Web version:
https://softengbook.org

How to cite:
Marco Tulio Valente. Software Engineering: A Modern Approach, 2024.

Bibtex:
https://softengbook.org/bibtex

Software Engineering:

A Modern Approach

Marco Tulio Valente

Chapter 1

Introduction

> *Our civilization runs on software.* – Bjarne Stroustrup

In this first chapter, we outline the objectives and challenges addressed by software engineering (Section 1.1). We also present and explore the primary topics studied in this field of computer science (Section 1.2). Our goal is to provide a comprehensive understanding of the field before exploring specific topics. As software engineering is a broad domain, we discuss the types of software systems that can benefit from the principles and practices presented in this book (Section 1.3). This discussion aims to avoid misconceptions concerning the scope of our work. Finally, we present an overview of the topics covered in the remaining chapters of the book (Section 1.4).

1.1 Definition and Historical Context

In today's world, software powers virtually everything. Organizations of all types and sizes, including businesses and governmental entities, depend on software systems to deliver their services effectively. Governments often interact with citizens through software applications. Many businesses sell a wide range of products directly to consumers via e-commerce platforms. Software is also embedded in physical products, including cars, airplanes, satellites, and robots. Furthermore, software is transforming traditional industries such as telecommunications, transportation in large urban centers, and advertising.

Given the vital role software plays in our society, it's not surprising that a field of computer science has emerged to develop systematic approaches for constructing

software systems, particularly large and complex projects. This field is known as **software engineering**.

Software engineering focuses on systematic, disciplined, and quantifiable approaches to developing, operating, maintaining, and evolving software systems. As mentioned, this field is central to computer science and involves applying engineering principles to software construction.

Historically, software engineering as a field emerged in the late 1960s when the first generation of computers began to be used for problem-solving. At that time, software took a back seat, as the primary focus was on building machines capable of solving a limited set of scientific problems.

However, advancements in hardware technologies changed this scenario. By the late 1960s, computers had become more common, appearing in many universities, and businesses began to envision the benefits of their adoption. Consequently, a new set of challenges surfaced, as users demanded more complex and diverse applications, including commercial systems for tasks such as payroll, accounting, and inventory management.

NATO Conference: In October 1968, around 50 eminent computer scientists gathered for a week in Garmisch, Germany, for a NATO-sponsored conference. The purpose of this meeting was to discuss a "crucial problem of computer usage, the so-called software." The conference produced a 130-page report advocating for software development based on practical and theoretical principles, drawing inspiration from other branches of engineering. This historic event is now considered a landmark in the establishment of software engineering as a distinct discipline within computer science.

Figure 1.1: Scientists at the 1968 NATO conference on software engineering. Reproduction kindly authorized by Prof. Robert McClure.

One of the participants in the NATO Conference summarized the challenges faced by the new research field:

> *The basic problem is that certain classes of systems are placing demands on us which are beyond our capabilities and our theories and methods of design and production at this time. There are many areas where there is no such thing as a crisis—sort routines, payroll applications, for example. It is large systems that are encountering great difficulties. We should not expect the production of such systems to be easy*.

More than half a century after the NATO Conference, techniques and methods for software construction have improved remarkably. It's now widely recognized that software, in most instances, should not be built in sequential phases, as happens with traditional engineering products. Many design patterns can assist software engineers in their work, avoiding the need to reinvent the wheel for every new design problem. Libraries and frameworks for various purposes are available, allowing developers to reuse code without mastering the details of tasks such as implementing graphical interfaces, manipulating data structures, accessing databases, and encrypting messages. Several automated testing techniques can—and should—be deployed to ensure that the produced code works as expected when used by actual customers. Like other engineering products, software also deteriorates over time and therefore requires maintenance, not only for bug fixing but also to ensure its long-term maintainability and readability.

No Silver Bullet

Software development has unique characteristics when compared to other engineering products, especially hardware. Frederick Brooks, a recipient of the 1999 Turing Award and a pioneer in Software Engineering, was among the first to underscore this fact. In 1987, he wrote an essay titled *No Silver Bullet: Essence and Accident in Software Engineering* (link[1]), in which he highlighted the peculiarities of software engineering.

According to Brooks, there are two types of difficulties in software engineering: **essential difficulties** and **accidental difficulties**. Essential ones are intrinsic to the field and are unlikely to be solved by novel technologies.

Indeed, the term "silver bullet", as used in Brooks's essay title, refers to a magical solution to a complex problem. Brooks argued that due to the essential difficulties,

[1] https://doi.org/10.1109/MC.1987.1663532

we should not anticipate the existence of silver bullets in the field. Nonetheless, novel software technologies are often marketed as if they were, in fact, silver bullets.

Brooks described the following essential difficulties:

1. Complexity: Of all human-made constructions, software stands out as one of the most intricate and complex. Even traditional engineering constructs, such as satellites, nuclear power plants, or rockets, are increasingly relying on software.

2. Conformity: Software must adapt continuously to its ever-changing environment. For instance, changes in tax legislation require prompt adaptations in related software. This level of adaptability is not required in other fields like Physics, where natural laws do not change due to human decisions.

3. Changeability: There is constant pressure for software to evolve and incorporate new features. In fact, the more successful a software product is, the more likely it is to attract requests for modifications.

4. Invisibility: It is inherently challenging to visualize the structure and complexity of software due to its abstract nature.

Challenges (2), (3), and (4) are largely exclusive to software systems; they do not exist in other engineering products, at least not with the same intensity. When environmental laws change, car manufacturers have years to comply with the new legislation. Also, cars, once manufactured, do not usually receive new functionalities. Furthermore, a car's physical properties, such as weight, height, width, number of seats, and shape, facilitate evaluation by consumers.

Software development also faces accidental difficulties, but these can be addressed by developers given adequate training and access to existing technologies and resources. Examples of such difficulties include an IDE that frequently crashes, a compiler with cryptic error messages, a framework lacking documentation, or a web application with a confusing interface.

Real World: The complexity involved in building software systems can be illustrated by considering their size. For instance, the open-source projects supported by the Linux Foundation have over 1.7 billion lines of code and involve 65,000 active developers, according to the foundation's 2023 report. Specifically, the Linux kernel, which is the core component of the operating

system, has more than 30 million lines of code. Companies such as Google have also reported that their systems comprise billions of lines of code (link[a]).

[a] https://doi.org/10.1145/2854146

1.2 Topics of Study

To present the topics of study in software engineering, we refer to the *Guide to the Software Engineering Body of Knowledge*, also known as the SWEBOK (link[2]). This comprehensive report is organized by the IEEE Computer Society, an internationally recognized scientific society. It is crafted with the expertise of multiple researchers and industry professionals. The aim of the report is to document and compile the body of knowledge that is representative of the field we now recognize as software engineering.

The SWEBOK defines the following key areas of Software Engineering:

1. Software Requirements
2. Software Architecture
3. Software Design
4. Software Construction
5. Software Testing
6. Software Engineering Operations
7. Software Maintenance
8. Software Configuration Management
9. Software Engineering Management
10. Software Engineering Processes
11. Software Engineering Models and Methods
12. Software Quality
13. Software Security
14. Software Engineering Professional Practice
15. Software Engineering Economics

The report also outlines three additional knowledge areas: Computing Foundations, Mathematical Foundations, and Engineering Foundations. However, given that they overlap with other scientific domains, they won't be featured in this book.

In the remainder of this section, we will summarize each of the twelve key areas identified above. Our objective is to give an overview of the knowledge accumulated

[2] http://www.swebok.org

over the years in software engineering and, consequently, to shed light on what is studied in this field.

1.2.1 Requirements

Requirements define both *what* a system should do and *how* it should behave. Accordingly, Requirements Engineering designates the activities carried out to elicit, analyze, document, and validate a system's requirements.

Requirements are classified as either **functional** or **non-functional**. Functional requirements define the features or services a system should provide. Non-functional requirements, on the other hand, outline *how* a system should operate, including constraints and the expected quality of service. Examples of non-functional requirements include, but are not limited to, performance, availability, fault tolerance, security, privacy, interoperability, scalability, maintainability, and usability.

Let's consider a banking application as an example. The functional requirements of this application may include account balance display, statement generation, transfers between accounts, and debit card cancellation, among others. The non-functional requirements might be as follows:

- Performance: The application must be able to provide an account balance within two seconds.
- Availability: The application must be online 99% of the time.
- Fault tolerance: The application must continue functioning even if a specific data center fails.
- Security: The application must encrypt all data exchanged with branches.
- Privacy: Customer data must be maintained as confidential and not be leaked to third parties.
- Interoperability: The application must integrate with Central Bank systems.
- Scalability: The application should be able to handle data for one million banking customers.
- Usability: The application must be accessible to visually impaired individuals.

1.2.2 Design

Software design involves determining the principal code units of a software system. However, this process only extends to the level of interfaces, which include **provided interfaces** and **required interfaces**. Provided interfaces are the services that

a code unit makes public for other parts of the system, while required interfaces are those that a code unit depends on for operation.

Consequently, during software design, the implementation details of each code unit, such as the specifics of method implementations, are not addressed. For instance, when designing a banking system, a class to represent bank accounts might be defined as follows:

```
class BankAccount {
  private Customer customer;
  private double balance;
  public double getBalance() { ... }
  public String getCustomerName() { ... }
  public String getStatement(Date start, Date end) { ... }
  ...
}
```

It's important to note that this is a simplified implementation intended for illustrative purposes. The `BankAccount` class provides an interface to other classes through its public methods, thus constituting its provided interface. However, `BankAccount` also relies on the `Customer` class, making the `Customer` interface a required one for `BankAccount`. In this case, we say that `BankAccount` depends on `Customer`.

When design becomes more abstract and involves larger units like packages or folders, it is called architectural design. Essentially, **software architecture** refers to the organization of a system at a higher level of abstraction than that involving classes or comparable code units.

1.2.3 Construction

Construction refers to the implementation phase, also known as programming the system. This involves making numerous choices, such as deciding on the algorithms and data structures to use, installing and configuring third-party frameworks and libraries, defining exception handling policies, reaching a consensus on coding standards (including naming conventions, indentation rules, and code documentation practices), and also selecting the development tools, such as compilers, integrated development environments (IDEs), debuggers, version control systems, databases, interface builders, and AI-based code completion tools.

1.2.4 Testing

Testing involves executing a program with a finite set of cases and verifying whether it delivers the expected results. As the 1982 Turing Award recipient Edsger W. Dijkstra succinctly put it:

> *Testing shows the presence, not the absence of bugs.*

There are at least three relevant points we would like to address about testing in this first chapter. First, we should mention that there are many types of tests, including **unit tests** (testing small code units like a class), **integration tests** (testing larger units like a set of classes), **performance tests** (checking system performance under specific loads), and **usability tests** (evaluating the usability of the system's user interface).

Second, testing serves both verification and validation purposes. Verification ensures a system conforms to its specifications, while validation checks if the system fulfills the customer's needs. The two concepts are distinct because specifications may at times fail to meet customers' needs. This might be due to misunderstandings between developers and users or to poor explanations by users.

Two phrases commonly used to distinguish verification and validation are:

- Verification: Are we building the product right? That is, according to the specification we received.
- Validation: Are we building the right product? That is, the one that meets the customer or market needs.

For instance, running a method to check whether it returns the specified result is a verification activity, while conducting an acceptance meeting to show the system to customers is a validation activity.

Third, it's also important to define three testing-related terms: **defects**, **bugs**, and **failures**. For this purpose, let's consider the following code that computes the area of a circle based on a certain condition:

```
if (condition)
    area = pi * radius * radius * radius;
```

This code has a defect, as the area of a circle is "pi times radius squared," not cubed. The term *bug* is used informally with the same meaning.

A failure occurs when the defective code is executed—for instance, when the `if` condition above is true, and as a result, the program delivers an incorrect result. Consequently, not every defect or bug results in a failure, since the defective code might never be executed.

In summary, defective (or buggy) code does not conform to its specification. If this code is executed and yields incorrect results, we say that a failure has occurred.

In-Depth: The literature on testing sometimes also mentions the **error** and **fault** terms. These terms carry the same meaning we attributed to defects. For instance, the IEEE Standard Glossary of Software Engineering Terminology (link[a], page 32) defines a fault as an "incorrect step, process, or data definition in a computer program; the terms error and bug are [also] used to express this meaning." In essence, defect, error, fault, and bug are synonymous.

[a] https://doi.org/10.1109/IEEESTD.1990.101064

Real World: There are many software failures that have had serious financial and human consequences. A prominent example is the 1996 explosion of the French rocket Ariane 5 shortly after its launch from Kourou, in French Guiana. About 30 seconds after the launch, the rocket exploded due to an unexpected behavior of one of its onboard systems. This resulted in a financial loss of approximately half a billion dollars. Curiously, the defect that caused the failure was confined to a few lines of an Ada function (which is a programming language widely used in military and space software). The defective lines were responsible for converting a 64-bit floating-point value to a 16-bit integer. During testing and likely during previous Ariane launches, the conversion always succeeded—–the real number always fit into an integer. However, on the fatal launch day, a previously untested condition required the conversion of a larger real number than the largest 16-bit integer could accommodate. This generated an erroneous result that caused the rocket's control system to malfunction and subsequently resulted in its explosion.

1.2.5 Maintenance and Evolution

Software systems, like traditional engineering systems, require maintenance. In this book, we categorize the types of maintenance that can be performed on software

into the following categories: **corrective**, **preventive**, **adaptive**, **refactoring**, and **evolutionary**.

Corrective maintenance aims to address bugs reported by users or other developers. Preventive maintenance, on the other hand, focuses on addressing latent bugs that haven't yet caused failures observed by users.

> **Real World:** An example of preventive maintenance was the action taken by many organizations before the turn of the millennium, from 1999 to 2000. At that time, a significant number of applications used two digits to represent the year in date values, i.e., dates were in the DD-MM-YY format. This raised concerns that date operations in 2000 and beyond could produce incorrect results. For instance, the calculation 00 - 99 might return an unexpected result. To mitigate this, organizations formed special task groups to convert all date variables and expressions in their systems to the DD-MM-YYYY format, which was therefore an example of preventive maintenance.

Adaptive maintenance aims to adjust a system in response to changes in its environment, including technological changes, new legislative rules, integration requirements with other systems, or customization demands from new customers. Examples include:

- Updating a system from Python 2.7 to Python 3.0.
- Customizing a system to meet the requirements of a new customer.
- Modifying a system to comply with changes in legislation.

Refactoring involves changes in a program that don't change its external behavior but improve its design and ease of maintenance. Refactoring operations include renaming a method or variable, breaking a large method into smaller ones, or moving a method to a more suitable class.

Evolutionary maintenance is performed to add new features to a system or to significantly enhance existing ones. Its purpose is to preserve the system's value for customers. For instance, many banking systems developed in the 1970s and 1980s have been continually updated and improved, ensuring their ongoing relevance and value.

Legacy systems are older systems built using outdated languages, operating systems, and databases. Despite being obsolete in technological terms, most legacy systems remain vital due to the critical operations they perform.

In-Depth: Some alternative classifications for software maintenance can be found in the literature. One, proposed by Lientz & Swanson in 1978 (link[a]), organizes maintenance activities into four categories: corrective (fixing bugs), perfective (adding new functionalities), adaptive (changes in the software's operational environment), and preventive (changes aimed at enhancing maintainability).

[a] https://dl.acm.org/citation.cfm?id=601062

1.2.6 Configuration Management

Version control systems, such as Git, are an integral part of contemporary software development. These systems store all versions of a software project, including source code, documentation, manuals, web pages, and reports. Thus, they enable the restoration of a specific previous version if a change introduces a bug.

Configuration management, however, encompasses more than just using a system like Git. It also includes the definition of policies to handle system versions. For example, a team might decide on an $x.y.z$ format to identify the versions of a library they're working on, where x, y, and z are integers. A change in x indicates a major version launch with substantial new features, changes in y denote a minor version with small updates, while changes in z point to a patch release with only bug fixes. This scheme is often referred to as **semantic versioning**.

1.2.7 Project Management

Project management plays a major role in software development. It involves activities ranging from negotiating contracts with customers, human resource management (hiring, training, and setting promotion policies and remuneration values), to risk management, monitoring competition, finance, and marketing. In this context, **stakeholders** is a term that refers to all parties with a vested interest in the project. This includes individuals or organizations that affect or are affected by the project, such as developers, project managers, contracted companies, suppliers, and in some cases, government entities.

Brooks' Law, formulated by Frederick Brooks, is a well-known adage in software project management:

Adding manpower to a late software project makes it later.

The rationale is that new developers need time to understand the codebase, architecture, and design before becoming effective. Moreover, larger teams require more communication and coordination to facilitate decision-making. For instance, a team with three developers (d_1, d_2, d_3) has three communication channels (d_1-d_2, d_1-d_3, and d_2-d_3). If the team expands to four members, the number of channels doubles to six. With ten developers, the number of communication channels increases to 45. For this reason, software is typically developed in small teams consisting of no more than a dozen engineers.

In-Depth: Brooks' Law is derived from a classic software project management book, *The Mythical Man-Month*. Its first edition was published in 1975 (link[a]), with Brooks documenting the lessons learned from his time as an IBM project manager. The 20th-anniversary edition includes a new chapter featuring the article *No Silver Bullet—Essence and Accidents of Software Engineering*, originally published in 1987. In 1999, Frederick Brooks received the Turing Award, the highest honor in computer science, which is equivalent to a Nobel Prize.

[a] https://dl.acm.org/citation.cfm?id=207583

1.2.8 Process

A software process defines the structured sequence of activities and events required to build, test, and deliver software. Indeed, software development can be compared to the construction of buildings, which need to follow a particular sequence of activities: foundation, masonry, roofing, plumbing installations, electrical installations, and painting, among others.

There are two main types of processes that can be used in the construction of software:

- Waterfall processes
- Agile processes

Originating in the 1970s as software engineering began to gain recognition, Waterfall processes were the first to be proposed. Taking inspiration from traditional engineering processes, they are centered on sequential activities, similar to the order of activities in the building construction analogy mentioned earlier. The use of Waterfall was prevalent until the 1990s, largely due to a standardization issued by the US Department of Defense in 1985. During this period, all software contracted by the Department of Defense had to be implemented using Waterfall.

Also known as a **plan-driven process**, Waterfall proposes a construction sequence where each stage flows sequentially, like a waterfall. As illustrated in the next figure, these stages are requirement specification, analysis, design, implementation, and testing. After these stages, the system is released for production.

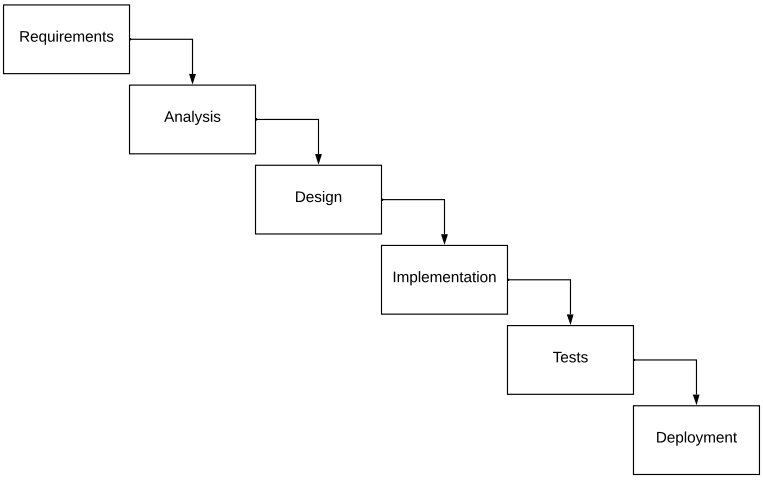

Figure 1.2: Phases of a Waterfall process

However, Waterfall faced severe criticism in the late 1990s due to frequent project delays and cost overruns. These issues usually happened because Waterfall requires a full requirements gathering phase, a complete design phase, followed by implementation and testing phases before deploying the system to users. Therefore, users may have to wait years to see a piece of software running. By then, the world may have changed, as well as the needs of the users, who may no longer need the system specified years before.

To address these challenges, a group of 17 software engineers devised an alternative approach, called Agile, at a meeting in Utah, United States, in February 2001. They also published a manifesto detailing the new approach, which they called the Agile Manifesto (link[3]). Contrary to the Waterfall approach, Agile recommends building software incrementally and iteratively, with validation by users at every iteration.

The concepts behind Agile have significantly impacted the software industry and are used today across organizations of many sizes. Various methods have been

[3] https://agilemanifesto.org

derived from these principles, such as **XP**, **Scrum**, and **Kanban**. Agile methods have also promoted the adoption of various programming practices, such as automated testing, test-driven development (i.e., writing the tests before the actual code), and continuous integration. This last practice recommends that developers push the code they produce to the main repository continuously. Ideally, every day, for example. The goal is to avoid integration conflicts, which occur when two or more developers change the same lines of code in parallel.

1.2.9 Models

Software models provide a higher-level representation of a system than its source code, enabling developers to analyze a system's properties and characteristics without immersing themselves in the details of the source code. These models can be created before the code, supporting Forward Engineering, or they can be created for understanding an existing code base, facilitating Reverse Engineering.

Typically, software models use graphical notations—for instance, **UML** (Unified Modeling Language), a notation featuring several diagrams to model structural and behavioral properties of a software project. The figure below shows a UML diagram—called a class diagram—for the example of code used in Section 1.2.2. In this diagram, the rectangular boxes represent classes, including their attributes and methods. Arrows denote relationships between classes. There are editors for creating UML diagrams, which can be used, for example, in forward engineering scenarios.

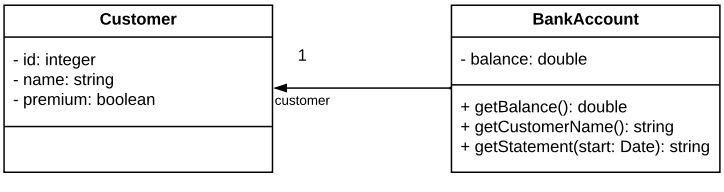

Figure 1.3: Example of UML Class Diagram

1.2.10 Quality

Quality is a fundamental goal in the engineering of products, reflected across various industries, such as automobile manufacturing, cell phone production, computer manufacturing, and construction. Similarly, quality is of utmost importance in software engineering. A classification proposed by Bertrand Meyer (link[4]) suggests

[4]https://dl.acm.org/citation.cfm?id=261119

that software quality can be evaluated in two dimensions: **external quality** and **internal quality**.

External quality refers to factors that can be evaluated without examining the source code. Thus, they can be evaluated by end-users who are not necessarily experts in software engineering. Some examples of external quality factors (or attributes) are:

- Correctness: Does the software align with its specification and perform as expected under normal conditions?
- Robustness: Can the software continue to function appropriately during exceptional circumstances, such as communication or disk failures? A robust software system should not crash due to such events; instead, it should alert users about the abnormal operation.
- Efficiency: Is the software making optimal use of the available computational resources?
- Portability: Is the software adaptable to other platforms and operating systems? Is it available for major operating systems? Does it run on mobile devices?
- Ease of Use: Does the software have a user-friendly interface, clear error messages, and support for multiple languages? Can users with disabilities, such as visual or auditory impairments, use it?
- Compatibility: Does the software support the most common data formats in its domain? For instance, a spreadsheet should import files in XLS and CSV formats.

In contrast, internal quality relates to properties associated with the system's implementation. Thus, assessment of internal quality requires expertise in software engineering and isn't something for end-users. Examples of internal quality factors include modularity, code readability, maintainability, and testability.

The assurance of software quality can be achieved via several strategies. First, **metrics** can be used to track the development process, including source code and process metrics. Code metric examples include the number of lines in a program, which provides an indication of its size. Process metrics include, for example, the number of bugs reported by end-users over a specific period.

There are also practices that promote the production of high-quality software. Notably, many organizations implement **code reviews**, where the code written

by one developer only moves to production after another team member reviews and approves it. This practice aids in early bug detection (before the system enters production) and contributes to improving the quality of the code (i.e., its maintainability, readability, modularity, etc.). It also encourages the dissemination of good software engineering practices within the team.

The next figure shows an example of a code review, referring to an example we used in Section 1.2.4. As we can see, this code was reviewed by another developer before being put into production. In this case, the reviewer noticed a bug and reported it in a comment. After that, the developer responsible for the code should fix the bug and resubmit the new code for review and approval. There are several tools to support code review practices. In the example, we used the tool provided by GitHub.

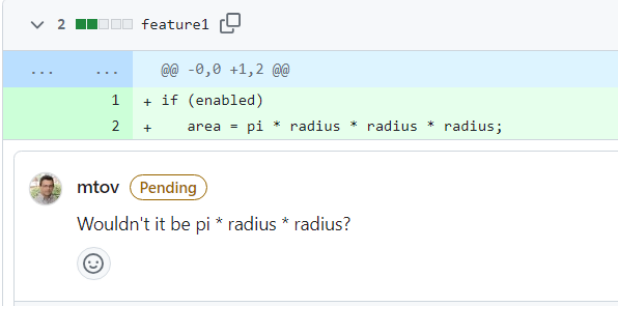

Figure 1.4: Example of code review

1.2.11 Professional Practice

The opening quote of this chapter by Bjarne Stroustrup states that *our civilization runs on software*. For this reason, there are myriad of opportunities for software professionals. However, the prevalence of software today also implies challenges and responsibilities. For instance, questions and challenges surrounding the professional practice of software engineering emerge when formulating curricula for undergraduate courses in this field. First, these courses should offer a solid foundation in fundamental aspects of computer science, such as algorithms and data structures. Additionally, they should also cover concepts, methodologies, and techniques that are widely used by professional software developers.

Equally important is the discussion about the **ethical responsibility** of software engineers, especially in a society where human interactions are increasingly mediated by software and algorithms. Scientific societies within the field have proposed codes to guide computing professionals in ethically exercising their profession. For

instance, the ACM's Code of Ethics (link[5]) and the IEEE Computer Society's Code of Ethics (link[6]) are examples of such guidelines. The latter puts special emphasis on the practice of software engineering, asserting the following:

> *Software engineers shall commit themselves to making the analysis, specification, design, development, testing, and maintenance of software a beneficial and respected profession.*

Real World: Stack Overflow holds an annual survey, which in 2018 received responses from over 100,000 developers worldwide. One section of the survey focused on ethical considerations (link[a]). One question asked if developers felt obligated to consider the ethical implications of the code they create—almost 80% responded positively. Another question asked who holds the responsibility for code that leads to unethical behavior, to which 57% pointed to the company's top management and 23% pointed to the developers themselves. When asked if they would write code for purposes they considered unethical, 58% said no, while 37% responded that it would depend on the code's requirements.

[a] https://insights.stackoverflow.com/survey/2018#ethics

1.2.12 Economics

Numerous economic factors are interlaced with software development. For instance, software startups must select their revenue model—perhaps a subscription-based or advertising-supported model. Mobile app developers must decide how much to charge for their apps, a decision that requires understanding their competitors' pricing, among other variables. With such complex and relevant economic considerations, it's no surprise that large software companies today hire economists to evaluate these factors related to their products.

As a practical example, economists frequently discuss the opportunity costs of a decision. These costs represent the missed opportunities associated with discarded solutions or choices. If you choose X over Y, the benefits of Y become your missed opportunities. For instance, imagine your company's main product has a list of bugs. While fixing these bugs will satisfy customers and potentially prevent churn, this

[5] https://www.acm.org/code-of-ethics
[6] https://www.computer.org/education/code-of-ethics

decision also has an opportunity cost. Instead of fixing the bugs, the company could invest in new features, which might contribute to attracting new customers. Thus, deciding between bug fixes and new features ultimately becomes an economic decision.

1.3 Classification of Software Systems

Software is part of a wide array of human activities. It comes in different sizes and types, supporting a variety of functional and non-functional requirements. As such, it's important to avoid the misconception that there is a single, universal method for software development. In other words, we should not assume that all software must follow the same processes, design principles, architectural patterns, or quality assurance practices.

Bertrand Meyer proposed a classification (link[7]) that assists in distinguishing between the types of software that can be developed and in identifying the most recommended software engineering practices for each type. According to Meyer, there are three primary types of software: Acute Systems, Business Systems, and Casual Systems. We discuss Casual and Acute systems first, followed by Business systems.

Casual systems, or Type C systems, are not under pressure for high-quality performance. They can tolerate minor bugs without jeopardizing their operation. As examples, we can mention an academic project script, a data migration program for one-time use, or a system to manage a student association's membership. These systems do not require high internal quality standards, optimal runtime performance, or sophisticated user interfaces. They are typically implemented by a single developer and are non-critical and lightweight. As a result, they do not benefit from the practices, techniques, and processes discussed in this book. In fact, **over-engineering** is a risk for such systems, as there's no need for advanced techniques.

On the other end of the spectrum, we have Acute systems, or Type A systems, where a single failure can have devastating consequences, including the loss of human lives. Notable examples include control systems used in autonomous vehicles, nuclear power plants, airplanes, ICU equipment, and subway trains. The software that controlled the Ariane 5 rocket is another example (see Section 1.2.4). Developing these systems requires rigorous processes, including comprehensive code review and external certification. It's common to have hardware and software redundancies—for example, two systems running in parallel that only make a

[7] https://bertrandmeyer.com/2013/03/25/the-abc-of-software-engineering/

decision when both agree. Sometimes, Type A systems are designed using formal languages based on logic or set theory.

> **Note:** In this book, we will not cover Type A (Acute) systems.

Lastly, we have Business Systems, or Type B systems. These systems benefit most from the principles and practices discussed in this book. They cover a wide array of enterprise applications (like finance, human resources, logistics, and sales), various web-based systems, software libraries and frameworks, general-purpose applications (such as text editors and spreadsheets), and basic software systems (like compilers and IDEs). The practices presented in this book were proposed to make the development of Type B systems more productive and to contribute to their quality, both internally (for example, resulting in systems that are easier to maintain) and externally (producing systems with fewer bugs, for example).

1.4 Upcoming Chapters

This book has ten chapters and one appendix:

Chapter 2: Processes focuses on agile development processes, specifically XP, Scrum, and Kanban. We chose to focus on agile methods due to their widespread adoption in modern software development. However, we also briefly cover traditional processes like Waterfall and the Unified Process.

Chapter 3: Requirements begins with a discussion of the importance and the main types of requirements. Then, we introduce two techniques for requirements elicitation and specification: user stories for agile methods, and use cases for more traditional and documentation-driven methods. The chapter also covers novel topics like Minimum Viable Products (MVP) and A/B Tests, whose importance today extends beyond startups.

Chapter 4: Models focuses on the use of UML to create sketches during software development projects. Indeed, UML is no longer widely used for creating detailed software models, as was its initial goal. However, we included UML in the book because developers frequently use it to create sketches for discussing or documenting design ideas.

Chapter 5: Design Principles covers two topics that every software engineer needs to know. They are as follows: (1) important properties (or considerations) in software design, including conceptual integrity, information hiding, cohesion,

and coupling; and (2) design principles, which are specific recommendations for building software projects, such as the widely discussed SOLID principles.

Chapter 6: Design Patterns summarizes the main design patterns defined in the literature. Essentially, design patterns are solutions to common problems faced by developers when designing software systems. The discussion of each pattern is divided into three parts: (1) a context, that is, a system in which the pattern can be useful; (2) a problem faced when designing this system; and (3) a solution to this problem using design patterns. We also provide several code examples to facilitate comprehension.

Chapter 7: Architecture begins with a discussion of the importance of this topic. After that, we present five architectural patterns, including: layered architectures, MVC (Model-View-Controller), microservices, message queues, and publish-subscribe. To conclude, we present an architectural anti-pattern, called Big Ball of Mud, which designates systems with no architectural organization at all. These systems might have had some architecture in the past, but it was progressively abandoned, turning them into a spaghetti of inter-module dependencies.

Chapter 8: Testing emphasizes unit tests, which are usually implemented using frameworks like JUnit. The chapter includes dozens of unit test examples and discusses various aspects of these tests. For instance, we discuss recommended principles for writing unit tests. We also present some test smells, which are patterns of tests that are not recommended. Then, we address testability, i.e., the importance of designing and writing code that can be easily tested. The chapter also includes a section on mocks and stubs, which are objects that enable unit testing of code with complex dependencies, such as dependencies on databases and other external services. After the discussion on unit tests, we cover two other types of tests: integration tests and end-to-end tests. These tests verify the properties of larger code units, like the classes that implement a given service (integration tests) or even all the classes in a system (end-to-end tests). To conclude, we include a brief discussion of other tests, such as black-box tests (or functional tests), white-box tests (or structural tests), acceptance tests, and tests to check non-functional requirements, like performance.

Chapter 9: Refactoring presents the main code transformations that can be performed to improve the design of a software system. The presentation includes several source code examples, some from actual refactorings performed on open-source projects. The aim is to provide a practical refactoring experience to the readers and thus to help them develop the habit of frequently improving the design of their code. We conclude with a presentation of code smells, i.e., indicators that

a code structure is not "smelling good" and therefore should be considered for refactoring.

Chapter 10: DevOps describes the movement to bring the development (Devs) and operations (Ops) teams of a software organization closer together. The operations team is responsible for keeping the software up and running, and consists of network administrators, database administrators, and Site Reliability Engineers (SRE), among others. In a traditional culture, these two teams tend to operate independently. That is, the development team implements the system and then "throws it over the wall" to the operations department. To solve this problem, DevOps proposes constant interaction between Devs and Ops teams, from the early days of development. The goal is to reduce the effort involved in the release of new features. In addition to an introduction to DevOps, we will study important practices for embracing this culture, including Version Control Systems, Continuous Integration, and Continuous Deployment and Delivery.

Appendix A: Git covers the essential Git commands, given that version control is an indispensable practice in today's development world.

Bibliography

H. Washizaki, eds., Guide to the Software Engineering Body of Knowledge (SWEBOK Guide), Version 4.0, IEEE Computer Society, 2024.

Armando Fox, David Patterson. Engineering Software as a Service: An Agile Approach Using Cloud Computing. 1st edition, 2014.

Frederick Brooks. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley, anniversary edition, 1995.

Exercises

1. According to Frederick Brooks, software development faces essential difficulties (for which there is no silver bullet) and accidental ones (for which there is a solution). Give an example of an accidental difficulty you have experienced while implementing software systems, even small ones. Suggestion: Consider difficulties related to tools you have used, such as compilers, IDEs, databases, operating systems, etc.

2. Describe the differences between functional and non-functional requirements.

3. Explain why tests can be considered both a software verification and validation activity. Which types of tests are best suited for verification, and which are recommended for validating a software system?

4. Why can't tests prove the absence of bugs?

5. Considering the historical context of early computing, explain why the first software development processes were sequential and based on detailed planning and documentation.

6. Studies show that maintenance and evolution costs constitute 80% or more of a software system's total costs over its lifecycle. Explain why this value is so high.

7. Refactoring is a code transformation that preserves behavior. What is the meaning of the expression *preserve behavior*? In practice, what constraint does it impose on refactoring activities?

8. Give examples of Type A (Acute or mission-critical) and Type B (Business) systems that you've interacted with.

9. Give examples of Type C (Casual) systems that you've implemented yourself.

10. In 2015, it was discovered that millions of cars manufactured by a major automobile company emitted pollutants within legal standards only during laboratory tests. Under normal driving conditions, the cars released higher levels of pollutants to enhance performance. That is, the code possibly included a decision structure similar to the following (simplified for illustrative purposes):

```
if "car being tested in a laboratory"
   "comply with emission standards"
else
   "exceed emission standards"
```

What would you do if your manager asked you to write an `if` like this one? For more information on this episode, refer to this Wikipedia page (link[8]).

[8] https://en.wikipedia.org/wiki/Volkswagen_emissions_scandal