CAMBRIDGE

Cambridge University Press & Assessment
978-1-316-51194-7 — Software Engineering
Ravi Sethi
Excerpt
More Information

1 Introduction

Software engineering is the application of engineering methods to software development and evolution. Its principles and practices address three fundamental goals: discover user requirements, manage software complexity, and build quality products and services. This chapter introduces the goals, their associated challenges, and how to deal with the challenges.

The main requirements goal is to pin down what users really want, so developers know what to implement. The main complexity goal is to overcome the intrinsic complexity of software, primarily through the design of modular systems. The main quality goal is to build dependable systems, despite the inevitability of defects. Defects are detected and removed by software checking and testing.

These software engineering goals are met by using systematic quantitative methods (processes) to organize development teams and their activities; see Chapter 2 for processes. Teams must balance the scope (functionality) they deliver against budget, schedule, and other external constraints. This chapter concludes with a brief discussion of professional conduct and social responsibility.

This introductory chapter will enable the reader to:

- Describe approaches to addressing the fundamental software engineering goals/challenges of identifying user requirements, managing software complexity, and achieving product quality.
- Explain the elements of the ACM/IEEE Software Engineering Code of Ethics and Professional Practice (short version).

1.1 What Is Software Engineering?

Most definitions of software engineering are variations on the theme of applying engineering to software. In other words, they focus on the application of systematic processes to create and maintain software products. Left unstated in most definitions is the fact that engineering projects have customers (users) and face business and regulatory constraints. As we shall see, customers and constraints are major driving forces (drivers) for projects. The definition in this section includes them explicitly.

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE

Cambridge University Press & Assessment
978-1-316-51194-7 — Software Engineering
Ravi Sethi
Excerpt
More Information

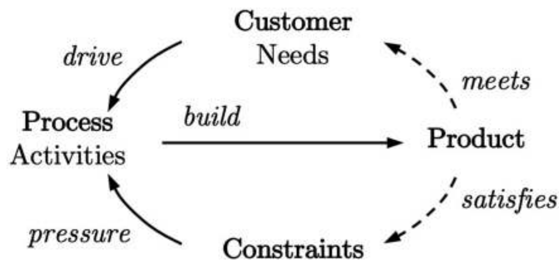

Figure 1.1 The arrows represent relationships between four pervasive drivers of software projects: Customers, Processes, Products, and Constraints.

The definition therefore includes four key drivers: customers, processes, products, and constraints.

This book uses the term *development* broadly to extend past initial development to changes after a product is created. Initial development spills over into evolution/maintenance because products continue to be updated long after they are first deployed. The distinction between development and evolution disappears with the practice known as continuous deployment; see Example 1.1.

1.1.1 Definition of Software Engineering

Software engineering is the application of systematic, quantifiable processes to the development and evolution of software products for customers, subject to cost, schedule, and regulatory constraints.[1]

The arrows in Fig. 1.1 illustrate the relationships between the four key drivers in the preceding definition. Customer needs are at the top in the figure – without them, there would be no project. Customer needs drive the process activities that build the product. Once built, the product must meet customer needs and satisfy any constraints on the product, the process, or the project.

Driving Forces on Software Projects

The four key drivers in Fig. 1.1 are representative of the many forces that a project must balance. The four key drivers appear in boldface in Fig. 1.2.

Customers and users belong in the same grouping of forces. In popular usage, the target audience for a product is called “the customer” or “the user.” Both roles drive requirements for what to build. We therefore use the two terms interchangeably, unless there is a need to be specific about a role.

Processes and teams are closely tied, since processes organize teams and their activities. Team skills belong in the same grouping.

Product is a convenient term for any deliverable from a software project. Examples of products include software systems, services delivered from the cloud, test suites,

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE

Cambridge University Press & Assessment
978-1-316-51194-7 — Software Engineering
Ravi Sethi
Excerpt
More Information

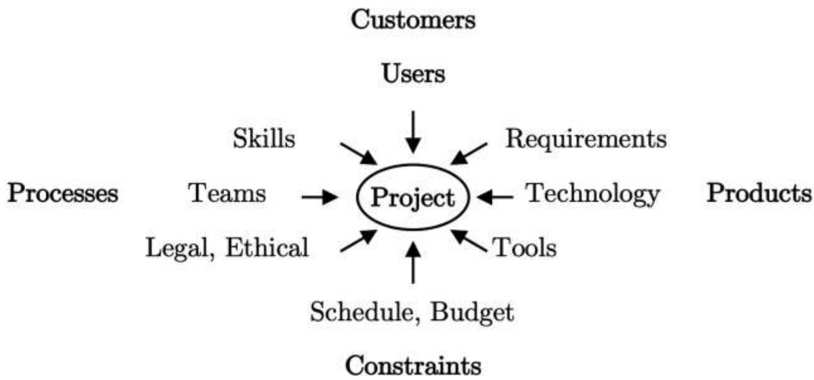

Figure 1.2 Some of the many drivers that projects must balance. Key drivers are in bold.

and documentation. A deliverable can also be an event, such as a live demo at a conference.

Products and technology are a natural fit. Technology takes two forms: (a) it is delivered in the form of products, and (b) it is used in the form of tools to build products.

Constraints are external forces from the context of a project. For example, European Union regulations prohibit products from exporting personal data. Cost, time, and legal constraints are mentioned explicitly in the definition of software engineering. Projects must also deal with ethical and social norms.

Box 1.1 Origins of the Term Software Engineering

Software engineering emerged as a distinct branch of engineering in the 1960s. The term "software engineering" dates back to (at least) 1963–64. Margaret Hamilton, who was with the Apollo space program, began using it to distinguish software engineering from hardware and other forms of engineering. At the time, hardware was precious: an hour of computer time cost hundreds of times as much as an hour of a programmer's time.[2] Code clarity and maintainability were often sacrificed in the name of efficiency. As the complexity and size of computer applications grew, so did the importance of software.

Software engineering was more of a dream than a reality when the term was chosen as the title of a 1968 NATO conference. The organizers, especially Fritz Bauer, chose the title to be

> provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering.[3]

Since then, much has changed. On November 22, 2016, Margaret Hamilton was awarded the Presidential Medal of Freedom for her (software) work leading up to the Apollo missions.

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE

Cambridge University Press & Assessment
978-1-316-51194-7 — Software Engineering
Ravi Sethi
Excerpt
More Information

1.1.2 A Tale of Two Companies

The key drivers – customers, processes, products, and constraints – are pervasive in the sense that a significant change in any one of them can affect every aspect of a project. The two examples that follow illustrate the far-reaching effects of forces related to customer needs and regulatory constraints. The examples illustrate how different situations drive dramatically differing decisions, as measured by the pace of the projects. One company chooses continuous deployment, the other chooses semiannual releases.

Example: Rapid Response to Customer Needs

In the following example, a technology company wants to respond rapidly to any change in customer needs. This goal drives every aspect of the company, from the development process to product design to organization structure to the reward system for employees.

Example 1.1 In order to respond rapidly to customer suggestions and feedback, a tech company updates its software frequently: it makes hundreds of changes a day to the software on its main servers. Each change is small, but the practically "continuous" small updates add up.

Each change is made by a small team that is responsible for "everything" about the change: design, coding, testing, and then deploying the software directly on customer-facing servers. The change cycle from concept to deployment takes just a few days. There are multiple teams, working in parallel. Together, they make hundreds of updates a day.

The cost of a misstep is low, since the company can trial an update with a few customers and quickly roll back the update if there is a hitch. After a short trial period, the updated software can be put into production for the entire customer base.

Every aspect of the preceding description is driven by the company's desire to respond rapidly to its customers. Rapid response implies that the cycle time from concept to deployment must be very short. Developers are responsible for making and deploying their changes directly onto live production servers, since there is no time for a handoff to a separate operations team. The short cycle time also means that changes are small, so work is broken down into small pieces that can be developed in parallel. Management support is essential, especially if something goes wrong and an update has to be rolled back.

In short, the company's whole culture is geared to its responsiveness to customer needs. □

Example: Regulations and the Pace of Deployment

In the following example, strict regulations on large banks ripple through the banks to their suppliers. The cost of a misstep is high, so both the banks and their suppliers do extensive testing. The time and cost of testing influence software release schedules.

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE

Cambridge University Press & Assessment
978-1-316-51194-7 — Software Engineering
Ravi Sethi
Excerpt
More Information

Example 1.2 A supplier of business software releases software semiannually, on a schedule that is set by its large customers, including highly regulated investment banks. Some of the bigger customers conduct their own rigorous acceptance tests in a lab, before they deploy any software from a supplier. The trial-deploy cycle takes time and resources, so the customers do not want small updates as soon as they are available; they prefer semiannual releases.

The supplier's development projects are geared to the customers' preferred release cycle. Since releases are relatively infrequent, they include more functionality to be included and tested. Dedicated product managers stay in close touch with major customers to ensure that a release includes what they want. □

Assessment: The Impact of a Change

What can we learn from Examples 1.1 and 1.2? Using the pace of release/deployment as a measure, the examples illustrate the range of projects that software engineering can handle. Both the tech company and the supplier of business software chose solutions that were best for their specific situations.

Based on his experience with large-scale software systems at Google, Jeff Dean advises that a tenfold change in any aspect of a project is a time to revisit the existing solution: it may no longer work as well, or it may no longer work at all due to the change. A hundredfold change may require an entirely new solution approach, perhaps even a new organizational structure.[4] There is well over a thousandfold difference in the pace at which the companies in Examples 1.1 and 1.2 release/deploy software.

1.2 The Requirements Challenge

Consider the following scenario. A development team talks to customers to identify their needs for a product. The developers then build a product to meet those needs, only to find that the completed product does not meet customer expectations. What went wrong? Customer needs "changed" while the developers were building the product. With respect to Fig. 1.3, needs identified at the beginning of a cycle can differ from needs at the end of the cycle.

A *user requirement* is a description of a customer/user need or want for a product. The *requirements challenge* is to define a coherent set of requirements that meets user expectations. There are two aspects to the challenge:

- Identify and prioritize user requirements that truly reflect user needs. The challenges include multiple users with differing needs and communication gaps between users and developers.
- Accommodate changes in user requirements. Users can change their minds, either because they are uncertain or because of some unexpected external event.

The requirements challenge is also referred to as the problem of *changing requirements*.

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE

Cambridge University Press & Assessment
978-1-316-51194-7 — Software Engineering
Ravi Sethi
Excerpt
More Information

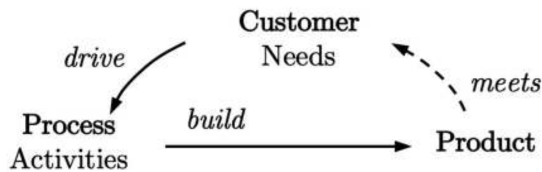

Figure 1.3 The cycle represents the iterative nature of product development based on customer needs.

1.2.1 Identifying Users and Requirements

To identify the requirements for a product, we can begin with the following questions:

- Who is the customer? There will likely be multiple classes of users, each with their own needs and goals.
- What do users really want? There can be gaps between what users are willing and able to communicate and what developers are able to grasp.
- Are there any external factors? Changes in external conditions can have repercussions on the project.

Any one of these situations can result in a mismatch between customer expectations and a snapshot of needs at the start of a project.

Who Is the Customer?

Different users can have different, perhaps conflicting, needs and goals for a project. These differences have to be identified, reconciled, and prioritized into a coherent set of requirements. If any class of users is missed or inadequately represented, the requirements will not match the needs and goals of the user community. The mismatch can result in later changes to the requirements. Part of the problem is that it may not be possible to satisfy all users.

Example 1.3 The makers of a speech-therapy app found a creative way of meeting the differing needs of two classes of users: parents and children. What was the problem? Market testing with a prototype revealed that parents really wanted the program for their children, but found it frustratingly hard to use. Children, meanwhile, had no trouble using the program but could not be bothered with it. They found it annoyingly like a lesson.

The two classes of users were therefore frustrated parents and annoyed children. Their differing needs were addressed by changing the program so it was easier to use for parents and was more like a game for children.[5] □

The term "stakeholder" is sometimes applied to a class of users. In general, a *stakeholder* is anyone with a stake in a project. Stakeholders include developers and marketers. For requirements, the focus is on users and their needs.

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE

Cambridge University Press & Assessment
978-1-316-51194-7 — Software Engineering
Ravi Sethi
Excerpt
More Information

What Do Users Really Want?

What customers say they want can differ from what will satisfy them. There may also be needs that they are unable to articulate. Even if they did know their needs, they may have only a general "I'll know it when I see it" sense of how a product can help them.

Example 1.4 Netflix is known for its video and on-demand services. Their experience with their recommender system is as follows:

> Good businesses pay attention to what their customers have to say. But what customers ask for (as much choice as possible, comprehensive search and navigation tools, and more) and what actually works (a few compelling choices simply presented) are very different.[6] □

Uncertainty about user needs is a known unknown, which means that we know that there is a requirement, but have yet to converge on exactly what it is. Uncertainty can be anticipated when proposing a solution or design.

Unexpected Requirements Changes

Unexpected changes are unknown unknowns: we do not even know whether there will be a change. Here are some examples:

- A competitor suddenly introduces an exciting new product that raises customer expectations.
- The customer's organization changes business direction, which prompts changes in user requirements.
- Design and implementation issues during development lead to a reevaluation of the project.
- The delivered product does what customers asked for, but it does not have the performance that they need.

Requirements changes due to external factors can lead to a project being redirected or even canceled.

1.2.2 Dealing with Requirements Changes

Changes in customer needs and requirements have repercussions for the development process and for the product, because of the relationships shown in Fig. 1.3. We have no control over changes in customer needs, but there are two things we can do.

1. Do as good a job as possible of identifying and analyzing user requirements. Requirements development is a subject in its own right; see Chapter 3.
2. Use a development process that accommodates requirements changes during development. In effect, iterative and agile processes go through the cycle in Fig. 1.3 repeatedly, evolving the product incrementally. Each iteration revisits customer needs to keep the product on track to meeting customer expectations when it is completed.

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE

Cambridge University Press & Assessment
978-1-316-51194-7 — Software Engineering
Ravi Sethi
Excerpt
More Information

1.3 Software Is Intrinsically Complex

Let us call a piece of software *complex* if it is hard to understand, debug, and modify. Complexity due to poorly written code is *incidental* to the problem that the code is supposed to address. Meanwhile, if well-written code is hard to understand, its complexity is due to the algorithm behind the code. This algorithmic complexity will remain even if we rewrite the code or use another programming language, because it is *intrinsic* to the problem; that is, it is due to the nature of the problem.[7]

We focus on intrinsic complexity. Architecture is a primary tool for managing software complexity. A software architecture partitions a problem into simpler subproblems. Layered architectures are used often enough that we introduce them in this section.

1.3.1 Sources of Complexity

The two main sources of complexity are scale and the structure/behavior distinction.

- **Scale** Program size is an indicator of the scale of a problem. A large software system can have tens of millions of lines of code. Sheer size can make a system hard to understand.
- **Structure versus Behavior** Here, structure refers to the organization of the code for a system. Behavior refers to what the code does when it is run. The challenge is that behavior is invisible, so we do not deal directly with it. We read and write code and have to imagine and predict how the code will behave when the code is run.

Example 1.5 As a toy example of the distinction between structure and behavior, consider the following line from a sorting program:

```
do i = i+1; while ( a[i] < v );
```

In order to understand the behavior of this loop, we need to build a mental model of the flow of control through the loop at run time. The behavior depends on the values of `i`, the elements of the array `a`, and `v` when the loop is reached. Control flows some number of times through the loop before going on to the next line. If any of these values changes, the number of executions of the loop could change.

The structure of this loop identifies the loop body and the condition for staying in the loop. The behavior of the loop is characterized by the set of all possible ways that control can flow through the loop. □

The single well-structured `do-while` loop in the preceding example was convenient for introducing the distinction between program structure and run-time behavior. Complexity grows rapidly as we consider larger pieces of code; it grows rapidly as decisions

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE

Cambridge University Press & Assessment
978-1-316-51194-7 — Software Engineering
Ravi Sethi
Excerpt
More Information

are added, as objects are defined, as messages flow between parts of a program, and so on.

To summarize, scale and the predictability of run-time behavior are significant contributors to software complexity.

1.3.2 Architecture: Dealing with Program Complexity

Informally, a *software architecture* defines the parts and the relationships among the parts of a system. In effect, an architecture partitions a system into simpler parts that can be studied individually, separately from the rest of the system. (See Chapter 6 for a more precise definition of architecture.)

In this section, the parts are *modules*, where each module has a specific responsibility and a well-defined interface. Modules interact with each other only through their interfaces. The complexity of understanding the whole system is therefore reduced to that of understanding the modules and their interactions. For the interactions, it is enough to know the responsibilities of the modules and their interfaces. The implementation of the responsibilities can be studied separately, as needed.

What is inside a module? The program elements in a module implement its responsibility and services. As long as its interface remains the same, the internal code for the module can be modified without touching the rest of the system. A module can be anything from the equivalent of a single class to a collection of related classes, methods, values, types, and other program elements. This concept of module is language independent. (Modules are closer to packages in Java than they are to classes.)

For more information about modules, see Chapter 6. In particular, modules can be nested; that is, a module can have submodules. For example, a user-interface module may have submodules for handling text, images, audio, and video – these are all needed for a user interface.

Layered Architectures

For examples, let us turn from a general discussion of architecture to a specific form: layered architectures, which are widely used. In a *layered architecture*, modules are grouped into sets called *layers*. The layers are typically shown stacked, one on top of the other. A key property of layered architectures is that modules in an upper layer may use modules in the layer immediately below. Modules in a lower layer know nothing about modules in the layers above them.

Example 1.6 The layered architecture in Fig. 1.4 is a simplified version of the architecture of many apps. At the top of the diagram is the Presentation layer, which manages the user interface. Below it is the Domain layer, which handles the business of the app. In a ride-sharing app, the Domain layer would contain the rules for matching riders and drivers. Next is the Service Access layer, which accesses all the persistent data related to the app; for example, customer profiles and preferences. At the bottom, the Platform layer is for the frameworks and the operating system that support the app.

© in this web service Cambridge University Press & Assessment www.cambridge.org

Cambridge University Press & Assessment
978-1-316-51194-7 — Software Engineering
Ravi Sethi
Excerpt
More Information

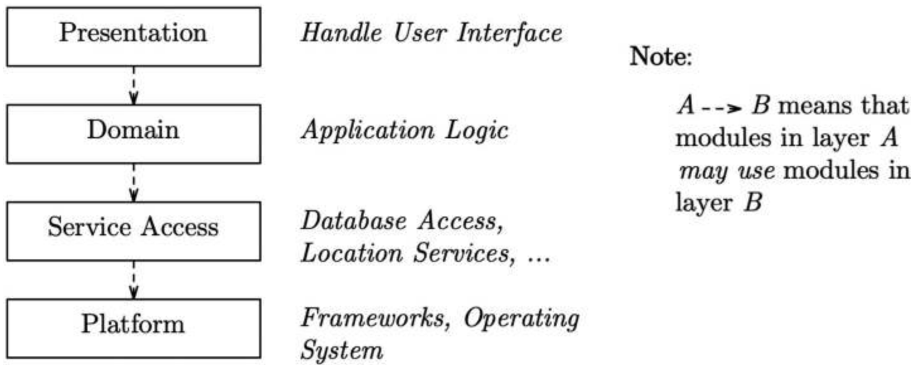

Figure 1.4 A simplified version of the layered architecture of many apps. Boxes represent layers containing modules, and dashed arrows represent potential dependencies between modules in an upper layer on modules in a layer just below.

The dashed arrows represent the may-use relation. Modules in the Presentation layer may use modules in the Domain layer, but not vice versa; that is, modules in the Domain layer may not use modules in the Presentation layer. Similar comments apply to the other layers. By design, all the arrows are down arrows from one layer to the layer immediately below it.. □

The arrows in Fig. 1.4 are included simply to make the may-use relationships explicit. Vertical stacking of layers can convey the same information implicitly. From now on, such down arrows will be dropped from layered diagrams. By convention, if layer A is immediately above layer B in a diagram, modules in the upper layer A may use modules in the lower layer B.

Example: Modules from an App

The next example is about the modules in a specific app that has a layered architecture like the one in Fig. 1.4.

Example 1.7 A local pool-service company has technicians who go to customer homes to maintain their swimming pools.[8] Each technician has an assigned route that changes by day of the week. At each home along their route, the technicians jot down notes for future reference. The company wants to replace its current paper-based system with a software system that will support a mobile app for technicians and a web interface for managers. Managers would use the web interface to set and assign routes to technicians. Technicians would use the mobile app for individualized route information, data about each pool along the route, and some customer account information.

The main modules in the solution correspond to the layers in Fig. 1.4.

© in this web service Cambridge University Press & Assessment www.cambridge.org