrum*. Currently about 70% of agile projects use Scrum [7] so it will be the focus our discussion of agile methods.

Management Responsibilities

Traditional project management assigns management responsibilities to individuals, called **project managers**, who are given authority to collect data and information, make decisions, and control resources. Agile methods devolve responsibility and authority to teams and rely more on processes that minimize the need for management oversight.

Traditionally, managers are responsible for the following activities.

Planning is the activity of setting goals, establishing constraints, setting requirements, determining tasks, estimating costs, formulating schedules, assigning resources, choosing quality assurance tools and techniques, and managing risks.

Execution is the activity of acquiring resources, training and educating people, establishing processes for doing work, monitoring work, monitoring and mitigating risks, and assuring quality.

Control is the activity of collecting and analyzing data, determining progress, reporting progress, and adjusting project scope, schedules, and budgets.

Leading is the activity of directing resources, motivating people, facilitating resolution of conflicts and problems, removing obstacles, and supporting people in their work.

In agile methods like Scrum, most of these responsibilities are taken away from managers and given to the team. Some planning activities are done when adjustments are made to the list of product features by the Scrum customer representative (the product owner). Scrum project facilitators (Scrum masters) perform most leadership activities. Traditional managers at levels above the Scrum team are still responsible for acquiring resources for Scrum teams, providing education and training, establishing some constraints, and setting business requirements. But everything else in the lists above is the province of the Scrum team.

We will consider roles and responsibilities in traditional and agile methods more in later chapters.

Technical Concerns

The first thing that software engineers must do when building or modifying a software product is to determine what the product must do; one cannot make something without a good idea of what to make. A software product **requirement** is some function, characteristic, or property that a software product must have. Product requirements can come from anywhere, but usually they come from whoever is paying for the product (**customers**) or whoever is using it (**users**). Other people, for example developers, managers, funders, and the public, may also have an interest in what a product does and how it does it, so all these **stakeholders** influence requirements. The development team uses various tools and techniques to elicit and record stakeholder needs and desires, record and analyze them, and eventually generate and document product requirements. We will learn some of these tools and techniques.

Having determined product requirements, developer are in a position to concoct a software system to satisfy them. A software **engineering design** is a specification of the parts of a software system, and their properties, relationships, and workings. Typically, software designs are large and complicated, so they are specified at several levels of abstraction from a variety of viewpoints. This requires various notations, several of which we will learn. Designers rely on a design process and design principles to choose between alternative deigns; we will study a generic design process and fundamental design principles. Designers also rely on standard solutions to problems, called **design patterns**, to help them make designs. We will study several high-level and mid-level design patterns.

Once software has been designed it can be implemented by writing code in some programming language. Besides the **compilers** and **interpreters** that realize languages, coders use tools to write code (**editors**), to keep track of code as it changes (**version control** tools), to make sure it follows standards and guidelines (**code checkers**), and to find and correct defects (**debuggers**). We will learn about such tools.

Quality must be assured during development. This can be done by adopting practices that help prevent defects. It also requires using various techniques to detect and correct defects in products, such as manual **reviews**, and **testing** code by executing it to see if its output is as expected. We will learn several review and testing techniques, as well as **unit testing** tools for testing code as it is written, and **code coverage** tools for ensuring most of the code in a program has been exercised by test cases. We will also study the capabilities and limitations of quality assurance activities and how to select a suite of quality assurance tools and techniques to achieve good results.

Finally, we will learn about **integrated development environments** that combine many tools into a single program that runs various tools automatically and coordinates and displays their outputs to enhance programmer productivity.

Summary

Software engineering is a discipline and an emerging profession concerned with the systematic and disciplined development, operation, and maintenance of high quality software products delivered on time, at minimum cost. As such it has both managerial and technical aspects. Managing a software project requires planning, execution, control, and leadership; we will study both traditional and agile techniques for managing a project. Technical work in a software project includes determining and recording stakeholder needs and desires, creating and documenting an overall software design, coding the product, and finally delivering and maintaining it. Along the way developers must ensure quality by both preventing defects and by detecting and removing them. We will study this development process and the tools and techniques used to make projects succeed.

Software engineering difficult and project often fail or fall short of their goals. However, application of software engineering tools and techniques has increased the likelihood of project success over several decades, so it is worthwhile to study software engineering and apply its findings in software development.

References

1. Pierre Bourque and Richard E. (Dick) Fairley, *Guide to the Software Engineering Body of Knowledge, Version 3*. IEEE, 2014. (www.swebok.org)
2. The Standish Group, *CHAOS Manifesto 2013*. February, 2013. (www.standishgroup.com)
3. McKinsey & Company, "Delivering Large-Scale IT Projects On Time, On Budget, and On Value." http://www.mckinsey.com/insights/business_technology/delivering_large-scale_it_projects_on_time_on_budget_and_on_value, accessed July 2, 2014.
4. Geneca, "Up to 75% of Business and IT Executives Anticipate Their Software Projects Will Fail." http://www.geneca.com/75-business-executives-anticipate-software-projects-fail/, accessed July 2, 2014.
5. General Accounting Office, "OMB and Agencies Need to Improve Planning, Management, and Oversight of Projects Totaling Billions of Dollars." http://www.gao.gov/new.items/d081051t.pdf, accessed July 2, 2014.
6. Patrick Thibodeau, "US Terror Threat System 'Crippled' by Technical Flaws." *ComputerWorld UK*, August 28, 2008. http://www.computerworlduk.com/news/applications/10723/us-terror-threat-system-crippled-by-technical-flaws/, accessed July 2, 2014.
7. Donald J. Reifer, "Quantitative Analysis of Agile Methods Study (2017): Twelve Major Findings." *InfoQ*, August 10, 2017. https://www.infoq.com/articles/reifer-agile-study-2017.