# Contents

1 Engineering Software 5

1.1 What is Software? .................................................. 5
1.2 What is Engineering? .............................................. 7
1.3 What is Software Engineering? .................................. 7
1.4 The Need for Software Engineering ................................. 9
1.5 Essential Tools for Engineering Software ......................... 12
   1.5.1 Organizing and Managing Software ........................ 12
   1.5.2 Reading and Writing Software Artifacts .................. 13
   1.5.3 Building Software ........................................... 13
   1.5.4 Running Software ........................................... 14
   1.5.5 Analyzing Software ......................................... 14
1.6 Essential Skills & Knowledge for Engineering Software ........... 14

2 Computational Thinking in Software Engineering 17

2.1 Introduction to Computational Thinking ............................ 17
2.2 Computational Thinking Activities ................................. 19
   2.2.1 Problem Formulation ....................................... 19
   2.2.2 Solution Design and Expression ............................ 19
   2.2.3 Solution Application ....................................... 21
   2.2.4 Solution Analysis .......................................... 21
2.3 Techniques .......................................................... 21
   2.3.1 Pattern Identification ...................................... 21
   2.3.2 Abstraction .................................................. 21
   2.3.3 Refinement ................................................... 22
   2.3.4 Decomposition .............................................. 22
   2.3.5 Composition .................................................. 22
Chapter 1

Engineering Software

Originally a sub-field within computer science, software engineering has recently become a discipline in its own right. It is a field in which both computing and engineering principles are applied to the development and maintenance of software systems. The tools and practices employed by software engineers enable large teams of software developers and other business and software specialists, e.g., designers, architects, test engineers to work collaboratively to design, build and maintain the large, complex software systems in place today (and which are likely to grow in size and complexity in the coming years). The field of software engineering is an important area of study. As a relatively young engineering field (only about 50 years old), software engineering practices are still evolving as we learn how to build software more efficiently and more effectively. At the same time, the technologies we use to build and maintain software are evolving and the nature of software itself is evolving to encompass applications for more diverse types of computing devices and to account for changing expectations of what software can and should do.

This chapter explores what it means to “engineer software” and why we need software engineering techniques and tools. It also provides a brief introduction to the essential software engineering skills and tools you will need in order to become a successful software engineer. At the end of this chapter, you should be able to:

1. Explain what software is and why it is not really ‘soft’.
2. Describe the high-level tasks performed by software engineers beyond writing code.
3. Explain why software engineering is essential for building correct, efficient and robust software.
4. Describe the essential skills necessary to be a successful software engineer.

1.1 What is Software?

Software is at the heart of nearly every system in modern day life, from the phones in our pockets to the appliances in our homes. For example, consider the software on a laptop or a phone. Each software application (i.e., app or program) provides a service or controls the device in some way. We use software from the time we wake up in the morning (e.g., an alarm-clock app) until we go to bed at night (e.g., a music app). We also use software to communicate with family, keep up with current events, and to socialize with friends who may live in another location. Sometimes we use software in less obvious ways. For instance, software regulates the engine, manages the brakes, and triggers the safety features in nearly every car built today. Software is also used to control traffic-light systems, medical devices, and the lights, heating, air conditioning and door locks in many office and commercial buildings. In each of these examples, software is an enabling
technology that has improved on or replaced another technology, or replaced a previously manual process. Software can also be used to improve existing technologies. For example, software in headphones can cancel background noise, and software in a camera can autofocus an image.

To better understand the many different types of software in use today, it is helpful to discuss categories of software. One approach to categorizing software is to consider its function. Using this schema, we can classify software into three general categories: system software, application software, and software libraries. **System software** is software that directly operates the computer hardware. For example, the operating system on your laptop is a type of system software that is actually a collection of software programs that manage the resources on your laptop (e.g., storage space) and provide common resources for other software running on the device. Examples of operating systems include Linux, Android, Mac OS, and Microsoft Windows. Users typically do not interact directly with system software. Instead, they work primarily with application software. **Application software** describes the family of programs that perform a specific task beyond the basic operation of the computer itself. Each Android or iOS app can be thought of as application software. Other applications include spreadsheets, word processing programs, and computer games. Special kinds of application software, referred to as **plugins** or **extensions**, modify or add functionality to another program. The third category of software is referred to as **software libraries**. This type of software plays a supporting role in system and application software by providing a solution to a specific problem that is available through a well-defined **Application Programming Interface (API)**. Software libraries are built for reuse, but are generally not intended to be used stand-alone (i.e., they are intended to be used only by other software). **Software frameworks** are a special kind of software library. One key difference between a framework and a regular software library is the locus of control; in the case of a library, the software calling the library retains control of how the program executes, but in the case of a framework, the software using the framework cedes control of how the program executes to the framework.

We can also classify software according to characteristics of the hardware it is written for and the domain in which the software is used. For example, **Embedded system** software applications reside as firmware on a device dedicated to a particular use, e.g., inside a car, aircraft or medical device. Embedded system software is often developed to operate in real-time, meaning that it must meet specified time constraints to support mission-critical applications such as antilock brakes or pacemakers. These programs require careful instruction selection to ensure that deadlines are not missed or missed very infrequently. **Mobile apps**, such as those running on a phone or tablet, are another type of software. Development of these apps often must take into account the limited memory and power available on the mobile device. **Distributed or parallel software applications** are developed to solve problems that have been divided into many tasks. Each task is performed on a different processor, where the processors may or may not reside on the same computer or share the same memory space. These programs require additional instructions to coordinate the tasks.

Up to this point, we have talked about software in general terms, but what is it that we are really referring to when we use the term **software**? Historically, the ‘soft’ in software referred to the relative ease with which **source code** could be changed as opposed to the electronic components, or **hardware**, which is more difficult to alter because reprogramming entails physical modifications to the electronics. In recent years, software has taken on new meaning to reflect the wide and diverse range of systems that can be modeled using computational resources. The definition of software has also evolved to account for the many **artifacts**, in addition to source code, that are created during the design, development, testing, deployment, and maintenance of computing and electronic systems. For instance, some of the non-code software artifacts that may be developed as part of the engineering process include:

- **Specifications** that describe the system that is to be developed,

---

1 In the following sections we will explore how in practice, modern-day software is not as ‘soft’ as one might think. While the act of modifying one or more lines of code in an editor is trivial, the work behind the scenes to determine what line or lines of code should be changed and how to change them—not to mention the effort to ensure that these modifications both accomplish their goal and avoid other, unintended ramifications—can require significant time and effort.
Project Planning and Tracking Documents that help developers determine and record the scope, schedule, and resources necessary for the project, and then to track project progress,

Build Scripts that are used to specify how the software is compiled, packaged, re-built, etc.,

Software Design Documents that describe features of the design and supporting information such as constraints, assumptions, dependencies, and other information to help others understand the design or reasons for why a particular design was chosen,

Test Cases that help assess the quality of software applications (and may or may not be written as code), and

User Guides that help users understand features and changes to the software application.

In future chapters, we will look more closely at the various types of software artifacts and how each contributes to the engineering of correct, reliable and efficient software systems.

1.2 What is Engineering?

The term engineering is derived from the Latin term in gignere, meaning “to contrive or devise.” It is a creative activity that involves the application of mathematical and scientific principles to the design, development, maintenance and improvement of everything from physical structures, to processes that define how something is built, to software. Engineers use their knowledge of math and science to find practical and cost-effective solutions to real-world problems. Prior to constructing a solution (e.g., a building or a process) engineers will often first create a mathematical model of the problem. Potential solutions are then analyzed using the model to evaluate different design choices. Disciplined and rigorous processes are used throughout the design and building phases to help ensure the solution works as intended and that it is safe and maintainable.

A crucial aspect of engineered solutions is that they are designed and built with a variety of situations that may impact the integrity or functionality of the end product in mind. For instance, a structural engineer designing a bridge will prepare the bridge to withstand various loads and weather conditions. They will include tolerances to account for possible construction flaws and consider how future maintenance will affect the bridge’s properties. They will also develop a good understanding of what situations they haven’t prepared for, situations in which it would be unsafe to trust the bridge. Similarly, a software engineer will consider safety, security, reliability, usability, performance, and other quality dimensions when designing and building a software system.

1.3 What is Software Engineering?

Now that we have defined the terms software and engineering, we can begin to explore software engineering as a field of study and practice. As you might expect, software engineering is the disciplined application of math and scientific principles and processes to the design, development, maintenance, and improvement of software. In the early days of computing and software development, software engineering practices received much less attention. Initially, software applications were often small and written by an individual or a small number of programmers. Over time however, software became more pervasive, more complex, and required larger and larger teams to design, develop, and maintain. Eventually, the use of engineering practices and techniques became essential in order to efficiently develop robust and maintainable software. Although software is not always delivered on time or within budget even today, the software engineering technologies and tools currently available have enabled software development teams to build, deploy and maintain complex, reliable and safe software systems that would not have been possible just a few short years ago.
Most engineering disciplines are physics-based and involve the engineering of physical objects. For example, a closely related field to software engineering is the field of computer engineering. Like software engineers, computer engineers are concerned with the engineering of software; however, computer engineers are also interested in the hardware components. Computer engineers study many of the same topics as computer scientists and software engineers, such as data structures and algorithm design. They also study how electronic components are designed and integrated, and how software is designed, built, and optimized for specific hardware components. Computer engineers design and build custom systems ranging from the embedded systems found in cars, aircraft, video games, medical devices, and just about every electronic device in our homes, to super computers and clusters of computers that are the backbone of nearly every business organization.

The relatively new field of software engineering can be viewed as an information-based discipline, rather than a physics-based engineering discipline. Software engineering is focused on the engineering of processes to efficiently organize and process representations that characterize potentially large corpora of data. Closely related to the field of computer science, which covers the core concepts and techniques for making a computer do something, software engineering is focused on the processes and techniques for designing, building, and maintaining software. To better understand the relationship between computer science and software engineering, consider the relationship between chemistry and chemical engineering. Chemists study the structure of chemicals, how they interact, and the theory behind chemical behaviors. Chemical engineers develop familiarity and expertise in chemistry so that they can apply the results of what chemists have discovered to solve a variety of problems that occur in the real world. Thus the relationship between the study of chemistry and the study of chemical engineering is one where chemical engineers view chemistry as a tool to solve problems, whereas chemists view chemistry as the object of study.

We can see a similar relationship between computer science and software engineering. Computer science courses focus on the theory of how computers and networks work and how they can be controlled through computer applications, as well as the fundamentals of algorithms and data structures. Computer science courses also teach us how to use computers in very sophisticated ways to build operating systems and databases, perform image processing, control robots, solve complex scheduling tasks, write algorithms that can be run in parallel or on distributed computers, and much more. Software engineers study many of these same topics to develop familiarity and expertise with computing concepts. They also study topics specifically targeting engineering aspects of how software is designed, built, and maintained in a systematic, controlled, and efficient manner, such as software testing and analysis, software architecture, and requirements engineering. In other words, software engineers study techniques and tools for applying what computer scientists have discovered to real-world problems.

A common misconception about software engineering, and software development in general, is that developers spend most of their time programming, i.e., writing code. In reality, writing code is only one of the many tasks performed by software engineers and software developers. Software engineers often spend time researching ways to solve problems and planning how they will change the code. They also read code to understand the current functionality and how it can be changed, or how it may be impacted by a change. They design testing strategies, write test cases and run the tests to determine if the changes have the intended results. They also spend time meeting with clients or team members to discuss software changes and new functionality. Software engineers also spend time tracking down software bugs, and reviewing changes made by other developers in order to provide feedback on the correctness and efficiency of the changes. The set of tasks performed by a software engineer varies from project to project depending on the needs of the project and may include other technical and management tasks.

Now that we have identified some of the common tasks performed by an individual software engineer, let’s take a step back and look at how these tasks fit into the overall process of engineering software. Software engineering processes vary from project to project, depending on which software development methodology, is followed by the team in planning, building, analyzing and deploying the software. Before exploring a couple of specific processes, let’s first identify the core set of activities common to all software engineering processes:

- **Requirements specification and analysis** activities identify what the software should do.
• System architecture and design activities determine how the software should be partitioned into modules and how the modules interface (i.e., interact) with one another.

• Construction activities are primarily focused on writing or generating code.

• Testing and analysis activities check the software system for correctness, maintainability, performance, security, and other qualities.

• Deployment activities prepare the software for delivery and distribute the software to the target computer.

• Maintenance activities keep the software running once it is deployed.

Although there appears to be a natural ordering to the activities listed above (e.g., we cannot write software if we do not know what it is supposed to do), there are actually many ways to organize and perform these activities. Hence, there are multiple software development methodologies that attempt to apply mathematical and scientific principles to the design, development, maintenance, and improvement of software. For instance, the waterfall model is a sequential model where the core set of software development activities is treated as a sequence of phases. The phases may overlap, but the main idea is that the requirements are known and specified early in development and that they remain relatively unchanged. One of the primary problems with software developed following this approach is that all of the requirements are rarely known upfront which, historically, has led to costly delays and rework as new requirements were discovered during latter activities. Another problem is that delaying the majority of the testing activity until late in the process increases the cost of fixing the problems that are detected during testing. The waterfall model has largely been superseded by processes that rely on multiple iterations of the activities listed above to incrementally build the system.

Evolutionary software development methodologies combine iterative and incremental approaches to software development. The basic idea behind evolutionary software development is to construct software by repeatedly adding small, manageable parts, performing all or most of the activities on each iteration. It also involves refining parts of the system as more information becomes available. This approach mitigates both of the problems mentioned above. First, it enables requirements and information about the system to be discovered as the software is designed, built, and tested rather than expecting the client to know all of the requirements upfront. Evolutionary software development also incorporates testing and analysis activities early in the development cycle, when it is typically easier and less costly to fix problems with the code. Both of these features of evolutionary development can reduce the cost of rework significantly. Agile software development takes evolutionary development a step further by explicitly recognizing the collaborative nature of software development and relying on continuous feedback from the project stakeholders.

Up to this point, our discussion has mainly focused on the technical activities related to engineering software. However, there are also many important non-technical activities you will perform as a software engineer, including project planning and tracking, identifying and managing risks, communicating with project stakeholders, and documenting various parts of the project. In future lessons we will explore both types of activities in more detail.

1.4 The Need for Software Engineering

Software has a reputation for being late, over budget, and buggy (i.e. unreliable or incorrect). Sometimes, it is even scrapped before it is delivered. However, many software systems have been successfully deployed and maintained for many years. Well engineered software, just as a well-engineered bridge or building, is not guaranteed to be successful. However, the application of rigorous and disciplined engineering practices can help offset many of the situations that keep projects from being successful. In this section we explore several aspects of software and software development that motivate the use of software engineering practices
to help improve the likelihood of delivering a robust, reliable and maintainable software system on time and within budget.

In some respects, engineering software is similar to engineering a physical structure, such as an office building. Like an office building, software is large and complex. It is difficult to imagine the size or complexity of software until you begin to study a real-world project. Modern software systems consist of huge numbers (hundreds of thousands and even millions) of lines of code, spread across many (hundreds and even thousands of) modules and packages. Many systems are written in multiple languages, and almost all software in place today relies on one or more libraries or frameworks. Recall too, that software is more than just code; it is a complex set of artifacts that include plans, design documents, specifications, tests, scripts, and many other types of artifacts that are created to ensure the software system operates correctly and efficiently. Disciplined software engineering processes and supporting tools are necessary to effectively organize, coordinate, and manage large complex software systems.

Engineering software is also different from engineering a physical structure. One critical difference is that software is developed as a series of changes and is expected to be malleable, or easily changed. However, changing software can be risky. Source code changes in particular have the potential to affect the way the application executes or operates. Non-source code changes typically do not affect how the application works, but they can. For example, a correction to the user documentation or a change to a comment in the code will not affect the operation of the application, but a incorrect edit to a build script could reference the wrong version of a library and ultimately introduce bugs.

In a previous section, we learned that, relative to changing hardware, software is easier to change. While this is true to the extent that making a change to the source code is easy, it is not the act of making the change that is difficult or costly, but the preparation for the change and the process of ensuring the change does not adversely affect other parts of the software that is costly. There are also costs associated with maintaining consistency across software artifacts; making a change to one artifact may trigger a slew of related modifications to other artifacts. Furthermore, there is the risk of deploying software changes that introduce new errors, or regression errors, that are worse than the bug(s) fixed by the change. In this case, additional costs may include the cost to fix the regression error and costs associated with lost revenue and or future sales due to lost user confidence. Added to all of these costs is the fact that estimating the effort and cost to make a change can be extremely difficult.

To better illustrate, consider the tasks involved with fixing a bug in deployed software. The cost may be significant in terms of developer time and effort to:

1. Determine whether the bug is both an actual bug and distinct from other known bugs,
2. Assess the bug in the context of the error’s severity,
3. Reproduce the bug in order to observe the failure,
4. Determine the source of the problem (which is usually not the location of the failure),
5. Design a solution,
6. Review the designed solution,
7. Implement the solution,
8. Review the implemented solution,
9. Write new test cases and/or correct old ones,
10. Document the solution and test cases,
11. Update documentation,
Similar costs accrue in other situations, e.g., when a new feature is added to the application.

Other costs include lost opportunities (e.g., other tasks that could be accomplished instead of fixing the bug), machine time, and company reputation. Future maintenance costs are another possible cost. For example, if the current design of the software does not lend itself to modification, it could make future changes more difficult and therefore costly, and these costs could actually outweigh the immediate cost of a redesign. Furthermore, the cost of the change may not be realized until long after the changed code is deployed. Consider a change to an on-line shopping application in a feature that is not often used, but eventually prevents a certain user from completing her purchase. In this case, there is likely lost revenue. There may also be a significant cost in fixing the bug if the developer of the code is no longer available to fix it (and even if she is, she may no longer be familiar with the code).

The sheer number of activities and costs surrounding any software change should be sufficient to dispel any notion that software is “soft”. However, we can take things even a step further and note that there are costs associated with there being so many costs! For instance, an additional cost results from the fact that it can be very difficult to estimate the amount of time for each of the activities when there are so many unknowns. We often do not know how long it will take to locate the source of a bug, we may also have trouble estimating the amount of effort to fix a bug, etc. As a result, activities such as scheduling and planning may be impacted by some unknown amount.

To better understand why software changes over time, consider the different drivers of change, i.e., the motivations behind different software changes. Some changes are beyond the developer’s control. For instance, some changes are due to the evolution of requirements or to updates to how those requirements are documented and communicated (e.g., requirements changes and specification changes). In other cases, software changes are initiated by the developer. Software development, like other creative activities, e.g., writing a book, involves a certain amount of trial-and-error, both with regards to individual components and to the way those components are organized into a whole. So, regardless of the formal software development process followed or the software artifact that is being developed, software engineers often leverage what they learn by evolving and changing their solution (e.g., design changes and bug fixes).

- **Requirements Changes:** The requirements imposed on the software can change for many reasons. New features might be needed, or old features might become obsolete. Additional clients may start using the software, or prior clients may stop using it. Requirements might also change as clients’ or developers’ understanding of the problem shifts, e.g., as a result of experimenting with models prior to release or after seeing the software in action after deployment. Furthermore, not all requirements come from the customer. The environment in which the software is deployed affects its operation and can in fact be a driver for change. For example, most applications rely on services or libraries provided by the underlying operating system (OS); changes to the software may be necessary when old services are discontinued and new services are introduced. Software evolution in response to a changing needs and changing environments is often referred to as adaptive maintenance.

- **Specification Changes:** The problems solved by modern software systems are exceedingly complicated; it is generally not possible for one person to keep all of a system’s software requirements in mind at once, which means that these requirements must be documented as a specification. Sometimes a specification will change in meaning even when the requirements do not, in order to correct errors, remove inconsistencies, reduce ambiguity, etc. These changes, usually a type of corrective maintenance, can in turn entail corresponding modifications to the code, especially in cases where prior misunderstandings or miscommunication of the requirements led to their incorrect implementation.

- **Design Changes:** Design changes are often initiated to better bridge specifications, i.e., what is desired, with what is delivered. These changes may affect large portions of the code and involve considerable risk and initial cost, but at the same time they may have considerable payoff in terms of future
maintainability and performance. Such changes are referred to as perfective maintenance or, in cases where they stave off potential future problems, preventative maintenance. Oftentimes, the goal of such maintenance is to improve the code without affecting actual functionality, i.e., to refactor the code.

- **Bug Fixes:** One of the most common motivations for changing software after it is deployed is to fix bugs resulting from an incorrect implementation of the requirements. That is, the implementation does not match the stated requirements. Bug fixes are another type of corrective maintenance.

The ability to evolve software over time is important—it enables us to use the software for long periods of time while still making improvements to it. With the support of feature rich version control systems, along with software tracking, e.g., bug tracking, systems and iterative software development practices, continual change can be effectively managed, and even viewed as an asset that enables us to make discoveries and improvements that facilitate continuous improvement of the software.

**Most software is designed, built and maintained by teams of people.** Team members may or may not be co-located and may or may not have the same work hours. Teams often have members with diverse backgrounds, skills, and interests. Some team members may have a technical background, similar to your own. Other team members may represent the client, or user, and may have limited or no software development knowledge. Furthermore, teams change over time. New developers join the team and existing team members move between projects or assume new responsibilities. The team-based nature of software development creates non-technical challenges that can greatly impact the success of the project. Software engineering processes and tools support both technical and non-technical tasks, helping teams coordinate their work, communicate with each other, and control and track the many changes by many individuals necessary to build and maintain successful software systems.

### 1.5 Essential Tools for Engineering Software

In this section we explore the types of software engineering tools you will soon use to organize and build various software artifacts. In general, each type of tool can be further organized based on its interface. The first type of interface is a Command-Line Interface (CLI). The second type is a Graphical User Interface (GUI). Users interact with a tool via the CLI by typing commands in a terminal or console window. Tools that provide a GUI interface use graphics, including windows, menus, buttons, scrollbars, etc. to enable users to interact with the tool via the keyboard, mouse, touchscreen, etc.

#### 1.5.1 Organizing and Managing Software

*Software Configuration Management (SCM)* is an important part of software engineering. It refers to the management, i.e., tracking and controlling, of changes to the software. The underlying operating system on your computer provides a mechanism to organize software artifacts (and other documents). Most operating systems support a hierarchical file system that organizes files into folders, and creates hierarchies of nested folders. Each file contains the current version of the artifact or document. For example, if you create a spreadsheet on your computer, save it and exit the spreadsheet application, the filesystem contains the saved version of your spreadsheet. If you open the spreadsheet, make changes, and save and exit the application, the filesystem again contains the current version of your spreadsheet. The previous version no longer exists (unless you saved it to a different location or with a different file name).

An important component of software configuration management is version control. While maintaining a single version of each file saves disk space, it also makes it difficult to revert to a previous version of the file. Periodic and on-change backup systems are commonly used to mitigate this problem; however, these systems generally lack the more advanced features necessary to support software development by teams of developers. For instance, if two people are working on the same file and their backups are stored on the same
drive, changes made by one developer may overwrite changes by the other developer. And, depending on
the number of backup versions stored, changes may be lost over time. If the developers’ backups are stored
on separate drives, then coordinating the changes from different backups becomes problematic. Similarly,
filesystems do not include support for simultaneous edits to files. At best, the filesystem will warn the user
that a file is already open for edits when it is opened for editing a second time, but it does not provide
support for merging changes made simultaneously by different users.

Software engineers typically use a Version Control System (VCS) to manage changes to software artifacts.
These systems typically provide support for merging changes to the same file and enable previous versions to
be restored. They also provide support for versions of the artifact to evolve in parallel. Traditional version
control systems followed a centralized model where a central software repository stores a copy of the files in a
location that is accessible by all users (e.g., developers). Each user performs a checkout of the files, copying
the files onto his or her computer. When the user has a change ready to share, a commit is performed to save
the change on the server. Other users can then see and checkout the changed code to their own machines.

In recent years, Distributed Version Control Systems (DVCSs) have been developed to enable software
development without the need for a centralized repository. Instead of relying on a shared repository to store
the project files, each user clones a copy of the project repository, including all of the history, to her own
machine. While many projects continue to rely on a centralized repository to store the “official” version
of the files, this practice is not required by a DVCS. When a user is ready to save changes, a commit is
performed to save the changes to the local repository. When the user is ready to share his changes, the
files are then pushed to a shared repository. The shared repository may be the centralized repository or
another repository (e.g., another developer’s repository). Similarly when a user wants to integrate changes
from other contributors, the changes are pulled from another repository into the user’s local repository.

Version control tools provide an interface to the version control system which enables users to maintain
multiple versions of a file, share files with other users, and identify file differences across versions.

1.5.2 Reading and Writing Software Artifacts

At its most fundamental level, most source code is simply a sequence of characters. As such, any text
editor can be used to read and write code. In fact, software developers used general text editors for many
years as their primary tool for reading and writing source code. One of the primary downsides of using a
generic text editor to write code is that it does not provide programming-language support features such
as auto-completion or highlighting of programming constructs. It also does not support navigation between
files based on the semantics, or meanings, of program elements. Sophisticated text editors such as word
processing software often attempt to “correct” source code to fit natural-language grammar, formatting, and
punctuation which may not be consistent with the programming language conventions.

Given the limitations of general text editors, many software developers choose to use an Integrated Devel-
opment Environment (IDE). These tools compensate for the limitations of general text editors mentioned
above, plus they are specifically designed to support software development by providing a single, integrated
environment to eliminate the need to set up and integrate individual tools (e.g., version control, build tools,
debuggers, etc.). Many IDEs support multiple programming languages, and most modern IDEs provide a
GUI.

Integrated development environment tools enable software developers to not only read and write code, but to
perform many other tasks involving software.

1.5.3 Building Software

Although source code can be auto-generated, it is often written by humans, and as such is written to be
read and updated by humans. Human-readable code, however, is not efficient for processing by an electronic
device. In order to deploy code so that it can execute, or run, on a computer, it first needs to be built. The process of building, or compiling, code includes a sequence of steps that essentially translate the source code into a binary format that is executable. The build process also includes steps that analyze the program to ensure its correctness. As software systems have grown in size and complexity, build tools have grown in scope and sophistication, providing support for identifying and managing project dependencies, and building, testing, and deploying software systems.

Build tools enable software engineers to automatically translate source code to binary format. These tools also perform a variety of checks on the code and may deploy the software to the target environment.

### 1.5.4 Running Software

Software applications begin executing when we issue a command to the operating system telling it to launch the program. For example, to run an application on your phone or laptop, you tap or click on the application’s icon. If the application is already running, and you click on its icon, the focus of the operating system will shift to the application. Similarly, to run a program with a CLI, you type a command that instructs the system to launch the application.

The underlying operating system is the ‘tool’ that enables software applications to run. We issue commands to the operating system to run software programs.

### 1.5.5 Analyzing Software

The last type of tool we discuss in this section are the tools used to analyze software programs. These tools enable software developers to check the correctness of the software, its performance, and other characteristics that can help determine if the application meets the specified requirements. Some analysis tools are run every time the application is built, for example, tools that determine if the source code follows the syntactic rules specified by the programming language, or those that try to identify regression errors. Other analysis tools are run on-demand (e.g., when a new feature is added).

Analysis tools enable software developers to evaluate software for properties such as correctness, consistency and efficiency, to determine if the software meets its specified requirements.

### 1.6 Essential Skills & Knowledge for Engineering Software

Software engineers are problem solvers. They apply computational theories, models and techniques to the development of software in order to solve problems specified by their requirements. These activities involve problem formulation and analysis, solution design, and activities related to the implementation of the solution (i.e., developing code to organize, store and process the data necessary to solve the problem). Software engineers also solve engineering problems related to how the software is built, analyzed, stored, documented, and maintained. As software systems have grown in size and complexity, these problems have become increasingly challenging. For instance, software engineering resources at organizations such as Google, Microsoft, and Facebook are specifically dedicated to finding ways to efficiently build, analyze and test the large bodies of evolving software that make up their products. To be an effective problem solver and software engineer requires a core set of technical skills and knowledge that include the ability to:

- Design a solution, given a computational problem.
- Read and understand code written by others.
- Know and apply basic algorithms and data structures.
- Analyze and evaluate alternative designs and solutions.
- Write correct and maintainable code.
- Design, write and use test cases.
- Use version control systems to store and share software artifacts.
- Learn and apply new models, technologies and techniques as they emerge in a rapidly evolving field.

Software engineers also need to have strong non-technical skills, such as the ability to work as part of a team and the ability to work with, and, in particular, communicate with non-software engineers—application developers, database analysts, and other technical specialists, as well as business managers and analysts, and clients who may have little understanding of how software is developed. For these, and other reasons, software engineers must also possess strong non-technical skills. These skills include the ability to:

- Work independently within a team.
- Communicate verbally and in writing with audiences with varying backgrounds and levels of technical expertise.
- Operate under tight deadlines
- Adapt to changes in the environment, software product, team composition, processes, etc.
- Ask for help and offer help to others.
- Persevere despite setbacks and failures.

While the lists included in this section are relatively short, and they contain somewhat generic skills, you should at least have an initial sense for what it takes to be a successful software engineer. It is interesting to note that this topic has recently become an area of research in the software engineering community. A relevant (and easy-to-read) paper that provides the insights of 59 experienced software engineers at Microsoft on this topic can be found in the paper, “What Makes a Great Software Engineer” by Paul Luo Li, Andrew J. Ko and Jiamin Zhu (available online).
Chapter 2

Computational Thinking in Software Engineering

Problem solving is a part of our daily lives from the time we wake up in the morning until we go to bed at night. It relies on many complex mental processes we develop from an early age as well as the ability to think creatively. We, as humans, are so accustomed to solving problems that we can often formulate solutions with seemingly little mental effort. Although many of the problem-solving techniques you currently use will be helpful, the nature of software development and the size and scale of software engineering problems requires a methodical approach and special techniques to efficiently and effectively design and engineer software solutions. In this chapter, we explore computational thinking—a core set of activities and techniques used by software engineers to solve problems. At the end of this chapter, you should be able to:

1. Describe the four main computational thinking activities.

2. Describe the key techniques applied in computational thinking.

3. Explain the concept of an algorithm.

4. Explain the tradeoffs between natural language, pseudocode, and source code for expressing an algorithm.

2.1 Introduction to Computational Thinking

Computational thinking is the thought processes involved in formulating a problem and expressing its solution(s) in such a way that a computer—human or machine—can effectively carry the solution out.¹ This approach to problem solving has been around for many years and is not limited to software engineering or computer science. As the pervasiveness of computing has grown, so has interest in computational thinking. In fact, other disciplines such as economics, law, and the arts are all beginning to teach and apply computational thinking.

Examples of computational thinking include:

- Breaking a large problem into smaller problems we can understand and manage.
- Analyzing a solution design for its correctness and future extensibility.
- Composing existing solutions to solve a larger problem.
- Expressing the solution to a problem as an algorithm that can be carried out by a computer.

There are two main components to computational thinking: 1) a set of core activities and 2) a collection of powerful techniques that enable us to handle large and complex software engineering problems.

It is important to note that the core activities in computational thinking go beyond problem solving:

- *Problem formulation* is probably the most important activity. During problem formulation, the problem solver is focused on understanding the problem deeply enough that a quality solution can be designed and implemented. By “quality solution”, we mean that the solution will need to not only solve the problem, but to do so in a way that also meets quality requirements such as extensibility, maintainability, testability, etc. Though often tedious and time consuming (and sometimes given little attention in our rush to solve the problem), problem formulation is a critical prerequisite to successful solution design and implementation.

- *Solution design and expression* constructs a solution based on the results from problem formulation. It can be viewed as an iterative process of designing and refining that ultimately leads to an algorithm, a finite process that the computer should follow, and an implementation, an expression of that process in enough detail that it can be applied to the problem.

- *Solution application* is the step where the problem solver uses the implementation to solve the problem.

- *Solution analysis* evaluates the solution to determine whether it actually solves the stated problem.

These activities do not necessarily occur in sequence; often a discovery made in one activity will cause the problem solver to revisit another, earlier activity.

In addition to these activities, computational thinking incorporates several key techniques to help manage the challenges associated with solving large, complicated problems:

- *pattern identification*: looking for and recognizing similarities and trends,
- *abstraction*: simplifying a problem by ignoring the details and focusing only on certain, relevant parts in order to manage complexity,
- *refinement*: reintroducing details into an abstraction when that abstraction is not specific enough (the dual of abstraction),
- *decomposition*: breaking a problem into smaller, more manageable parts, and
- *composition*: combining existing solutions to implement a desired abstraction (the dual of decomposition).

In the remainder of this chapter, we take a closer look at the individual activities and techniques that make up computational thinking and how they fit together.
2.2 Computational Thinking Activities

The four main activities in computational thinking form a set of core activities that are repeated as many times as necessary to solve the problem. Although a natural ordering is present (e.g., we cannot apply a solution without first designing it), the order of the activities may vary depending on, for example, the outcome of the previous activity. A good mental model of the computational thinking process is that after every activity the problem solver is (hopefully) left with a partial solution that is a step closer to the final solution than what they began the activity with.

2.2.1 Problem Formulation

The first principle in problem solving is to understand the problem. In computational thinking, the problem formulation activity is focused on developing an understanding of the stated problem. It is often not possible to understand the problem fully, however, we need to understand it well enough that we can begin to design a solution and the solution we design can be performed by a computer—machine or human. In software engineering, problem formulation centers on the specified requirements for the system. Challenges to successful problem formulation are missing or misunderstood requirements, missing assumptions, and unrecognized or unstated biases. Computational thinking techniques that help us overcome these challenges and manage large and complex problems include pattern identification, abstraction, and decomposition. The key output from this activity is a restatement of the problem in a format that enables solution design.

2.2.2 Solution Design and Expression

In the solution design and expression activity, focus is shifted from the problem to the solution. The outputs of this activity are the specification of one or more algorithms. An algorithm is a sequence of steps that when followed will address some particular instance of the problem. It can be helpful to think of an algorithm as being similar to a recipe in that an algorithm is a general plan that can be reused across problem instance without reinventing a solution each time, just as a recipe can be followed each time we want to make a certain dish, even if the particulars of how we follow that recipe might vary. Much as a recipe usually only makes on dish, each algorithm typically solves only one problem, and, like recipes, algorithms can be specified in a variety of ways and using different notations.

Some of the notations we use to specify algorithms are easy for humans to read and understand, while others are easy for electronic devices to process. At first, it may seem that natural-language specifications would be easiest for humans to read and understand, but in practice, natural language specifications tend to be ambiguous, meaning that it is possible (and even likely!) for there to be many valid interpretations of what is written. Furthermore, natural language specifications can be verbose and therefore harder to decipher than specifications in a more concise format.

Like natural language, pseudocode is a high-level description of an algorithm that is easy for humans to read, but it is more structured, and it avoids many of the ambiguities of natural language specifications. It mimics actual programming languages without tying the specification of the algorithm to a particular programming language, so it is often used in documentation. Pseudocode is not intended to be run by a machine and, unlike a programming language, there are no formatting or syntax rules; however, there tend to be certain conventions that are generally followed. Think of pseudocode as middle ground for specifying algorithms that falls between programming languages and natural language. Writing an algorithm in pseudocode prior to writing it in a programming language allows us to identify patterns, make abstractions, and refine the design without the need to learn or know a programming language.

Sometimes it is necessary to specify more details in an algorithm, including details that are only necessary for solution application. In this case, solution expression may be done using code. Algorithms specified
using a high-level programming language, such as Java, Python or C, are more structured and succinct with respect to natural language and, like natural language specifications and pseudocode, they are also human readable. Code, however, has the advantage of being (automatically) translatable into a low-level machine language that can run on an electronic computer.

Consider the simple problem of deciding whether “X” has won a game of tic-tac-toe. A natural language description of an algorithm to do this might look like this:

If there are three “X”s in a row on the board, declare that “X” has won.

This natural language description is quite easy to read, but it is also full of ambiguity. Does “in a row” mean along a horizontal or just in a straight line? If in a straight line, can that line be diagonal? What happens if there are not three Xs in a row; does that mean that “X” has lost? Someone unfamiliar with tic-tac-toe might read the specification rather differently than intended.

We can use more words to reduce the ambiguity:

If, anywhere within the board, there are three distinct, contiguous (either orthogonally or diagonally), collinear locations, all of which are occupied by an “X”, declare that “X” has won. Otherwise, take no action.

but now readability suffers. And, as you can imagine, if the algorithm had been more complicated, we would have needed even more verbiage to close every ambiguity, which not only makes the algorithm harder to read, but also increases the chances of some error slipping by unnoticed. Furthermore, translating this algorithm into a low-level language, suitable for execution on a computer, phone, etc., requires manual effort because computers cannot currently cope with the many different ways that algorithms can be specified in natural language.

If we instead use pseudocode, we can write a clear, concise description of the algorithm as shown below:

let $R \leftarrow$ the board’s rows, columns, and diagonals
for each $r$ in $R$
  if the number of “X”s in $r$ is 3 then
    declare that X wins
  end
end

This would still be difficult to make executable, but notice how it does use certain common wordings like “for each” and “if” as well as indentation to convey meaning in ways that other readers are unlikely to misunderstand. Note also how it separates parts of the process onto their own lines so that they can more easily be read and reasoned about and so that omissions or other errors are easier to spot.

Another advantage is that the pseudocode looks very much like the code that would be written to implement the algorithm in a high-level programming language. Below is one implementation in Python:

```python
rows = {(0, 1, 2), (3, 4, 5), (6, 7, 8), (0, 3, 6), (1, 4, 7), (2, 5, 8), (0, 4, 8), (2, 4, 6)}
for row in rows:
  if [board[position] for position in row].count('X') == 3:
    return Outcomes.X_VICTORY
```
The only major differences are in cases where the program must specify details unimportant to the algorithm but needed by the computer. For example, on the first line, the code identifies each board location by a number from zero to eight, whereas in the natural language and pseudocode we did not worry about how locations were represented.

### 2.2.3 Solution Application

Solution application is the computational thinking activity that involves running or executing the solution on one or more problem instances. We defer a more detailed discussion of solution application to a future chapter, but note that although solution application can be automated, it is also a non-trivial activity to set up and includes a wide range of deployment and support activities.

### 2.2.4 Solution Analysis

Solution analysis, like problem formulation is a critical, but sometimes rushed activity. The objective of solution analysis is to determine if the solution actually solves the stated problem. Solution analysis involves software testing, code reviews, and other analysis techniques that we will discuss in future chapters. Note that this activity should not be viewed as a final step in the computational thinking process, but should be practiced during and between the other activities.

### 2.3 Techniques

In the previous section we identified four computational thinking activities that describe “what” we do when solving computational problems. In this section, we describe techniques for “how” we can solve the large, complex problems we encounter in software engineering.

#### 2.3.1 Pattern Identification

Pattern identification is a technique that is used to identify similarities and trends within a problem or across problems. We can look for sameness and trends in date. We can also look for similarities in processes an operations.

#### 2.3.2 Abstraction

Abstraction is a mechanism for dealing with complexity where, instead of considering something in full detail, we substitute a mental model (also called an abstraction) that is easier to grasp and reason about. Abstraction typically occurs at many levels, with higher levels of abstraction ignoring more and more details.

For example, if someone says that they are “curled up in a comfy chair with a good book to read”, it is enough to understand that a chair can be sat in and can be comfortable—we do not need to know whether the chair has arms, its color, how many wheels its has, whether it reclines, whether it rocks, etc. Thus, the term “chair” is an abstraction of the highly detailed concept that is the actual chair. The term “furniture” is a higher abstraction; when we talk about “furniture”, we do not even specify whether it can be sat in.

Similarly, we use abstractions to generalize software concepts, usually specified as either data abstractions or procedural abstractions, which abstract data or processes, respectively. For instance, an app that is created to record information about hospital patients may create a data abstraction referred to as a ‘Patient’ so that its documentation, code, and tests can refer to the idea of a patient instead of to “a first name, an associated
last name, an associated date of birth, etc.” It might also have a procedural abstraction referred to as a “log in” to substitute for the actual process of reading and verifying a username and password.

As a design technique, abstraction is an iterative process of identifying levels of abstraction that will be useful, deciding what mental models should be used at each level, and identifying the data or code that can be grouped together under those mental models. While abstraction is a powerful tool, it can also be difficult to choose the right abstraction to solve the problem.

### 2.3.3 Refinement

Refinement is the process of introducing details to an abstraction when that abstraction alone is not specific enough. That is, while abstraction is the framework that allows us to think about software at different levels of detail, refinement is the process of “zooming in” and providing more detail exactly in those areas relevant to a particular task.

For example, when ordering furniture for a classroom, it is not enough to write “furniture for a classroom” in the order form. We must specify what sorts of furniture and how many of each kind we want.

Likewise, in software engineering, we often start with high-level designs and refine them until they can be implemented. For instance, a team might start with a requirement that the app they will develop should be easy to set up, which is quite abstract. In the next round of design, the specification might be refined to say that the app will provide a setup wizard with each configuration option on a separate page. The next round might refine that design further to determine the order the options will appear in, and eventually it would be specific enough that the setup wizard could be coded.

As a design technique, refinement is an iterative process of identifying in which abstractions detail is needed, deciding which details should be specified immediately and which should be deferred, and determining which choices for those details best support the requirements.

### 2.3.4 Decomposition

Decomposition is the process of breaking a problem into smaller, more manageable parts. Decomposing a problem into smaller problems has the advantage of enabling multiple people to work on the problem, sometimes in parallel. It also has the potential to create reusable solutions or use an existing solution. Although not all problems can be decomposed into smaller problems, the inability to decompose a problem may signal a poorly understood problem that requires more information or more analysis of the problem. It also may signal a problem that is already simple enough and does not require further decomposition.

### 2.3.5 Composition

Composition is the process of combining existing, fully refined components in a way that implements a desired abstraction.

For instance, when a classroom is being designed for a lecture, the school must decide how the furniture available for purchase can be used to support the concept of “lecture-style seating”. And later, when the classroom needs to support teamwork instead of a lecture, the school probably does not order entirely new furniture just to realize the abstraction “teamwork seating”. Instead, the existing furniture might be composed differently, perhaps by arranging it into clusters.

The same is true in software. For instance, a software engineer designing the user interface for a medical records software system might refine the design to have pages for choosing a patient, for entering observations such as weight, height, blood pressure and temperature, and for confirming entries. Then, once these designs...
are completed, compose them to get the design for a “vitals” module. Later, a lab-entry module might not bother to refine its design and instead just compose the existing components in a slightly different way.

As a design technique, composition is an iterative process of selecting components to support an abstraction, finding ways to combine those components while still adhering to the intended abstraction, and identifying any gaps that remain to be filled by refinement. A key benefit of composition is often solution reuse (i.e., using solutions that were created previously).
Chapter 3

Software Quality

“High-quality software” is something all software engineers aspire to build. While the idea of “high-quality” software seems intuitive, its actual meaning has long been debated and a precise definition is actually quite difficult to come by. Is software high-quality if it crashes only once each year? Is software high-quality if it contains fewer than 10 bugs? Or, if it has 10 bugs, does that mean its quality is low (or somewhere in between)? Is the definition of high-quality different for the software controlling the brakes on your car versus a weather app on your phone? Some say that it is not possible to precisely define quality in the context of software, and others adopt a minimalist perspective and equate “high-quality” with the absence of bugs. No matter which philosophy we align with, quality is an important issue for software consumers. Most software projects cannot afford to take a “we will know it when we see it” approach, so in this chapter we explore the topic of software quality by looking at the many aspects of software that can affect its quality. We also discuss various metrics and techniques that have been developed to measure software quality, and finally we explore where software quality processes fit into the software development lifecycle.

After reading this chapter, you should be able to:

1. Explain why defining high-quality software is not straightforward.
2. Describe two software metrics.
3. Explain why software quality includes—but is not limited to—software doing the right thing.
4. Explain the role of software quality processes in software development.

3.1 The Relationship between Software Quality and Requirements

Let’s begin with a simple definition of software quality that one might find in a software engineering book:

Software quality is the degree to which software conforms to its requirements.

This may seem like a reasonable definition, however, it is not immediately practical. Recall from the previous chapter that it is difficult (if not impossible) to accurately specify all of the requirements for a real-world software system, in part because all of the requirements are rarely known upfront. Without a complete set of specifications, we cannot therefore use this definition to measure quality. Furthermore, even if we were to have a system and a perfect set of requirements for it, it is not clear how to summarize any differences between them as a “degree of conformance”. Is conformance all-or-nothing, or is there such a thing as
partially conforming? Can software still be high-quality without conforming to every requirement, if it conforms to the ‘critical’ or ‘most important’ requirements? And what role does context play? For example, are there differences in how we measure quality, or what level of quality is acceptable between flight-control software and a mobile app?

The usual solution to these questions is to challenge the premise: instead of insisting on one overarching definition, we consider the many dimensions of software quality and discuss how each might be defined, expressed in requirements, and measured. Although the various dimensions of software quality may have different levels of importance to different projects, taken together, they provide a common lexicon for describing software quality that can be applied to all software.

3.2 Dimensions of Software Quality

When we talk about software quality, one of our most basic concerns is about what the software does and does not do (i.e., its functionality or behavior). For instance, does the software compute the correct value, or display a list in the proper order? Requirements about behavior are called functional requirements, and software that conforms to these requirements does so in two ways: it exhibits the required behaviors, and at the same time it does not exhibit other, extra behaviors. Consider a functional requirement like this:

If the user submits the form with a valid name, credit card number, and security code, the order should be relayed to the sales server.

A system that does not send orders would be considered defective because of missing functionality, while a system that sends the orders even when they included invalid data would also be considered defective because of extra, unwanted functionality.

Other requirements are nonfunctional requirements, which are generally concerned not with what the software does, but how it does it. For instance:

Orders will reach the sales server within five minutes of being placed.

Under this requirement, a system that cannot transmit orders within this timeframe would be considered defective, as would be a system that regularly experiences network outages.

Typically, nonfunctional requirements are further subdivided according to more specific quality concerns. These subdivisions give us the following nonfunctional dimensions of software quality:

- **Maintainability (source)**: the ease with which the software can be changed.
  - **Code complexity**: how complicated the implementation is (low complexity is generally—but not always—better).
    * **Code size**: the amount of code in the software (less code is generally—but not always—better).
    * **Structural complexity**: the extent to which data and control structures in the language affect the execution flow of the program.
  - **Readability**: the extent to which another developer can understand the code.
  - **Modularity**: the extent to which the software can accurately be thought of as separate pieces solving separate problems.
    * **Cohesion**: the extent to which pieces of the software that are grouped together are actually related.
* **Coupling:** the extent to which pieces of the software that are not grouped together are related (low coupling is generally—but not always—better).

  - **Testability:** the ease with which the software can be tested.
  - **Testedness:** the extent to which the software’s behaviors are ones that have been verified by testing.

* **Scalability:** the ability of the software to remain functional when its problem size increases.

  - **Performance/Efficiency:** the amount accomplished by the software relative to the resources it consumes.

    * **Time efficiency:** the amount accomplished by the software relative to the time it consumes.
    * **Space efficiency:** the amount accomplished by the software relative to the memory it consumes.
    * **Energy efficiency:** the amount accomplished by the software relative to the energy it consumes.

* **Compatibility:** the ease with which the software can communicate with, or match, the behavior of other software.

* **Portability:** the ease with which the software can be made to function in a variety of environments.

* **Usability:** the ease with which human users can learn and interact with the software.

  - **Accessibility:** the absence of variance in the ease with which human users can learn and interact with the software.
  - **Communicativeness:** the amount of information made available to human users (an intermediate amount of communicativeness is generally—but not always—better).
  - **Comprehensibility:** the ease with which human users can understand the information the software makes available to them.

And in addition to all of these aspects of quality, there are also dimensions that straddle the line between functional and nonfunctional:

* **Dependability:** the extent to which the software can justifiably be trusted.

  - **Availability:** the amount of its time that the software, once deployed, spends in a usable state.

    * **Reliability:** the likelihood that the software will behave correctly during any one particular use.
    * **Maintainability (deployments):** the ease with which the software can be restored to a usable state after it behaves incorrectly.
  - **Safety:** the absence of dangers posed by the software.
  - **Security:** the extent to which the software resists malicious use.

    * **Confidentiality:** the absence of opportunities for malicious collection of data.
    * **Integrity:** the absence of opportunities for malicious modifications to data.

As you can see, when someone tells you they are delivering high-quality software, “high-quality” may have a multitude of meanings, and it is ultimately the software’s particular requirements that determine which dimensions come into play and their relative importance.
3.3 Software Quality Metrics

While the dimensions we identified are useful for articulating what quality means in a particular context, it is only through analysis and measurements that it is possible to answer questions such as “Which of these pieces of software is higher quality?” or “Where should our time and effort be spent to best improve this software?” Hence, we now turn our attention to the metrics that enable us to measure software along those dimensions.

As with any measure, the value of each of these metrics depends on how accurately it can be measured, how the measurement is interpreted, and at what time the findings factor into decision making. Furthermore, we will see that many metrics, e.g., the number of lines of source code, provide little information on their own, but become more meaningful when combined with other measures.

3.3.1 Metrics for Code

First, we consider software quality metrics regarding the functionality of code. Historically, these metrics have focused on a lack of quality (versus presence of quality) by measuring the frequency of functional defects, commonly called bugs. Lower values, e.g., fewer bugs per line of code, were treated as an indicator of higher quality software. But in practice we know that such metrics can be misleading when considered in isolation. For example, suppose we have a codebase with one million lines of code, and the developers tell you that it is high quality because there are no known bugs. Do you agree that since no bugs were found that it is high quality? Even if you are clever and think to ask how extensively the software has been tested, you still may not have a good picture of the quality of the code, as we will discuss below.

The most basic approach to measuring functional correctness is to directly compare the expected functionality, as specified in requirements, with the actual or observed behaviors; this process is referred to as software testing. In software testing, the tester prepares a collection of test cases, called a test suite. Each test case is a scenario paired with an test oracle, or set of expectations about the software’s behavior in that scenario. The program is then run through each scenario, which makes software testing a type of dynamic program analysis—the measurements are taken while the code is executing. A test case is said to pass if the software’s observed behavior matches the oracle and to fail otherwise. The failure count and the pass/fail rate then quantify the software’s functional quality.

Software testing is subject to a number of possible inaccuracies, which must be considered when interpreting test results. There is the risk that a tester incorrectly specifies the scenario, the oracle, or both for a test case, which can cause correct software to fail or—more insidiously—incorrect software to pass. There is the risk that different test cases exercise the same parts of the code, meaning that hundreds of failed test cases may only indicate a single bug. And there is a risk that no test case in the test suite covers a particular requirement or a particular piece of code, so that bugs may go undetected.

Software quality related to nonfunctional requirements is also challenging to measure, but equally important. For example, if an application includes exactly the features specified in the requirements, but personal data collected by the app are not stored in a secure location, the app is not likely to be judged as high quality. Since there are many, and wide-ranging, nonfunctional dimensions of quality, they can play an important role in helping choose between two apps when they both provide the same, or similar, functionality.

Like functional requirements, some nonfunctional requirements can be analyzed using quantitative metrics. For example, we can count the number of lines of code or the number of test cases using a static program analysis. Static analysis techniques are the dual of dynamic analysis techniques—they compute metrics without running the software. Quantitative metrics that assess nonfunctional requirements also include memory usage and response time. Metrics like these are computed using dynamic analyses in a variety of scenarios, i.e., both with small and large datasets. Additionally, for the latter set of metrics we can collect both average and maximum values to get a better picture of typical and worst-case scenarios. We can also
combine metrics collected statically and dynamically. For instance, an application’s memory footprint is the total amount of memory it consumes, both to hold its own code (which we can measure statically) and to hold its data (which we can measure dynamically).

The size of the code as a metric on its own provides little information regarding the quality of the code. Good code is not necessarily small or large, but whatever size it needs to be to be readable, maintainable, testable, etc. Still, the size of the code may provide just the little bit of information we need to help us make a decision. For example, if two apps implement the same functionality and one app contains twice as many lines of code as the other, certain questions are raised. The bigger app may be over-engineered or perhaps written by developers who were not very familiar with the programming language, while the smaller app’s code is clean and tight. Or, the smaller app may be written in a very terse, difficult-to-understand, difficult-to-maintain format while the larger app has been written with maintainability in mind. Either way, the discrepancy in size is a red flag that indicates it may be necessary to look more closely before making a decision.

Some dimensions of quality are not easily measured. For example, how does one objectively measure the readability or maintainability of code? In this case, we may need to rely on one or more individuals to estimate quality based on their educated opinions after they perform a code review. To determine maintainability, we may combine several metrics to help us infer the level of quality. For example, we may analyze the version history of the code to determine the number of bugs detected in a code module in the past year, and combine that information with the amount of code tested in that module. If the code is well tested but continues to be the source of many defects, then we may decide the quality of that code is low.

It is interesting to note that some nonfunctional requirements are important to software developers, while others are important to end users. For example, maintainability, scalability, and testability are important to software developers and their management, but do not directly impact end users. The majority of end users simply want the software to work and are not concerned with how it is built; their primary concerns are functionality and usability and, depending on context, safety, security, and dependability.

### 3.3.2 Metrics for Non-Code Artifacts

As you learned in Chapter 1, software is much more than code. It includes the many artifacts that are developed to support the development and maintenance of software systems. It is not difficult, then, to imagine that the quality of the system is impacted by not only the quality of the code, but also by the quality of the other artifacts. Furthermore, quality tends to be transitive—in other words, if the quality of an artifact, such as a requirements specification, is low, then the quality of artifacts that depend on it (e.g., the source code) may be low simply because they are based on a low-quality starting point.

Many non-code software artifacts will have similar quality dimensions as those presented in Section 3.2. For example, readability and testability are important dimension of quality in the context of requirements specifications. Many non-code artifacts will also have additional quality dimensions. For example, completeness and consistency are also important quality of requirements specifications.

Measuring the quality of non-code artifacts involves many of the same challenges as measuring the quality of code: some metrics are quantitative, while others are qualitative and rely on human judgement. In some instances it may be necessary to use a proxy metric to judge the quality of non-code artifacts. For example, code coverage metrics are widely used in practice to determine the quality of a test suite due to the lack of better alternatives. Note that care must be taken when interpreting these metrics. Simply running code during a test does not mean that its behavior was actually verified, let alone verified well. However, code coverage metrics can indicate which parts of the application’s code were not executed (i.e., tested) when the test suite was run.
3.4 Software Quality Processes

In Chapter 1 we learned that software engineering, like all engineering disciplines, involves the use of rigorous processes to design, develop, test and maintain the complex software systems built and maintained today. Software quality assurance processes codify the many lessons learned about building both successful and not-so-successful software systems. They describe preventative activities and processes specifically intended to build quality into the software and avoid defects. Similarly, conventions, i.e., agreements about how something is done, specify rules that guide the development and maintenance of software artifacts and are also intended to ensure quality is built into the product. Then, as various software artifacts are produced, software quality control processes are used to validate the artifacts’ compliance with their specified requirements and to detect any problems. While software quality processes may incur considerable project overhead, e.g., testing time may delay the deployment of a new feature, failure to establish or follow software quality processes can result in costly rework or repairs in the future.

Each software development team establishes processes that govern the development of the software artifacts. The formality and scope of quality processes can vary widely across projects, ranging from ad hoc processes to well-defined and repeatable processes. For example, most teams specify how changes are managed (i.e., version control processes), how defects (bugs) are reported and tracked, and how code changes are tested. Many teams also specify coding conventions that govern various aspects of how code is written, including how the code is formatted, the use of third party libraries, and which coding constructs are allowed or disallowed. Teams may also specify when products can proceed to the next stage, how project information is documented or communicated, and even what information must be documented.

Software quality control processes include the reviews, inspections, and walkthroughs used to ensure the quality of the artifacts. For example, a review of the requirements specification may be performed to ensure the requirements are consistent and complete, while a design review is typically focused on aspects such as the testability of the design, the assumptions made, and the degree to which the design satisfies the requirements. Code reviews typically focus on the correctness of the implementation and adherence to the requirements. Code reviews also may addresses the readability, testability, and maintainability of the code. Even artifacts such as the test strategy and test cases themselves may be subject to review to ensure the testing is adequate.
Chapter 4

Coding Fundamentals I: Expressions, Variables, and Assignment Statements

Writing code is a fundamental task in software engineering. Although it is similar to writing pseudocode, writing code to run on an electronic device requires specialized knowledge of computing constructs that enable the algorithm to be executed in an automated manner. In this chapter, we explore basic computing constructs for specifying algorithms in code (i.e., in a programming language). Although each language has its own syntax, or way of expressing these constructs, the fundamental idea of each construct is similar across programming languages.

At the end of this chapter, you should be able to:

1. Identify the components of an algorithm.

2. Draw a plumbing diagram to represent data dependencies.

3. Translate a process to an expression.

4. Explain the difference between a variable and a value.

5. Draw a series of state diagrams to represent context changes during program execution.

4.1 The Components of an Algorithm

Up to this point, you have used natural language and pseudocode to specify algorithms to solve a variety of simple problems. We often write algorithms in pseudocode to help us develop a deeper understanding of the problem and proposed solution and to communicate our solutions to others. Algorithms specified in pseudocode, however, use ambiguous terminology and leave out important details that are necessary for algorithm execution, such as how and where data (information) used in the algorithm are stored. Furthermore, the lack of a standard language for pseudocode hinders the process of translating the algorithm from a human-readable form into an executable form.
Before we transition to learning computing constructs that will help us address these issues, let’s first analyze an algorithm (recipe) for making a birthday cake:

1. Stir together cake mix and water. (Here we assume we don’t need eggs or oil.)
2. Bake the batter from step 1.
3. Frost the cake from step 2 with frosting.

At its most basic level, an algorithm is a high-level process, or sequence of steps, to be followed. We refer to the order in which the steps are performed as the execution order. When analyzing an algorithm (e.g., to locate a bug or to better understand the solution), it is sometimes helpful to abstract away some of the details of the algorithm and represent the algorithm in a way that clearly depicts the order in which the steps will be performed. Below is a Control Flow Graph (CFG) representation of the birthday cake algorithm. It shows the order in which the steps will be performed: Step 1 → Step 2 → Step 3. Although this algorithm has a very simple CFG, many algorithms will have much more complicated CFGs.

![Figure 4.1: CFG for the birthday cake algorithm.](image)

As you learned in the previous chapter, maintainability is an important dimension of software quality. As software engineers, we attempt to write algorithms in such a way that they are easy for others to read and maintain without additional information. In some instances, however, it may be necessary to include additional details that are not required for the algorithm to run, but that will help explain why the algorithm is implemented in a particular way (e.g., the assumption on Step 1). Information for human readers of the algorithm is specified in the form of comments or annotations. This information can be ignored by the computer as it does not affect the processing of the algorithm. Care should be taken when embedding comments within an algorithm to ensure the algorithm and comments remain consistent when one or the other is changed.

Now that we have observed the components of an algorithm at a high-level, let’s take a closer look at the algorithm steps. Notice how each step in the birthday cake algorithm specifies an operation (i.e., an action to take):

1. Stir together cake mix and water. (Here we assume we don’t need eggs or oil.)
2. Bake the batter from step 1.
3. Frost the cake from step 2 with frosting.
and zero or more *arguments* (data to use when performing the step):

1. Stir together cake mix and water. (Here we assume we don’t need eggs or oil.)
2. Bake the batter from step 1.
3. Frost the cake from step 2 with frosting.

*Arguments* can either specify data literally, as a *literal*:

1. Stir together cake mix and water. (Here we assume we don’t need eggs or oil.)
2. Bake the batter from step 1.
3. Frost the cake from step 2 with frosting.

or indirectly, as a *data reference*:

1. Stir together cake mix and water. (Here we assume we don’t need eggs or oil.)
2. Bake the batter from step 1.
3. Frost the cake from step 2 with frosting.

Data references may establish *data dependencies* between steps. For instance, the baking step above has a data dependence on the stirring step, and the frosting step has a data dependence on the baking step.

### 4.2 Plumbing Diagrams (a.k.a. Machine Diagrams)

A *plumbing diagram* (sometimes also called a *machine diagram*) represents data dependencies explicitly. For example, this is the same process for making a birthday cake presented as a plumbing diagram, where data dependencies are indicated by arrows:

```
<table>
<thead>
<tr>
<th>cake mix</th>
<th>water</th>
</tr>
</thead>
<tbody>
<tr>
<td>stir</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>bake</td>
<td>frosting</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>frost</td>
<td></td>
</tr>
</tbody>
</table>
```

The left-to-right order of arguments often matters in a plumbing diagram; covering the cake with frosting is different from covering the frosting with cake.
### 4.3 Java Expressions

From a plumbing diagram, we can translate a process to an *expression*.

To create a Java expression for cake making, we must first abide by one of Java’s rules, that names may contain letters, digits, underscores, or dollar signs, but not spaces:

- `cakeMix`
- `water`
- `stir`
- `bake`
- `frosting`
- `frost`

Java expressions can represent operations in two ways. One case is when operations are *functions*, which are written with their arguments separated by commas and in parentheses after the function name. If `stir` is a function, then the diagram above could be simplified to:

```
stir(cakeMix, water)
```

Similarly, if `bake` is a function:

```
bake(stir(cakeMix, water))
```

And finally, if `frost` is a function:

```
frost(bake(stir(cakeMix, water)), frosting)
```

So, written with functions, the expression is `frost(bake(stir(cakeMix, water)), frosting)`.

On the other hand, if the operations are *operators*, the arguments (called *operands*) are mingled with the operator symbol(s), and parentheses can be used to clarify grouping. Inventing some operators for mixing, baking, and frosting, we have (using dots for argument placeholders):
which simplifies to:

which simplifies to:

which simplifies to:

So, written with operators, the expression is \( !\text{(cakeMix + water)} \cdot \text{frosting} \).

It is also possible for functions and operators to be used together. For instance, the same recipe could be written \( \text{frost}(\text{bake}(\text{cakeMix + water}), \text{frosting}) \).

### 4.4 Data and Variables

Information in a computer is divided into pieces called *data* or *values*, where each datum or value is the answer to some question. For example, the information that I will have Dorothy Lynch dressing on my salad could be represented by the value “Dorothy Lynch” as an answer to the question “What dressing will I have on my salad?”

A *variable* is a named, dedicated location to store the answer to a particular question. For instance, to answer the question “What dressing will I have on my salad?”, a programmer might create a variable named *dressing* and store the value *dorothyLynch* in it.
Thus, variables are associated with questions, and values are associated with answers. Questions (and therefore the variables a program uses) tend to be the same across different situations, but answers to those questions (and therefore the values stored in the program’s variables) tend to change as the situation changes.

### 4.5 Types

Different kinds of questions are answered with different *types* of values. “Ranch”, for instance, if not the correct answer to the question above, is at least the right sort of answer. But the answer “Australia”, for example, would make no sense, because its type is “continent”, not “salad dressing”.

Types also affect what operations can be performed on data. While an expression like

\[
\text{topWith(topWith(salad, dorothyLynch), croutons)}
\]

might make sense, the expression

\[
\text{topWith(topWith(salad, australia), croutons)}
\]

would not.

### 4.6 Contexts and State Diagrams

A *context* associates variable names with values of the appropriate types. Contexts are drawn using *program state diagrams*, like the following:

```
  topping  croutons
  dressing dorothyLynch
```

This particular diagram says that the answer to the question “What topping will I have on my salad?” is “Croutons” and that the answer to the question “What dressing will I have on my salad?” is “Dorothy Lynch”.

When an expression is evaluated in a context, variable names that appear in the expression act as placeholders for the answers to the questions they represent. Effectively, the variable names are replaced by the corresponding values. For instance, given the context illustrated above, the expression

\[
\text{topWith(topWith(salad, dressing), topping)}
\]

is equivalent to

\[
\text{topWith(topWith(salad, dorothyLynch), croutons)}
\]
4.7 Variable Declarations

Variables are created and their names added to a context by a variable declaration, which in Java is written as the name of the type the variables will hold, followed by the names of the variables separated by commas, followed by a semicolon. Initially these variables will either be undefined or hold the closest equivalent to “nothing” that their type allows, usually the special Java value null, depending on where the declaration appears. For example, given the context

```
topping ----> croutons
```
```
dressing ----> dorothyLynch
```

the declaration

```
Food garnish, dinnerRoll;
```

would introduce two new variables that can hold foods and are initially undefined:

```
topping ----> croutons
dressing ----> dorothyLynch
garnish
```
```
dinnerRoll ----> [undefined]
```

4.8 Assignment Statements

An assignment statement is an instruction to change the contents of a variable. A Java assignment statement is written as the name of the variable, an equals sign (=, which, in this case, means “becomes”, not “equals”), an expression, and then a semicolon. For instance,

```
garnish = cucumberFlower;
```

The computer performs three steps when an assignment statement is run:

1. Compute the value of the right-hand side,
2. Disconnect the left-hand side from any previous value, and
3. Connect the left-hand side to the value of the right-hand side.
So, given the context

```
    topping  ➔ croutons
    dressing ➔ dorothyLynch
    garnish  ➔ cucumberFlower
    dinnerRoll ➔ [undefined]
```

the assignment statement

```
garnish = cucumberFlower;
```

produces first the evaluation of the right-hand side:

```
    topping  ➔ croutons
    dressing ➔ dorothyLynch
    garnish  ➔ cucumberFlower
    dinnerRoll ➔ [undefined]
```

then disconnects garnish:

```
    topping  ➔ croutons
    dressing ➔ dorothyLynch
    garnish  ➔ cucumberFlower
    dinner_roll ➔ [undefined]
```

and finally reconnects garnish to the value from the right-hand side:

```
    topping  ➔ croutons
    dressing ➔ dorothyLynch
    garnish  ➔ cucumberFlower
    dinnerRoll ➔ [undefined]
```
4.9 Expression Statements

Besides declarations and assignment statements, expressions can also be followed by a semicolon to form an *expression statement*. An expression statement ignores its result, but the functions inside of it might have side-effects. For instance, in the statement

```
taste(garnish);
```

`taste`’s result, if it even has one, is not available to further computations. But the side-effects, like our having tasted the cucumber flower, still occur.

4.10 Order of Execution

When a program has multiple statements, statements are run in order from top to bottom. Therefore, the following code tastes a cucumber flower:

```
garnish = cucumberFlower;
taste(garnish);
```

whereas this code tastes the current `garnish` and then replaces it with a cucumber flower:

```
taste(garnish);
garnish = cucumberFlower;
```

In Java, assignment statements can also be combined into declarations:

```
Food garnish = cucumberFlower, dinnerRoll;
```

is equivalent to

```
Food garnish;
garnish = cucumberFlower;
Food dinnerRoll;
```

4.11 Common Uses of Assignment Statements

There are three common uses of assignment statements:

1. To produce a side-effect,
2. To simplify one or more later statements, or
3. To preserve a value that will be disconnected by another assignment statement.
### 4.11.1 Assignment Statements for Side-Effects

If the changed variable is one that affects another part of the program, setting it can alter that other part’s behavior. For example,

```java
logFileVerbosity = 0;
```

might make the log-file-keeping code stop logging information.

### 4.11.2 Assignment Statements for Simplification

Instead of a complicated assignment like

```java
birthdayCake = frost(bake(cakeMix + water), frosting);
```

one could write

```java
Food plainCake = bake(cakeMix + water);
birthdayCake = frost(plainCake, frosting);
```

to make each line easier to read at the expense of using more lines.

Instead of

```java
normalizedDx = (xb - xa) / Math.sqrt((xb - xa) * (xb - xa) + (yb - ya) * (yb - ya));
```

one could write

```java
double dx = xb - xa;
double dy = yb - ya;
normalizedDx = dx / Math.sqrt(dx * dx + dy * dy);
```

(where `double` is the type for double-precision numbers) to save on bug-prone repetition.

### 4.11.3 Assignment Statements for Value Preservation

Finally, if one assignment statement is going to disconnect a value that needs to be preserved, another assignment statement can be used to retain it.
oldUsername = username;
username = oldUsername + "2";
changeMessage = "Your username has changed from '" +
         oldUsername + "' to '" + username + "'.";
Glossary

A | B | C | D | E | F | G | H | I | L | M | N | P | R | S | T | V | W
A

abstraction
A process for identifying general principles, rules or concepts by filtering information or selecting only the relevant aspects. The term can also refer to the outcome of the abstraction process. 18, 19

adaptive maintenance
Software changes performed in response to environmental changes or changing needs. 11

agile software development
An approach to software development in which the requirements and the software solution evolve based on continuous feedback from the project stakeholders. 9

algorithm
A finite sequence of operations that can be performed to solve a specific problem. 18, 19

API
Application Programming Interface. 6

app
See software application. 5

application software
A family of programs that perform a specific task beyond the basic operation of the computer itself. 6

argument
The data value passed to an operation, function, or program. 33

artifact
A tangible work product created during software development. 6, 12

B

build script
The sequence of instructions followed to compile or translate software into another (typically executable) format. 10

C
CFG
Control Flow Graph. 32

CLI
Command-Line Interface. 12, 14

code coverage metric
A measure used to describe the degree to which a test suite executes the source code under test. 29

composition
The process of combining existing, fully refined components in a way that implements a desired abstraction. 18

corrective maintenance
Software changes performed to rectify errors and inconsistencies detected in the software. 11, 12

D
decomposition
The process breaking up a problem or system into smaller, manageable parts. 18, 19

DVCS
Distributed Version Control System. 13

dynamic program analysis
Software analysis techniques that compute metrics about the software while executing it. 28

E
embedded system
A computer system that solves a specific function that is located within a larger mechanical or electrical system. 6, 8

evolutionary software development
An approach to software development that delivers software iteratively, each time with additional or improved functionality. 9

expression
A construct that is made up of variables, literals, operators, and function calls that evaluates to a single value. 34

extensibility
The extent to which the software can be extended or added to in order to include new features or functionality. 18

F
file system
The part of the operating system on a computer that controls how files are organized and stored. 12

functional requirement
A type of requirement that specifies what the software should do. 26
GUI

Graphical User Interface. 12, 13

hardware

The collection of physical components, such as the monitor, keyboard, hard drives, CPU, etc. that compose a computer system. 6

IDE

Integrated Development Environment. 13

literal

A fixed value. 33

maintainability

The ease with which the software can be changed. 18

nonfunctional requirement

A type of requirement that specifies how the software operates, rather than a function it performs. 26

pattern identification

the process of looking for and recognizing similarities and trends. 18

perfective maintenance

Software changes to improve performance, maintainability and other aspects of how the software operates or how it is maintained. 12

preventative maintenance

Software changes intended to avoid future problems in how the software operates or in how it is maintained. 12

program

See computer program. 5

programing language

A formal language that specifies the set of unambiguous instructions that can be used to process and store data. 19, 31
proxy metric
A metric that measures a value that close to what is desired, typically used when it is not possible or feasible to measure the actual value desired. 29

pseudocode
An informal, but structured, high-level description of an algorithm. 19

R
refactor
Code changes that restructure the code but do not affect its actual functionality. 12

refinement
The process of introducing details into an abstraction when the abstraction alone is not specific enough. 18

regression error
An error that is introduced when a change is made to the code. 14

requirements specification
A description of the software system to be developed. 29

S
SCM
Software Configuration Management. 12

software bug
A defect in the functionality of the software; a violation in a functional requirement. 8

software development methodology
An approach for developing software that includes techniques, processes, roles and deliverables. Also known as software development life cycle. 8

software library
A collection of code modules that solve a particular problem and are developed with reuse in mind. 6

software quality assurance
Software engineering activities and processes specifically intended to build quality into the software and avoid defects. 30

software quality control
Software engineering activities and processes that are used to check if the software meets its specified requirements. 30

software repository
A structured repository for storing and archiving software artifacts. 13

software testing
An assessment of the software to determine its level of quality and conformance to requirements. 28
source code
A collection of computer instructions written in a human-readable programming language. 6

specification
A description of the software that is used to design, build and analyze its behavior and qualities. 11

static program analysis
Software analysis techniques that compute metrics about the software without actually executing it. 28

system software
Software that operates the computer hardware and is designed to provide services to software applications. 6

t

test case
The inputs, sequence of actions, and expected results used to check if the software meets its specified requirements. 28

test oracle
A mechanism for determine whether a test case has passed or failed. A test oracle describes the expected behaviors or output for the given inputs to the software under test. 28

test strategy
A description of how the software will be tested, often including the types of testing to be performed, the tools that will be used, environmental dependencies, and a timeline for when the testing will be performed. 30

test suite
A finite collection of test cases. 28

testability
The ease with which the software can be tested. 18

V

variable
A named storage location for storing a value. 35

VCS
Version Control System. 13

W

waterfall model
A sequential software development process where the core set of software development activities is treated as a sequence of phases. 9