Undergraduate Topics in Computer Science

mirror_mod.use_y = True
mirror_mod.use_z = False
elif _operation == "MIRROR_Z":
mirror_mod.use_x = False
mirror_mod.use_y = False
mirror_mod.use_z = True
#selection at the end -add back the deselected mirror modif
mirror_ob.select= 1
modifier_ob.select=1
bpy.context.scene.objects.active = modifier_ob
print("Selected" + str(modifier_ob)) # modifier ob is the active

Pankaj Jalote

A Concise Introduction to Software Engineering

With Open Source and GenAI

Second Edition

UTiCS

Springer

Undergraduate Topics in Computer Science

Pankaj Jalote

A Concise Introduction to Software Engineering

With Open Source and GenAI

Second Edition

Springer

Pankaj Jalote
Indraprastha Institute of Information
Technology Delhi (IIIT-Delhi)
New Delhi, Delhi, India

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-031-74317-7 ISBN 978-3-031-74318-4 (eBook)
https://doi.org/10.1007/978-3-031-74318-4

Preface

Any software development now will probably use open source software and large language models (LLMs) or generative AI (genAI) for helping in some of the tasks. The open source movement has matured over the last few decades and open source software is now an important and integral part of the software ecosystem. LLMs, which arrived relatively recently, have taken the world by storm and are impacting all activities where generation of text (or other type of artifacts) is the goal, including software development. These two important trends are the reason for the sub-title of this edition "using open source and Generative AI". While this edition of the book, like the previous one, will focus on developing software, for the major tasks involved in a development project, it will also discuss the use of open source and LLMs.

Goals

An introductory course on Software Engineering remains one of the hardest subjects to teach, primarily because of the wide range of topics the area encompasses. I have believed for some time that we often tend to teach too many concepts and topics in an introductory course, resulting in shallow knowledge and insufficient skills for applying the concepts.

I believe that an introductory course on Software Engineering should aim to empower students with the knowledge and skills that are needed to successfully execute a small team project to deliver a modest-sized software application to its users while employing proper practices and techniques. It is worth pointing out that a vast majority of the projects executed in the industry today fall in this scope—executed by a small team over a few months.

Hence, the goal of this book remains the same as before: introduce to the students a limited number of concepts and practices that will achieve the following two objectives:

- Help students develop the skills needed to build a modest-size application in a small team while employing proper practices and methodologies.
- Provide the students with decent conceptual background for undertaking advanced studies in software engineering.

Organization

I have included in this book only important concepts and some contemporary practices through which the two objectives mentioned above can be met. Complete coverage of all the concepts for a particular topic is not a goal. Similarly, advanced topics have also been omitted.

As open source and LLMs are viewed as an integral part of developing an application, they are introduced in Chap. 1, and a brief discussion on using them for the major tasks in a development project is included in relevant chapters. Regarding open source, the book considers not just the use of open source software but also the use of open source practices for developing software, which have also become common and mainstream. Regarding using LLMs, the book is proposing using LLM as a tool like any other tool—the software engineers still remain responsible for whatever tasks they execute and how to use this tool properly. It should be kept in mind that as LLMs are recent and how to effectively use them is a rapidly evolving topic, the suggestions in this book should be taken as a starting point for exploring the use of LLMs.

The book is organized simply, with one chapter for each of the key tasks in a project. For engineering, these tasks are requirements analysis and specification (Chap. 3), architecture design (Chap. 4), software design (Chap. 5), coding (Chap. 6), testing (Chap. 7), and deploying the solution (Chap. 8). The key tasks for project management are project planning, including the development process planning, project monitoring and control, and team-working—these are discussed together in one chapter (Chap. 2). Chapter 1 introduces industry-strength software, quality and productivity in software projects, open-source software, and basics of prompt engineering for using LLMs for software tasks.

Each chapter opens with some introduction and then clearly lists the chapter goals, or what the reader can expect to learn from the chapter. For the task covered in the chapter, the important concepts are first discussed, followed by a discussion of the output of the task, and some practical methods and notations for performing the task. Examples are provided to support the explanations, and the key learnings are summarized in the end for the reader.

Target Audience

The book is primarily intended for an introductory course on Software Engineering in any undergraduate or graduate program. It is targeted at faculty who teach this course and students who take it—generally those who know programming but have not had formal exposure to software engineering. The book can also be used by professionals who are in a similar state—know programming but want to be introduced to the systematic approach of software engineering.

Teaching Support and Supplemental Resources

Though the book is self-contained, some teaching support and supplemental resources are available through a website. The URL is: https://iiitd.ac.in/ConciseIntroToSE-ed2/

The resources available on the site include: the PowerPoint presentations for most topics in PPT format so instructors can change them to suit their style, and some case studies with most of the major outputs of the project which can be used as examples in the course.

New Delhi, India Pankaj Jalote

Acknowledgments

I would like to express my gratitude to my editor, Wayne Wheeler, who encouraged me to revise the book and supported its development.

I would like to thank students in my course Software Development using Open Source, who experimented with using LLMs and open source and other technologies in their projects. Without engagement with them, this book would not have been possible. To express my gratitude, I am happy to commit that royalties from this book will be donated as scholarships to students for higher education and/or to support open source activities among students. Finally, I would like to express my thanks to my wife, Shikha, and my daughters Sumedha and Sunanda for their love and support.

Pankaj Jalote

Contents

1 Industry-Strength Software

Software is about programs that run on computers. Starting from simple programs written in machine language or assembly language, software became more complex as hardware became more powerful and higher level languages became popular. By the 1960s, developing software became a profession, and the discipline of "Software Engineering" had evolved, which focused on methods for developing high quality software in a cost-effective manner.

Earlier, software was added to a system to sell the hardware as the hardware of a system contributed to the majority of its cost. The personal computer changed this as general-purpose software tools became common for private and public use, and the use of software for various purposes exploded.

Software today is everywhere, and new applications and uses keep emerging—we use applications on our mobile phones and on our laptops for a host of our daily tasks like banking, booking tickets, communicating with others, sharing photos, etc. The other equipment and machines we use increasingly have a huge amount of software embedded in them—our cars, washing machines, microwave ovens, vacuum cleaners, etc. Besides individual users, organizations use a lot of software applications for their operations.

These software, which organizations and users rely upon for their work are what we will call as industry-strength software. Developing these industry-strength software is the goal of much of software industry, and which is the domain of the field of software engineering and the focus of this book. In the remainder of this chapter we will learn

- That industry-strength software is extremely different from demo or student software. It is developed in teams, lasts a long time while evolving, and needs to be of high quality.

P. Jalote, *A Concise Introduction to Software Engineering*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-74318-4_1

- Productivity and quality are the critical drivers for software projects, and how they can be characterized.
- Productivity and quality can be improved by using good processes and methodologies, open source software, and LLMs.
- How open source evolved and the different licence types, and open-source libraries and frameworks which can be used for new application development.
- Different prompting approaches for LLMs that can be used for helping in various software tasks.

1.1 Industry-Strength Software

Software can be viewed as the set of programs and associated documentation. In contrast to industry-strength software, we have demo or student software, i.e. software that is largely used to demonstrate something—a capability of what can be done in software, or a capability of the programmer. While underneath both types of software are finally code or programs, there is a world of difference between the two. The term software does not clearly distinguish between the two, even though demo software and industry-strength software are two extremely different types with very different goals and expectations. In this section we will discuss the industry-strength software a bit more, starting with its differences from student software.

1.1.1 Demo and Industry-Strength Software

Give a student an assignment to build an application (say in Java) that finally has about 5000 lines of code (LOC) or 5 KLOC. Most students who are proficient in programming will be easily able to code this in about a month, that too working part-time on it. Let's assume that a student is able to develop this application and demo it about a month, working half time on it, i.e. the total effort the student has put in is two person-weeks or half a person-month of effort. In other words, the student productivity is about 10 KLOC per person-month.

Now let us take an alternative scenario—we act as clients and pose the same problem to a company that is in the business of developing software applications for clients. Though there is no standard productivity figure and it varies a lot, it is fair to say a productivity figure of 1,000 LOC per person-month is quite respectable (though it can be as low as 100 LOC per person-month for embedded systems). With this productivity, a team of professionals in a software organization will take 10 person-months to build this software.

Why is there a difference in productivity in the two scenarios? Why do the same students who can produce software at a productivity of a few thousand LOC per month while in college end up producing only about a thousand LOC per month when working in a company?

The answer, of course, is that two different things are being built in the two scenarios. In the first, a *demo application* or a *student software* is being built, which is primarily meant for demonstration purposes, and is not expected to be used later. Because it is not to be used, nothing of significance depends on the software, and the presence of bugs is not a significant concern. Neither are the other quality issues like usability, maintainability, portability etc.

On the other hand, *industrial strength software*, is built to solve some problem of a client. It is used by the client's organization for operating some parts of the business. A malfunction of such a system can have a huge impact in terms of financial or business loss, inconvenience to users, or loss of property and life. Consequently, the software needs to be of high quality with respect to properties like reliability, usability, portability, etc.

This need for high quality and to satisfy the the end users has a major impact on how software is developed and its cost. The rule of thumb Brooks gives suggests that the industrial strength software may cost about ten times the student software[1].

The software industry is interested in developing industrial strength software, and the field of software engineering focuses on how to methodically and rigorously build such software. That is, the problem domain for software engineering is industry-strength software. In the rest of the book, when we use the term software, we mean industrial-strength software.

Besides the need for high quality, there are two other important characteristics of industry-strength software that further distinguish it from demo software and have a huge influence on how such software is developed.

A key distinguishing characteristic of industry-strength software is that they often tend to be very large (often hundreds of KLOC), and are invariably developed by teams of engineers and rarely by an individual programmer. Hence, effective team-work is essential for developing such applications—a factor that is of minimal importance in demo software. This requires effective processes for developing the application and managing the development, as well as proper teamwork spirit and culture.

The other distinguishing feature is that such software often has a long lifespan of use spanning many years or decades. (Demo software typically has a short lifespan—it is used to demonstrate something and then it may not be used.) This dimension of time brings in some important new issues. The use of software over time inevitably creates demand for new features, which necessitate changes in the existing application. Changing an existing software is a complicated task, and if the software is not developed with its evolution in mind, change can become very hard and tricky. Over the lifetime of a software, the effort required to make necessary changes to an application (sometimes called software maintenance) is many times more than the initial cost of developing the application. To facilitate making changes more easily later as software evolves, the software has to be developed to be much more modular so modules can be changed easily.

The long life of a software also means that over time, different groups of engineers will work on the software and make enhancements. In other words, code written by one programmer will be modified later by other programmers (rarely does this happen in demo software). This requires that the programs be written in a manner that they are easy to understand, and the overall application code be organized so that its design can be easily understood.

1.1.2 Types of Software

At the top level, broadly speaking, there are two types of (industry-strength) software—system software and application software.

- System Software: This is software that helps us use the hardware more efficiently. It forms the layer between the hardware and the users. Typically, system software runs all the time in the background, is preinstalled, and generally runs in system mode. Examples of such software are operating systems (OS), networking software, device drivers, compilers/loaders/linkers, firmware, system utilities like anti-virus, security protection, zipping files, file systems, etc.
- Application Software. This is the most common type of software. These are user programs that help users perform some type of tasks, and generally, a user decides whether to install such an application or not. There are some general purpose applications like browsers, database software, multi media software (e.g. photo/video editing), spreadsheets, word processors, etc., which can be used by the user. There are also customized software applications which are often developed for an organization for the purposes specified by the organization. Examples of this are e-commerce applications, banking software, railway reservations, scientific software for different types of problems, and the applications that companies develop for their clients.

The vast majority of software we see is application software. System software is now developed only in a few organizations; the bulk of the development work today is also about building applications. In this book, our focus is on application software development.

Application software can be broadly divided into two groups—stand alone installable applications that run on a computer system (typically within one address space) and distributed applications where different parts of the application run on different machines in different address spaces. Applications that we download and install and then run on our laptops are often in the first category—they run on our local machine. Applications that we access through the browser or a mobile app (e.g. an e-commerce site, a reservation system, etc.) are distributed applications where the front end of the application runs on our machine (e.g. within the browser or in the mobile), and the back end of the application runs on a different machine (e.g. on the cloud).

The distributed application is also sometimes referred to as software as a service (SaaS)—the software we use through the front-end interface is not provided to us directly—rather, it is provided as services that our front end invokes and uses. With the internet becoming ubiquitous and cloud computing becoming more prevalent, applications are increasingly becoming distributed. Even the applications we download and install on our machine often have some components at the back end hosted on some cloud (e.g. user registry). Mobile applications (apps) that we use and which earlier were generally of the standalone type (e.g. many games or a calculator app) that run on the mobile itself now are often of the distributed kind—the app provides the front end, and some computation which runs on the mobile with many services being provided on servers to which the app makes a request as needed.

1.2 Key Drivers: Productivity and Quality

With industry-strength software, we have two primary parties involved—the consumers of the software and the producers of software. The consumers include the client (or sponsor or customer) who may commission the development of the software and users, who end up using the software for their work. Producers are the development organization and team that develops the software. Getting a high quality software application at a low cost is a basic goal of the consumers. As high productivity reduces the cost of development, developing high quality software with high productivity is the fundamental goal of the software producers. (Delivery time is also a driver, but in software, productivity impacts delivery time; hence, we do not consider it separately.) In this section, we discuss these overarching goals and the need to improve them.

1.2.1 Productivity

Productivity in a project is how much output is produced per unit of the key input. In software projects, as the primary input is human effort, productivity is with respect to unit effort, i.e. person-month. For the output of the software project, we will continue to use thousands of lines of code (KLOC) delivered as the measure of size. While this measure has known deficiencies, this measure can suffice for discussion purposes. (It should be pointed out that to assess the productivity of individual programmers, this is a poor measure as software development is a creative activity, and there are many factors that impact how much code a programmer writes in a unit time.)

Productivity (e.g. in terms of KLOC per person-month) can address both cost and schedule concerns in a software development project. If productivity is higher, it should be clear that the cost in terms of person-months will be lower (the same work can now be done with fewer person-months.) Similarly, if productivity is higher, the potential of completing the project in a shorter time improves—a team of higher productivity is likely to finish the project in less time than a same-size team with lower productivity. (The actual time the project will take depends on the team size and characteristics of the project.) Hence, the pursuit of higher productivity is a primary driving force behind software engineering and a major reason for using different tools and techniques.

It is important to distinguish between code-writing and delivered productivity in a project. Code-writing productivity is how much software a team can develop per day/month in a project. Delivered productivity of a project is how much software has the project delivered per person-month. When most of the software that is delivered to a user is written by programmers in the team, there is no difference between the two—as was the case in earlier days of software. However, today for many applications, the software delivered includes software that is not developed by the team, e.g. by including earlier developed software or software generated through some tools. Hence the delivered productivity can be different from the code-writing productivity. Clearly for users and businesses, the delivered productivity is of importance. We will also focus on delivered productivity.

1.2.2 Quality

For a software application development project, developing a high quality software is clearly a desired goal, and users expect high quality. However, what is quality for a software product or application? The ISO/IEC 9126 standard [2,3] defines eight quality characteristics, each further divided into sub characteristics. These characteristics provide a comprehensive view of software quality. The eight main quality characteristics:

- **Functionality.** The capability to provide functions which meet stated and implied needs when the software is used.
- **Performance.** The capability to provide appropriate performance relative to the amount of resources used.
- **Reliability.** The capability to provide failure-free service.
- **Usability.** The extent to which the software can be understood, learned, used, and attractive to the user when used.
- **Security.** The software's ability to protect information and data so that unauthorized access or modifications are prevented.
- **Maintainability.** The capability to be modified for purposes of making corrections, improvements, or adaptation.
- **Compatibility.** The software's ability to coexist with other software, hardware, or operational environments.
- **Portability.** The capability to be adapted for different specified environments without applying actions or means other than those provided for this purpose in the product.

With multiple dimensions to quality, different projects may emphasize different attributes, and a global single number for quality is not possible. However, despite the fact that there are many quality attributes, reliability is generally accepted to be the main quality criterion. As the unreliability of software comes due to the presence of defects in the software, one measure of quality is the number of defects in the delivered software per unit size (e.g. KLOC). With this as the major quality criterion, the quality objective of a project is to reduce the number of defects (per KLOC) as much as possible before delivering the application. Current best practices in software engineering have reduced the defect density to less than 1 defect per KLOC.

It should be pointed out that to use this definition of quality, what is a defect must be clearly defined. A defect often is considered a problem in the software that causes the software to fail, e.g. crash or malfunction. However, the exact definition of a defect depends on the project or the standards the organization developing the project uses. A suitable definition of defects can incorporate most of the quality characteristics mentioned above. For example, by having issues in the user interface also mentioned as defects (e.g., a problem that causes output to be not properly aligned is a defect), usability can be included in the defect-based quality measure.

To determine the quality of a software product, we need to determine the number of defects in the software that was delivered. This number is unknown at delivery time, though there are approaches to estimate it (e.g. based on past data, defects found in the last stages of testing or beta release). However, the goal of having a low delivered defect density is useful for a project that aims at achieving high quality.

1.2.3 Improving Productivity and Quality

Today, most software consumers expect that the applications they want will be delivered in a short time and with high quality. In other words, the expectation is to deliver a large volume of high quality code for the application in a short time, i.e. the delivered code productivity is expected to be very high. At the same time, we know that writing programs is hard, particularly when they are being written for industry-strength software. Programmers often cannot write more than a few hundred lines of code a week. So, how can this expectation of delivering large applications of high quality in a short time be achieved? Clearly, the productivity of delivered code has to be high while maintaining quality, even if the productivity of writing code is modest. Three top-level strategies can be used by a team to improve productivity.

First is the use of good processes and methods. Software is generally developed by a team of programmers managed by some team leader. From production engineering, and experience in software engineering, we know that overall productivity and quality in a project can be improved by using suitable software development processes and methods. This is the approach traditionally followed by software engineering. Much of the discussion in the book is focused on this aspect—we discuss the development process models and approaches for performing major tasks in a project. Processes and methods can be effectively supported by tools, and a host of tools are available for almost all the tasks to be done in a project.

Besides the use of good methodologies, two other newer strategies can be used to achieve high productivity and quality: the use of open source software and generative AI or Large Language Models (LLMs). We discuss these briefly in the following sections. We will also discuss their use in later chapters.

1.3 Open Source Software

One way to increase productivity is to reuse existing code for parts of the application. This has been proposed as a solution for decades, but recently with the emergence of powerful general purpose open source libraries and frameworks, their use has increased. Most applications developed today utilize such libraries and frameworks. As productivity pertains to delivered software, this strategy clearly improves it. For example, consider an application with 50,000 LOC of delivered code, developed in 5 person-months. Suppose 40,000 LOC comes from freely available libraries and frameworks. The delivered productivity in this project is 10KLOC/person-month, considerably higher than the coding-productivity (which will be 2KLOC/person-month).

It should be clear that this strategy also has a positive impact on quality. Open source software is often of high quality—various reasons have been suggested for this [4,5]. Popular libraries and frameworks having been tested over time by users are generally of very high quality and will have few or no defects (we will assume no defects to simplify our discussion). If the development team develops code with 5 defects/KLOC as the defect density (totalling 50 defects in the 10KLOC developed), the final defect density of the delivered application will be only 1 defect/KLOC (50 defects/50 KLOC)—a significant quality improvement.

Thus, for enhancing productivity and quality in a project, using open source libraries and frameworks, which are often available for free, is important. Here we briefly discuss the evolution of open source software (which is also industry-strength software but has evolved quite differently than commercial software), the concept of libraries and frameworks which facilitate reuse, and open source software licensing.

1.3.1 Evolution of OSS and Current State

Open Source software (OSS) originated in the 50s and 60s when computer scientists in corporate labs like Bell Labs and universities developed and distributed software openly and freely amongst researchers. The source course was always shared (without licensing), as there was no other way to distribute code. The Unix system, created by Bell Labs in the 70s, really started the taste for open source. Its publicly available source code made Unix very popular in academia, and most vendors based their operating systems on the enhancements provided by Unix. The availability of source code facilitated research and innovation.

In the 1980s, Richard Stallman, a programmer at MIT, was perturbed by the loss of access to communally developed software when companies licensed it. He started the Free Software Foundation, believing that software licensing was morally wrong and software should be openly published and developed collaboratively. His team developed the GNU (GNU's Not Unix) OS, Emacs, GCC (the most popular C compiler), Lisp, and more. The Free Software Foundation laid four freedoms for software developers and users:

- Anybody can use/run a program for any purpose
- Study and modify an available program
- Redistribute copies of software
- Improve and share modified versions of the software for public use

Finally, they created the General Public License (GPL) in 1989, which states that anyone has the right to use, study, modify, and distribute the software licensed under it. It is a "copyleft" license, i.e. any derivative works of a GPL-licensed software must also be licensed under GPL. GPL and copyleft became widely used in OSS, preserving the integrity of open-source software and ensuring that any contributions would benefit the community. The license ensured that any altered software versions would be made available, leading to a huge innovation boost.

The Free Software Foundation envisioned to make software free for everyone, considering it a moral and ethical issue. However, others saw sharing code as a practical matter. The term "Open Source" emerged in 1998, focusing on licenses, the culture, and the development approach, without making any moral claims. Generally, OSS is free and is sometimes referred to as Free and Open Source Software, or FOSS.

In the 1990s, OSS became an alternative movement for software development. Widely used open-source tools like Python, PHP, Ruby, Unix BSD, MySQL, Apache Webserver (still the most popular server), Netscape (later became Firefox), and Open Office were developed during this time.

Early open source software aimed to ensure that software was freely available, focusing on operating systems, databases, etc. It then evolved into a platform for developing tools and systems—by developers for developers, generally to solve their problems. Nowadays, it even provides components for building applications and has a vast user base. There are currently thousands of active open-source projects spanning various applications, with most organizations and businesses using some FOSS. It is estimated that more than a million people are involved in the community, and vast majority of codebases have some open-source software as a critical component. Data suggests that open source is a reasonable or even a better choice for software products [6].

Different business models have also allowed for OSS to become popular. Many companies also use and create open source software, e.g. Google's Android, TensorFlow, etc.—by making the software open source they benefit from the larger user and programmer base. Another business model is where companies developing OSS keep the basic version free and distribute an "Enterprise" version for a fee. Clearly, open source is no longer a fringe phenomenon; it has become a mainstream and inseparable part of the software industry. With corporate involvement and widespread use, a code of conduct has also evolved for OSS projects.

Multiple ecosystems have emerged to manage free and open-source software. Free platforms were created to host OSS projects, with SourceForge being the most popular until recently. GitHub is currently the biggest repository for OSS with over 40 Million users. Besides hosting, it provides strong support for developing software in an open source style, making it very popular for software development projects.

In this book we will use examples from GitHub. For more information about the evolution of open source software, the reader can refer to [7,8].

1.3.2 Libraries and Frameworks

Using open source libraries and frameworks is the most common approach for leveraging open source in application development projects—most such projects today use some open source library and/or framework. Here, we briefly explain these two concepts.

A library is a collection of pre-written code that can be used by other programs. It consists of reusable functions, routines, and procedures providing ready-made implementations of commonly used operations. Developers can integrate these libraries into their projects, saving time and effort while benefiting from tested and optimized code.

To use a library, it is imported in the application code through an import statement, and then, the application code can simply call any functions/operations/services provided by the library. During execution, the application code calls library modules as needed. It should be noted that when a library is imported its code essentially becomes part of the application code. This can result in extra code in the application, for example, when only parts of the library are actually used. However, as libraries are often optimized, their size is generally not an issue.

There are thousands of libraries available for many purposes. In Python alone over 200 libraries come with standard Python, and over 100,000 libraries developed by others are freely available. Libraries exist for almost all commonly used operations. There are libraries for: mathematics and numerical computation (e.g. NumPy); data manipulation and analysis (e.g. Pandas); text processing; graphics and visualization; networking and accessing operating system operations; concurrent and parallel execution; database access; building graphical user interfaces; etc. There are also libraries for specific domains, such as image processing or computer vision (e.g. OpenCV); machine learning and artificial intelligence (e.g. TensorFlow); developing web applications; finance (e.g. Quantlib); imaging in medicine (e.g. DICOM); signal processing; using sensors and actuators with Arduino; maps (e.g. Leaflet); geospatial data (e.g. GDAL); blockchain (e.g. web3.js); games development (e.g. unity); etc.

An application designer needs to know about libraries that may be useful for the domain for which the application is being developed and understand what functions they provide. Using libraries can reduce the effort needed to build an application considerably.

A framework, while also providing pre-developed code for application development, is quite different from a library. It is essentially a template for a full application, using application-specific code to provide functionality for an application. It can be viewed as implementing an abstraction of an application offering generic facilities that can be tailored or enhanced by application-specific code developed by the development team, thereby providing application specific functionality. While a framework allows the extension or modification of functionality through application specific code, the framework's code itself is not to be modified. Therefore, like libraries, users utilize the framework code in their application, along with their application specific code.

With framework, there is inversion of control—it is the framework code that calls the functions developed by the application designer, unlike in libraries where the code being developed calls the library functions. In other words, when the application is executed, essentially the framework is executed, which then calls the code written for the application. Since the application developers need to provide some application specific functions to be called by the framework, often the application specific code must be placed in specific directories or named in a certain manner so the framework can locate them easily. Generally, a framework imposes more restrictions on the code organization and the architecture an application can use.

Frameworks often come with a set of libraries to ease the task of integrating the application-specific modules developed by the team with the framework. An example of a framework is Django, widely used for developing web applications. It provides libraries for common tasks (e.g., database interaction, form handling) but also imposes rules for structuring project specific code, integrating it, etc.. This requires application developers to first learn how to use a framework effectively, which often involves considerable effort. As with libraries, using frameworks can result in extra code in the application since an application may not use many of the features or functionalities provided by the framework.

Frameworks, by their nature, are generally for specific types of applications. Common framework categories are: web development/backend frameworks (e.g. Django, Rails, Node.JS); frontend/UI frameworks (e.g. React, angular JS, Bootstrap); mobile app development frameworks (e.g. Flutter); game development frameworks (Unity) software testing frameworks (e.g. pytest), real-time application frameworks.

Libraries are generally more modular and flexible, allowing developers to pick and choose specific components for their projects. They are easier to learn and incorporate into applications. Frameworks provide a more opinionated structure, guiding the entire application architecture and workflow, and require a considerable learning before effective use. Both are valuable in application development, and application designers need to have good familiarity with them in order to leverage their benefits.

1.3.3 Open Source Software Licensing

A license is a legal and binding contract between the software authors and the software users, specifying what users can and cannot do with the software and what obligations they need to fulfil. Even if the software is freely available, many users hesitate to use it without an explicit license. Hence, most open source software will specify their license. When working with open source libraries and frameworks, understanding the licence type of the OSS being used is essential as it can significantly impact how the final application can be distributed.

Though there are many types of open source licenses available, two main categories of open-source licenses are most common—copyleft and permissive. While in early 2000s, more than two-thirds of licenses in use were copyleft while the rest were permissive, now this trend has reversed. Here we will briefly discuss these categories. For further discussion on these and other licenses the reader can refer to [7,9,10].

Copyleft or Reciprocal

The copyleft license type allows the users to use, modify, and share source code as long as all its derivative works are also freely available as copyleft. The GNU general public licence (GPL) is an example of this type of license. GPL-3, the most flexible version, states that anyone can copy, modify, and distribute the covered software freely. However, any software using any part of GPL-licensed software must release its full source code under GPL. If any software uses a GPL component, it is considered a derivative and hence, must also be put under GPL. The license cannot be changed, and no additional terms and conditions can be added. If the modified version is for personal use, then releasing it is unnecessary; however, if made public, the code must be released as well. The software is provided without warranty—authors or licenses cannot be held liable.

This is a "viral" license—the GPL licensed software infects new software and spreads to any software that uses it. Big companies sometimes build Chinese walls or clean rooms to ensure that no code from GPL OSS is used in their software. GPL is a 10-page document and remains the most common copyleft license (about 20% of all licenses).

The Lesser General Public License (Lesser GPL or LGPL) is a weaker and more permissive version of the GPL. It mandates sharing source code only if modifications are made to the original code. Any code that simply uses LGPL-licensed libraries/components need not be made open, allowing the building of proprietary software. In other words, LGPL software may be linked to proprietary software without requiring the release of the code. This makes it more permissive than GPL. Other variants of this type of licence are: Mozilla Public License (MPL), Common Development and Distribution License (CDDL), Reciprocal Public License (RPL), Open Software License (OSL), Common Public License (CPL), etc.

Permissive or Academic

The Permissive-type license guarantees the freedom to use, modify, and redistribute software, including in proprietary software. This means one can modify the software, use it commercially, and keep it private. There is no obligation to share the code under this license—the user only needs to acknowledge the original author/creator.

The MIT License is one of the most permissive licenses. It allows a user to do anything with the software with the only requirement being the inclusion of original MIT License and copyright acknowledgements. It is a simple, clean, and popular

OSS license, with more than a quarter of all OS licenses in use being MIT Licenses. Permissive licenses like MIT do not guarantee that changes will remain publicly available, i.e. a modified version can be licenced under a proprietary license. This allows one to build commercial software or services on top of such OSS. The MIT License is about half a page long and contains three clauses, allowing free usage, copying, modification, merging, publishing, distribution, sub-licensing, selling copies, etc.

The BSD License is another small (one page with three clauses) permissive license. It states that the source code must retain the copyright notice and the binary format of the code should also retain the copyright. However, the name of the copyright holder or contributors cannot be used to endorse or promote the derivative products.

The Apache License is a permissive license spanning over six pages. It allows users to use the software commercially, modify, license, or redistribute it freely. It does not provide any warranty. All redistributions and derivatives must include the copyright notice, license, and any changes made. Other variants of permissive licences also exist, such as, ISC License (Internet Systems Consortium) and CC0 (Creative Commons Zero) License.

1.4 LLMs and Prompt Engineering

Another way to improve productivity in a project is to use generative AI or Large Language Models (LLMs). Recent advances in LLMs like ChatGPT have made generating code or other parts of the software possible. Using an LLM like ChatGPT can enhance the productivity in a software development project by generating parts of the code that programmers would otherwise had to write. For example, consider the project discussed above in which 10 KLOC of code was to be written by software developers and 40 KLOC of code was provided by libraries and frameworks for delivering an application with 50 KLOC code. Of the 10KLOC of code to be written by programmers, if the team can generate 5KLOC code using an LLM with minimal effort, then the effort required to develop the code will be reduced, thereby improving productivity. As LLMs have been trained on intensive codebases, the quality of generated code is often quite good.

The output of an LLM depends significantly on how the prompts given to it are constructed. To gain productivity advantages by using LLMs for some tasks in software development, they must be prompted effectively. If prompting consumes too much effort (e.g. due to trying different prompts) to generate the desired output or produces low quality output requiring considerable effort to make it usable, productivity gains will diminish. Therefore, it is important to understand how to prompt LLMs effectively for software development tasks.

Prompt engineering is the deliberate design and construction of prompts used to elicit desired output from an LLM. Here we will discuss some prompting approaches useful for many software development tasks. We will use ChatGPT as the LLM in our examples.

At a high level, prompting techniques can be divided into two groups—single prompt approaches and multi-prompt approaches. These can be enhanced by structuring the prompts in specific ways. We will discuss some approaches for enhancing each of these. Collectively, they provide common ways to interact with an LLM that will suffice for many software development tasks. This discussion is based on experimentation and inputs from industry professionals. There are online resources that discuss prompting approaches (e.g. https://www.promptingguide.ai/, which has a catalog of prompting techniques with references for each, or https://platform.openai.com/docs/guides/prompt-engineering), and there are papers on prompting for software specific tasks (e.g. [11–13]). Readers can refer to these and other such resources for more information.

It should again be pointed out that in a software development project, the development team remains responsible for all the tasks that need to be performed. An LLM is another tool which a developer can use, but the responsibility for ensuring the task is done properly remains with the developer. Additionally, LLMs and their use is a rapidly evolving area and new approaches are bound to keep coming up, requiring suitable revisions of the set of common approaches for software tasks periodically.

1.4.1 Single Prompt Approaches

LLMs are trained on a vast amounts of data and can often generate the desired output with a single prompt. Single prompts can be quite effective even for moderately complex tasks if constructed well. Providing a single prompt for the task is sometimes referred to as zero-shot prompting. We will first discuss parts of an effective prompt for software tasks, and then cover some special approaches that can be used to achieve better results.

Parts of an Effective Prompt

To get the desired output from a prompt, the developer must explain the task, provide necessary information, specify the type of output that is expected, etc. A well-structured prompt that effectively guides the model to produce accurate and relevant outputs for a wide range of software tasks should contain the following parts:

- **Task Description.** The objective of the task stated in a concise and specific manner. Clearly, this is an essential component of a prompt. Example: "Identify and document the functional requirements for a new online shopping platform."
- **Input Data.** Any relevant input data, context, or examples necessary for completing the task. Most software tasks require some inputs (e.g. generating test cases for a function will require the definition or code for the function). Therefore, under this part of the prompt, all necessary information for the task, such as existing documents, examples, or context should be provided. This helps the model generate outputs that are consistent with the provided input. Example: "Here are the

user stories gathered from stakeholders: (1) As a user, I want to be able to browse products by category. (2) As a user, I want to add items to my shopping cart."

- **Output Format.** The desired format of the output should be clearly specified. This includes detailing the different parts of the output, the ordering of the parts, contents of each part, etc. For example, a software task might require a document with a specific structure, or code in a particular programming language. Specifying the output format clearly ensures that the LLM produces results consistent to the specified format which allows for easier interpretation and usability. Example: "Please provide a document outlining the functional requirements, organized by feature, with clear descriptions and acceptance criteria."
- **Additional Instructions (if any).** Any additional instructions or constraints that the model needs to follow while generating the output. The LLM can be provided specific instructions on how to solve the task or some guidelines, principles, considerations or constraints to follow. Specifying these ensures that the output meets the desired criteria and adheres to any relevant standards or principles. Example: "Ensure that the requirements for each user category are grouped together. Consider scalability and internationalization aspects."
- **Evaluation Criteria.** Outline of how the generated output will be evaluated. This helps the model understand what constitutes a successful completion of the task. When we want a task done, we would like the output to have some desirable properties. Defining the criteria for assessing the quality of the generated output helps in getting the desired output. This can also provide a basis for evaluating the effectiveness of the model's performance and providing feedback for improvement. Example: "We will evaluate the requirements document based on its alignment with stakeholder needs, clarity, completeness, and traceability."

The above template can be applied to a wide range of tasks. The headings of the parts may not be included in the prompt and may be omitted, especially if it is clear what different portions of the prompt are providing. Additionally, not every prompt will have each of these parts. For example, a simple well-understood task may only need the task definition. Some examples are:

> Prompt: Write a python program to take as input a number between 0 and 99, and print the text equivalent of the number

> Prompt: Give an example of using a decorator in Python

However, even slightly more complex tasks may require more parts to be specified. For example, if we want to debug a code which is giving a runtime error, the prompt will have to include both the code and error as inputs for the task. A prompt for this can be:

> Prompt: Debug the following Python code which gives a runtime error
> Code: ...code given here...
> Error: ...the text of the error generated by the runtime...

Sometimes triple-quotes are used to delimit information being provided. For example, if we want a summary of some text/code, we can provide the instruction for the same, and then include the text within triple quotes. This helps the LLM separate instruction from data or context and often results in better results.

ChatGPT allows the ability to separately specify some information for a prompt by explicitly providing two parts—instructions and the nature/structure of the output desired. If this is to be used, then instructions, evaluation criteria and some context can be provided in the first part, and output structure in the second part. A common approach for providing inputs to help LLM perform tasks is to give them examples of solving the task. If the model is given a single example to learn from, it is sometimes called "one-shot prompting" and if multiple examples are given, it is called "few-shot prompting". It should be noted that while examples can guide the model, they do not always enhance its performance and can confuse the LLM resulting in outputs that may be worse than solutions obtained from a zero-shot prompt approach (where no examples were given). An example of few-shot prompting is:

> Prompt: Write a function in Python for computing a function of two numbers.
> Examples of this function are:
> Example 1:
> Inputs: 15, 25
> Answer: 5
> Example 2:
> Inputs: 88, 66
> Answer: 22

Role Prompting

For a task, the prompt can instruct the LLM to respond as if it were taking on a specific role or perspective within a given context. This is called role-prompting. The role can be according to task needs and can be included in a prompt as a role directive such as "You are a Designer/Data Analyst/Cybersecurity specialist". This can be specified along with the task, or as additional instructions. Role prompting is particularly useful for guiding the language model to generate responses that align with the expectations, concerns, and responsibilities of different stakeholders in software development. By instructing the model to assume the role of different stakeholders such as end-users, product managers, developers, or system administrators, one can get focused responses that align with the expectations and requirements of the designated role. Some examples for using this approach are:

> Prompt: Imagine you are an end user interacting with a new e-commerce platform. Describe the features and functionalities you would expect to see for a seamless shopping experience.

> Prompt: You are a system architect responsible for designing a scalable back-end for a social media platform. Describe the architectural components and considerations to ensure high performance and reliability.

Prompts with clearly defined roles can work with lesser context information as the LLM is likely to have an understanding about these roles and their requirements.

Chain-of-Thought

Another effective strategy is the "chain of thought" method, which involves guiding LLM to solve the problem by providing additional instructions. Generally, this involves asking the LLM to solve the problem step-by-step and specifying the steps in the prompt. The LLM will then follow the steps specified—and give the output of performing each step. This give the prompter visibility into how the problem is being solved, which can help better understand the proposed solution, obtaining the solution in the desired format, debugging it, etc. An example of this type of prompting is:

> Prompt: Generate python code for this problem: ask for a file name, then read the numbers given in the file, and then output the mean and medial of all the numbers. I would like you to solve this step by step as follows:
> 1. Assume that the file exists and that the file contains numbers and write code for this
> 2. Then extend the code to handle the case that the file may not exist—in this case program should ask the user to provide a correct file name
> 3. Further extend the code to handle the case that the file may contain some tokens which are not numbers—in this case, the program should just ignore these tokens and report at the end how many such tokens were found

There are many other ways to enhance a prompt to get better output. Most of these are likely to be around what type of information should be provided under the different parts of the template discussed above.

1.4.2 Multi-prompt Approaches

Single prompt approaches work well for many tasks, particularly if the prompts are well structured and provide the desired information precisely. As LLMs get more powerful, the scope of single prompt approaches will expand.

However, for complex tasks, a single prompt is often insufficient and a series or prompts may be needed. We discuss some multi-prompt approaches here. Note that even when using multiple prompts, the approaches mentioned above for single prompt can be employed while constructing each of the prompts, particularly the initial prompt which will generally set the stage for task execution. Techniques like role playing, and providing examples can still be used. We will discuss some common approaches to using multiple prompts.

Prompt Chaining

The most common way of using multiple prompts is prompt-chaining. In this approach, the task is broken down into a series of subtasks, such that the output of one subtask can be used for solving the subsequent ones. The LLM is then prompted for each subtask in sequence creating a chain of prompts. This approach builds context through the replies of each prompt, allowing for a more dynamic and interactive conversation with the model, enabling users to create complex instructions or queries that evolve over multiple interactions. This can improve the output compared to giving the whole task as one prompt. Besides enhancing the performance, prompt chaining also improves transparency—something useful for software designers who, while using the LLM as a tool for their task, are still responsible and answerable for the tasks they execute. It simplifies prompt creation, as each prompt now needs to be straightforward, and if the output of the step is not as desired, allows for corrections or adjustments.

For example, the first prompt may ask the LLM to generate code for a specific functionality, and the subsequent prompts can ask it to enhance the code with some special cases. An example of this type of prompting is:

> Prompt 1. Generate python code for this problem: ask for a file name, then read the numbers given in the file, and then output the mean and median of all the numbers. Assume that the file exists and that the file contains numbers.
> Prompt 2. Now enhance this code to handle the cases where file does not exist.
> Prompt 3. Now enhance the code such that if some inputs in the files are not numbers they are skipped.

This chain of prompts worked as expected—the first prompt generated basic code, the second added exception handling for a file not existing, and the third added the code to check for numbers.

Prompt Chaining with Chain of Thought

Prompt chaining can also be used where further prompts are decided after reviewing the response of the previous prompt. In a sense, the chain-of-thought approach is being added here—the problem is being solved step-by-step, with each subsequent step being determined based on progress so far. Using this approach typically involves a few steps:

1. **Initial Prompt.** Start with an initial prompt to instruct the model about the task or query. This prompt serves as the starting point for generating responses.
2. **Evaluate Model's Response.** Examine the model's response to the initial prompt. Identify areas where the response is accurate, and note any inaccuracies or areas for improvement.
3. **Refine Prompt.** Based on the evaluation of the initial response, refine the prompt to address the identified issues. This may involve providing additional context, rephrasing the prompt, or specifying instructions more explicitly.
4. **Repeat.** Continue refining the prompt until the model produces the desired output or meets the criteria for accuracy and completeness.

This approach can be used for many software development tasks. For example, it can iteratively improve code until the desired result is achieved, solve complex problems or debug scenarios where initial suggestions by the model refine the prompt until the desired answer is obtained. It can also be used for learning where simple examples are first asked for and then progressively more complex concepts or examples are examined.

For example, to identify and fix a bug in a code snippet, the approach can be:

> Initial Prompt: Identify and fix the bug in the following Python code that is intended to calculate the factorial of a number.
> Evaluation: Review the model's response and refine the prompt to focus on specific aspects of the code or provide additional context.
> Refined Prompt: The bug is related to the incorrect initialization of the loop variable. Fix the bug in the code to calculate the factorial correctly.
> Iteration: Continue refining the prompt until the model correctly identifies and fixes the bug.

This approach can be also useful for learning and exploration. For example, to explore learning about sorting algorithms, the prompting can be:

> Initial Prompt: Explain the bubble sort algorithm in simple terms.
> Evaluation: Assess the clarity and accuracy of the explanation. Refine the prompt to improve specific details or add more context.
> Refined Prompt: Elaborate on the time complexity of the bubble sort algorithm and compare it with other sorting algorithms like quicksort.
> Iteration: Continue refining the prompt to get a detailed and accurate explanation.

For generating test cases, this approach can be particularly useful. Based on the test cases it has generated, it can be asked to generate additional test cases to cover conditions not addressed by the initial set. For example, a programmer can run the test cases generated, and then based on the testing, can ask the LLM to generate more test cases for specific conditions.

Multiple Prompts for Self-consistency

The self-consistency approach is also a multi-prompt strategy that involves iteratively refining and adjusting prompts based on the model's responses. The goal is to gradually converge towards a desired output or solution by refining the prompt based on the model's previous responses. At the simplest, the approach involves prompting the model with variations of the same input and task statement to check the consistency in the generated output.

Exploratory Prompting

With multiple prompts, an exploratory approach is also possible. For example, the first prompt may ask the LLM to generate multiple approaches or possible solutions to the problem posed. Then the prompter can review the results, select one of the suggested approaches, and ask the LLM to develop that further. It is possible to explore more than one of the suggested approaches further, and for each, it is possible to again ask for generating multiple approaches for the next step. This type of prompting is like a tree-of-thought exploration. This type of approach is also supported in Copilot for generating code, where the programmer can ask for multiple possibilities and then select from the choices provided. This prompting style is expensive as the prompter has to review all the choices generated, but can lead to better outputs.

1.5 Summary

- The problem domain for software engineering is industrial strength software. This type of software is designed to solve some problem for a set of users, and is expected to be of high quality. It typically lasts a long time while evolving, and is usually developed by a team.
- Productivity and quality are fundamental driving forces in a software project. Productivity is about the amount of code delivered (e.g. KLOC) per unit of programmer effort (e.g. person-month). Software quality encompasses many attributes which include functionality, reliability, usability, efficiency, maintainability, and portability. Since software quality is often affected by defects, it can be characterized by number of defects per thousand lines of code.
- Improving productivity and quality is a goal of software engineering. This can be achieved through proper processes and methods, utilizing open source software libraries and frameworks, and leveraging LLMs.
- Open source software has come a long way since its inception and now thousands of libraries and frameworks are available to developers for free. Using such software can substantially improve productivity in a project. The distribution of an application using an open source library or framework depends on the license of the open source software used—copyleft or restrictive licenses require that the new

software to also be made open source, while permissive licenses allow unrestricted usage and distribution.
- Using LLMs to perform some of the tasks in an application development project is another way to boost productivity. However, using LLMs efficiently requires proper prompting. Single prompt approaches work well for many tasks if prompts contain sufficient context, precise task definitions, and structure and quality of the desired output. In some cases, multi-prompt approaches like prompt-chaining are better suited.

Self Assessment Exercises

1. What are the main differences between a student software and industrial strength software.
2. If developing a program for solving a problem requires effort E, it is estimated that an industrial strength software for solving that problem will require 10E effort. Where do you think this extra effort cost is spent?
3. Productivity is output produced per unit effort. LOC as a measure of output considers the code as the output. Output can also be considered in terms of functionality the software provides. Suggest some user-facing measures to measure the size of the output for a project.
4. Suppose for an application for announcing events and registering for them we are most interested in ensuring that the application has a good look-and-feel, provides fast response time, and is easy to use. For recording defects found in this application, list the types of defects such a project should test for.
5. How would use of open source software improve productivity and quality of an application?
6. If a programmers uses LLM for tasks like design and coding, how will the nature of programmer's work change?
7. Will the use of LLMs always improve productivity? If not, what are the conditions under which using LLM may not improve productivity.
8. An application is using some open source libraries. What will be the impact on use and distribution of this application, if the libraries used are under GPL, LGPL, or MIT licence.
9. Libraries and frameworks both provide code to be used for building a new application. Which of these is likely to be easier to use in a project and why?
10. Single prompt approaches for LLMs can work well in many situations, provided the prompts are constructed well. Please list the parts of a good prompt. What other techniques can you use (besides a well constructed prompt) to get a better response for software tasks?
11. Think of a problem in which single prompt approaches will not work well, and suggest how you will construct series of prompts for this problem.

References

1. F. Brooks, *The Mytical Man Month* (Addison-Wesley, 1975)
2. P. Botella, X. Burgués, J.-P. Carvallo, X. Franch, G. Grau, J. Marco, C. Quer, ISO/IEC 9126 in practice: what do we need to know, in *Software Measurement European Forum* (2004)
3. J. Tian, *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement* (Wiley, New York, 2005)

4. A. Mockus, R.T. Fielding, J.D. Herbsleb, Two case studies of open source software development: Apache and mozilla. ACM Trans. Softw. Eng. Methodol. (TOSEM) **11**(3), 309–346 (2002)
5. K. Crowston, K. Wei, J. Howison, A. Wiggins, Free/libre open-source software development: what we know and what we do not know. ACM Comput. Surv. (CSUR) **44**(2), 1–35 (2008)
6. D.A. Wheeler, Why open source software/free software (oss/fs)? look at the numbers. Accessed 13 Oct 2003
7. H.E. Pearson, Open source licences: open source-the death of proprietary systems? Comput. Law & Secur. Rev. **16**(3), 151–156 (2000)
8. D. Bretthauer, Open source software: A history
9. A.M.S. Laurent, *Understanding Open Source and Free Software Licensing: Guide to Navigating Licensing Issues in Existing & New Software* (O'Reilly Media, Inc., 2004)
10. J. Lindman, M. Rossi, A. Paajanen, Matching open source software licenses with corresponding business models. IEEE Softw. **28**(4), 31–35 (2011)
11. X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, H. Wang, Large language models for software engineering: a systematic literature review (2023). arXiv:2308.10620
12. J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, D. Schmidt, A prompt pattern catalog to enhance prompt engineering with chatgpt (2023). arXiv:2302.11382
13. J. White, S. Hays, Q. Fu, J. Spencer-Smith, D. Schmidt, Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design (2023). arXiv:2303.07839

2 Planning a Software Project

Once a software development project is identified, whether it is a "new idea" for a startup or an application for an organization or user groups, planning must be done on how to execute the project. The goals are to achieve high quality and productivity, as discussed in the previous chapter. There are many aspects for which planning is required, but in this chapter we will focus on three key dimensions—planning the development, managing the development, and teamwork.

The first high-level task of planning is to decide on the development process to be used. Therefore, we will start this chapter with a discussion on various process models that are useful for different types of projects. Once a suitable process model is selected, it must be executed properly, requiring effective management. We will briefly discuss some aspects of managing project execution. Since software development is done in teams, teamwork is essential—without effective teams, achieving high quality and productivity will be harder. We will discuss some aspects of teamwork in one section.

In this chapter, we will discuss:

- Software development process and different models like waterfall, prototyping, iterative, agile. We will also discuss the open source development process, which is quite different from traditional ones.
- How software development projects are planned and managed using traditional project management approaches.
- How project management may be done using an open source process.
- What it means to have an effective team, the key principles for building effective teams, and how these principles can be put into practice by a team.

P. Jalote, *A Concise Introduction to Software Engineering*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-74318-4_2

2.1 Software Development Process

Software engineering is defined as the systematic approach to the development, operation, maintenance, and retirement of software [1]. Processes form the heart of this systematic approach.

A process is a sequence of steps performed for a given purpose [1]. A process specification for a project is a description of the process that should be followed in some project. The actual process is what is implemented in the project. We will use the term project's process to refer to the specification of the process, assuming that the project follows it.

A project's development process defines the engineering tasks and the order in which they should be performed. It limits the degrees of freedom by specifying the activities that must be undertaken and in what order. The process drives the project and heavily influences the outcome—a sub-optimal process can lead to poor quality and/or productivity. Therefore, it is important to follow a suitable process.

A *process model* specifies a general process, that is "optimal" for a class of projects. That is, in the situations in which the model is applicable, using the process model as the project's process will lead to the goal of developing software with high productivity and quality. A process model is essentially a compilation of best practices into a "recipe" for success in the project. In this section we will discuss some of the major models.

2.1.1 Waterfall Model

The simplest process model is the *waterfall model*, which states that development is done in phases organized in a linear order. The model was originally proposed by Royce [2], though variations have evolved depending on the nature of activities and the flow of control between them. In this model, a project begins with a feasibility analysis. Upon successfully demonstrating the feasibility of a project, requirements analysis and project planning begin. Design starts after the requirements analysis is complete, and coding begins after the design is finished. Once programming is completed, the code is integrated and tested. Upon successful completion of testing, the system is installed and is followed by regular operation and maintenance.

The basic idea behind the phases is the *separation of concerns*—each phase deals with a distinct and separate set of concerns. Using this strategy, the large and complex task of building the software is broken into smaller distinct tasks (which, by themselves are quite complex) of specifying requirements, doing design, etc. Separating the concerns and focusing on one in a phase gives a better handle to engineers and managers on dealing with the complexity of the problem.

Executing a project as a linear order of phases requires a certification mechanism at the end of each phase; otherwise, errors from earlier phases get passed on to the next. This is usually done through verification and validation to ensure that the output of a phase is consistent with its input (the output of the previous phase), and with the overall requirements of the system. The model is shown in Fig. 2.1.

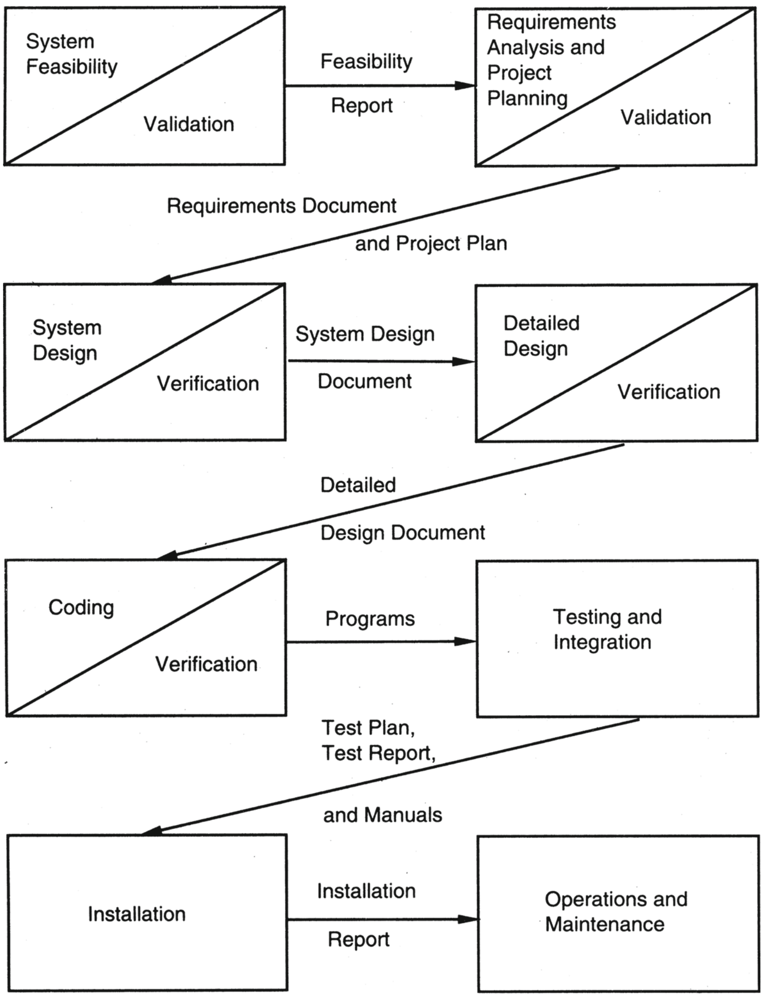


Fig. 2.1 The waterfall model

One of the main advantages of the waterfall model is its simplicity. It is conceptually straightforward and divides the large task of building a software system into a series of cleanly divided phases, each dealing with a separate logical concern.

The waterfall model, although widely used in the past, has some strong limitations that restrict its use today. Some key limitations are:

1. It assumes that the requirements of a system can be frozen before the design begins. This is feasible for systems designed to automate an existing manual process. However for new systems, determining requirements is challenging as the users

may not even know what they need. Hence, having unchanging requirements is unrealistic for such projects.
2. It follows the "big bang" approach—the entire software is delivered in one shot at the end. This entails heavy risks, as the user does not know what they are getting until the very end. Additionally, if the project runs out of funds mid-way, there will be no software delivered. That is, it has the "all or nothing" value proposition.
3. It is a document-driven process that requires formal documentation at the end of each phase.

Despite these limitations, the waterfall model has been the most widely used process model. It is well suited for routine projects where the requirements are well understood.

2.1.2 Prototyping

To address the first limitation of the waterfall model, the prototyping model was proposed. The basic idea is to build a throwaway prototype to help understand the requirements. This prototype is developed based on the currently known requirements. The prototype development obviously involves design, coding, and testing, but each of these phases is not done very formally or thoroughly. By using this prototype, the clients can get an actual feel of the system, enabling them to better understand the requirements of the desired system. This results in more stable requirements that change less frequently.

Prototyping is an attractive idea for complicated and large systems where there is no manual process or existing system to help determine the requirements. In such situations, letting the client "play" with the prototype provides invaluable and intangible inputs that help define the system requirements. It is also an effective method for demonstrating the feasibility of a certain approach. This might be necessary for novel systems, where it is unclear if constraints can be met or if algorithms can be developed to implement the requirements. In both situations, prototyping reduces the risks associated with the project. The process model is shown in Fig. 2.2.

A development process using throwaway prototyping typically proceeds as follows [3]. The development of the prototype typically begins when a preliminary version of the requirements specification document has been created. At this stage,

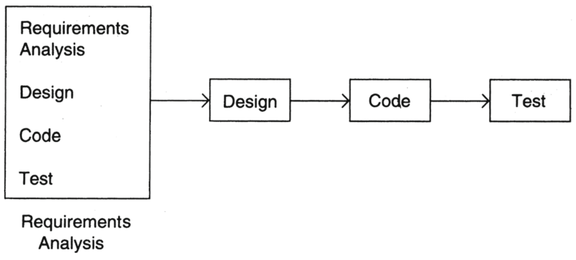


Fig. 2.2 The prototyping model

there is a reasonable understanding of the system and its needs as well as which needs are unclear or likely to change. After the prototype is developed, the end users and clients are given an opportunity to use and explore it. Based on their experience, they provide feedback to the developers regarding what is correct, what needs to be modified, what is missing, what is not needed, etc. Based on the feedback, the prototype is then modified to incorporate some of the suggested changes that can be done easily, and the users and the clients are allowed to use the system again. This cycle repeats until, in the judgment of the prototype developers and analysts, the benefit from further changing the system and obtaining feedback is outweighed by the cost and time involved in making the changes and obtaining the feedback. Based on the feedback, the initial requirements are modified to produce the final requirements specification, which is then used to develop the production quality system.

Overall, prototyping is well suited for projects where requirements are hard to determine and the confidence in the stated requirements is low. In such projects where requirements are not well understood in the beginning, using the prototyping process model can be the most effective method for developing the software. It is also an excellent technique for reducing certain types of risks associated with a project.

A key concern in this model is the additional cost of developing a prototype. Various approaches can be used to manage this cost, such as, including only those features that will provide a valuable return from the user experience, and keeping the development light-weight (e.g. minimizing code for exception handling, recovery, conformance to standards, formats etc.; only essential testing, etc.). The hope is that these reduced costs will be offset by the benefits of having better requirements specification.

2.1.3 Iterative Development

The iterative development process model addresses all the limitations of the waterfall model. The basic idea is that the software should be developed in increments, with each increment adding some functional capability to the system until the full system is implemented. This approach directly addresses the all-or-nothing limitation, and does not require all requirements to be frozen early. Variations of iterative development approach have now become the standard for developing applications.

The iterative enhancement model [4] is an example of this approach. In the first step of this model, a simple initial implementation is done for a subset of the overall problem. Following aspects that are easy to understand and implement, forming a useful and usable system. A *project control list* is created that contains, in order, all the tasks that must be performed to obtain the final implementation. Providing an idea of the projects progress at any given step.

Each step involves removing the next task from the list, designing the implementation for the selected task, coding and testing it, analyzing of the partial system obtained after this step, and updating the list based on the analysis. These three phases are called *the design phase*, *implementation phase*, and *analysis phase*. This process

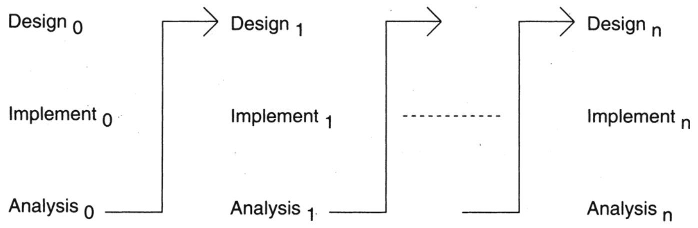


Fig. 2.3 The iterative enhancement model

is iterated until the project control list is empty, resulting in the final implementation of the system. The iterative enhancement model is shown in Fig. 2.3.

Another common approach for iterative development is to select and freeze the architecture of the application, for which the detailed functional requirements need not be known, though understanding of the non-functional requirements is crucial. The functionality is then delivered iteratively. At the start of each delivery iteration, the requirements which will be implemented in that release are decided, and the design is enhanced and code developed to implement them. Each iteration ends with delivery of a working software system providing some value to the end user. Requirements for each iteration are selected primarily based on their value to end users and how critical they are for supporting other requirements.

The main challenge in employing an iterative development model is that the requirements in an iteration may require changes in the design of the software impacting the implementation of existing features also. A loosely coupled architecture and design, can minimize this impact.

A variation in iterative development is to have time boxed iterations. In the timeboxing model, the basic unit of development is a time box, which is of fixed duration. Since the duration is fixed, a key factor in selecting the requirements or features to be built in an iteration is what can fit into the time box. This is in contrast to regular iterative approaches where functionality is selected first and then the time to deliver it is determined. Time-boxing changes the perspective of development and makes the schedule a non-negotiable and a high priority commitment.

To speed up development, parallelism between the different iterations can be employed. One approach to support parallel execution has been proposed in the the timeboxing model [5,6]. In this approach, iterations are timeboxed and each time box is divided into a sequence of stages of approximately equal duration. Each stage performs a clearly defined task for which it has a dedicated team. Having time boxed iterations with stages of equal duration and having dedicated teams renders itself to pipelining of different iterations. (Pipelining is a concept from hardware in which different instructions are executed in parallel, with the execution of a new instruction starting once the first stage of the previous instruction is finished.)

To illustrate the this model, consider a time box consisting of three stages: requirement specification, build, and deployment. These stages are such that they can be done in approximately equal time, and can be relatively independent. When the requirements team finishes requirements for timebox-1, which are given to the build team

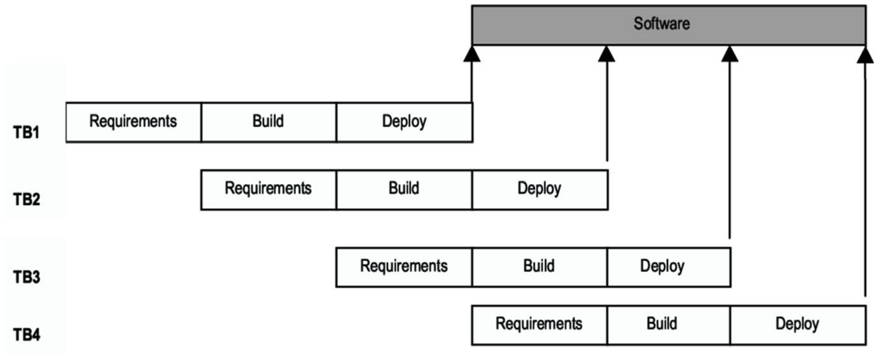


Fig. 2.4 Executing the timeboxing process model

for building the software. The requirement team then starts preparing the requirements for timebox-2. When the build for timebox-1 is completed, the code is handed over to the deployment team, and the build team moves on to build code for requirements for timebox-2, and the requirements team moves on to doing requirements for timebox-3. This pipelined execution of the timeboxing process is shown in Fig. 2.4 [6].

With a three-stage time box, at most three iterations can be in progress concurrently. If the time box is T days, the first delivery occurs after T days. The subsequent deliveries, however, take place every T/3 days. Timeboxing provides an approach for utilizing additional manpower to reduce the delivery time. It is well known that with standard methods of executing projects, substantially compress a projects cycle time by adding more manpower [7]. However, through the timeboxing model, we can use more manpower in a manner such that by parallel execution of different stages we are able to deliver software quicker.

The iterative approaches are becoming extremely popular. There are a few key reasons for its increasing popularity. First and foremost, in today's world clients hesitate to invest too much without seeing tangible returns. In the current business scenario, it is preferable to see returns continuously on the investment made. The iterative model permits this—by delivering some working software after each iteration, allowing clients to see continuous returns on their investment and limiting the risk. Secondly, as businesses are changing rapidly today, they never really know the "complete" requirements for the software, and there is a need to constantly add new capabilities to the software to adapt the business to changing situations. Iterative process allows this. Thirdly, each iteration provides a working system to garner feedback, which helps in developing stable requirements for subsequent iterations. Iterative approach is also employed in the agile processes and the open source process, which we discuss next.

2.1.4 Agile Process

Agile development approaches evolved in the 90s as a reaction to documentation and bureaucracy based processes, particularly the waterfall approach. Agile approaches are based on some common principles, some of which are [www.extremeprogramming.org]:

- Working software is the key measure of progress in a project.
- Therefore, software should be developed and delivered rapidly in small increments for progress in a project.
- Even late changes in the requirements should be entertained (small-increment model of development helps in accommodating them.)
- Face-to-face communication is preferred over documentation.
- Continuous feedback and involvement of customer is necessary for developing good quality software.
- Simple design which evolves and improves with time is better than doing an elaborate design up front for handling all possible scenarios.
- The delivery dates are decided by empowered teams of talented individuals (and are not dictated).

Many detailed agile methodologies have been proposed, some of which are widely used now. Extreme programming (XP) is one of the most popular and well known approaches within the family of agile methods. Like all agile approaches, it believes that changes are inevitable and rather than treating changes as undesirable, development should embrace change. And to accommodate change, the development process has to be lightweight and quick to respond. For this, it develops software iteratively, and avoids reliance on detailed and numerous documents which are hard to maintain. Instead it relies on face-to-face communication, simplicity, and feedback to ensure that the desired changes are quickly and correctly reflected in the programs. Here we briefly discuss the development process of XP, as a representative of an agile process.

An XP project starts with *user stories* which are short descriptions (a few sentences) of scenarios that customers and users would like the system to support. They are different from traditional requirements specification primarily in their lack of detail—user stories do not contain detailed requirements which are uncovered only when the story is to be implemented, allowing details to be therefore decided as late as possible. Each story is written on a separate card, so they can be flexibly grouped.

The empowered development team provides a rough estimate of how long it will take to implement a user story. Using these estimates and the stories, *release planning* is done which defines which stories are to be built in which system release, and the dates of these releases. Frequent and small releases are encouraged. Acceptance tests are also built from the stories, which are used to test the software before the release. Bugs found during the acceptance testing for an iteration can form work items for the next iteration. This overall process is shown in Fig. 2.5.

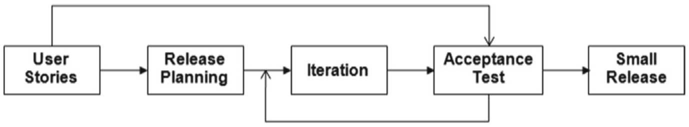


Fig. 2.5 Overall process in XP

Development is done in iterations, each lasting no more than a few weeks. An iteration starts with *Iteration planning*, in which the stories to be implemented in this iteration are selected—high value and high risk stories are considered as high priority and implemented in early iterations. Failed acceptance tests in previous iteration must also be handled. Details of the stories are obtained in the iteration for doing the development.

The development approach used in an iteration has some unique practices. First, it envisages that development is done by pairs of programmers (called pair programming and which we will discuss further in Chap. 6), instead of individual programmers. Second, it suggests that for building a code unit, automated unit tests be written first before the actual code is written, and then the code should be developed to pass the tests. This approach is referred to as test-driven development, in contrast to regular code-first development in which programmers first write code and then think of how to test it. (We will discuss test driven development further in Chap. 6.) As functionality of the unit increases, the unit tests are enhanced first, and then the code is enhanced to pass the new set of unit tests. Third, as it encourages simple solutions as well as change, it is expected that the design of the solution devised earlier may at some point become unsuitable for further development. To handle this situation, it suggests that *refactoring* be done to improve the design, and then use the refactored code for further development. During refactoring, no new functionality is added, and only the design of the existing programs is improved. Fourth, it encourages frequent integration of different units. To avoid too many changes in the base code all happening simultaneously, only one pair at a time can release their changes and integrate into the common code base.

This is a very simplified description of XP. There are many other rules in XP related to issues like rights of programmers and customers, communication between the team members and use of metaphors, trust and visibility to all stakeholders, collective ownership of code (in which any pair can change any code) team management, building quick *spike solutions* to resolve difficult technical and architectural issues or to explore some approach, handling bugs, estimating what can be done within an iteration based on progress made in the previous iteration, how meetings are to be conducted, how a day in the development should start, etc.

XP, and other agile methods, are suitable for situations where the volume and pace of requirements change is high, and where requirement risks are considerable. Because of its reliance on strong communication between all team members, it is effective when teams are collocated and of modest size (up to about 20 members). Furthermore, as it envisages strong involvement of the customer in development, as well as in planning the delivery dates, it works well when the customer is willing to be heavily involved during the entire development proves, acting as a team member.

2.1.5 Open Source Software Process

Unlike commercial software projects which are commissioned by a sponsor, most open source software (OSS) projects are not commissioned. An OSS project usually starts with a group of programmers with a purpose in mind. They develop an initial baseline software (analogous to a working prototype) and make the project open source. (Some projects, however, start as commissioned projects and become open source later, e.g. Netscape, Eclipse.)

Thus, an OSS project often has two distinct stages. The first is the project initiation stage in which an initial working prototyping solution for a problem is developed by a group of programmers. The second is the community stage, where the open-source development actually begins and in which the project invites contributions from a much larger community of programmers. The process used in the second stage is what is generally referred to as open-source process. Unlike other development processes, the open-source process does not have a standard template—an OSS project follows its own process. Many studies have reported the processes followed by some of the major OSS projects (e.g. [8–11]). However, some commonly used practices have emerged. Here we discuss some common aspects of the OSS development process. We include the project initiation stage also for sake of completeness.

- **Project Initiation.** A small team (maybe of one person) identifies some problem and conceptualizes a possible solution for it. The team identifies the technology stack to be used and develops an initial working prototype that can demonstrate the potential of the solution. The team may also develop initial documentation, decide the licence type, and some initial guidelines for contributors.
- **The Team and Core Developers.** The core team at the start of an OSS project is typically the initial team that started the project. The core team then solicits additional team members or collaborators (though mailing lists, community calls, etc.). Teams are distributed worldwide and hence rarely meet face-to-face (in contrast with Agile or traditional processes). All coordination among the teams is electronic. Hence, remote working is inherent to the open-source rather than an alternative approach. Team members are generally volunteers, thus acting as free technical manpower, though some may be hired on a paid basis to work on a specific project. The core team members contribute most of the code. Other members of the team help in identifying and/or resolving bugs, maintaining modules, etc. Over time, some core team members may move out and new members may join.
- **Code Base.** All open-source projects start with an evolving code base, which is stored in a code repository, generally structured as modules (directories with multiple files). Modules help serve as individual components, which can be developed independently by different contributors at different times, promoting distributed and asynchronous development. The repository automatically tracks changes to the

file and allows “undoing” changes. Each module often has an owner responsible for most of the changes, and approving changes made by others.

- **Requirements as Modification Requests.** There is no formal software requirements analysis in the OSS development since a prototype already exists. Development team members generally initiate new requirements or feature requests that emerge from discussion among user groups. The user community may also communicate its needs directly through bug reports or feature requests. The feature requests that are accepted for incorporating in the software, are sometimes called Modification Requests (MRs) or change requests. That is, any new requirement (e.g. a new feature) that is desired in the software is first discussed (over email) and once it is agreed by the core team that it should be included, it is added to the project as a modification request. The priority of the MR is decided by the core group based on the discussions.
- **Issues Tracking, Including Defects.** Most OSS projects maintain an issue log using some easily accessible tool like Bugzilla, GitHub Issues, etc. through which the project keeps track of what is to be done in a project and its progress. All potential work items in the project are listed as issues. Requirements show up in the issue log as an MR (i.e. an issue which is tagged as MR). Defects identified are also logged as issues (with defect tag). Other types of issues may also be there, e.g., refactoring, documentation, refining test scripts, etc. Issues are assigned to team members and are tracked to closure. Often, the list of issues is quite large; hence, only some are implemented for the next release based on priorities set by core developers.
- **Work Assignment.** The unit of work assignment is an issue (an MR or a bug). In OSS, work is not strictly “assigned” by a manager but is taken up voluntarily by contributors. The work can also be assigned to the person managing the module in consideration, but they are also free to decline. The core developers are the most active in addressing issues while other contributors work on non-core modules, minor bug fixes, etc.
- **Testing the Changes.** Testing of the modification being done is the responsibility of the contributor making that change. The contributors test the new code on their private copy to ensure that the changes do not break the application (i.e. the application can build and execute). A programmer can request to update the repository with the changes after successfully executing their tests. Typically, some automated tests for checking that the application continues to work, may be run before accepting an update request.
- **Release.** For releasing the software, typically, there is a feature freeze, in which the code is frozen, and after which only bug fixes are allowed. This is followed by a few weeks of extensive testing of the frozen code, often done as a planned task by the core group. Once testing results are satisfactory, the update is released.

In summary, the open-source development process follows the iterative approach. For each release, the contributors first identify a direction or key new features, bug fixes, and a tentative date for the next release. Each contributor picks up an item from the issue log for implementation. Coding and programmer level testing (e.g. unit and

1. Identify new features and bug fixes for next release in the issues log
2. Plan the date for the next release
3. Repeat by each contributor
    - Pickup one item from the issue log for implementation
    - Code and test in your copy (ensure application does not break)
    - Request code to be integrated in main code
    - Add new issues found to the issues log
4. Freeze the code for release; test the release candidate; fix bugs found
5. Release the code; manage the release

Fig. 2.6 Open-source process

feature testing) are performed in the programmer's private copy to ensure that the application does not break due to changes made by the programmer. Finally, they request their code to be integrated into the main code base, where some automated test scripts may have to run successfully for the code to be accepted. A release candidate is ready when the issues for this release have been closed. The release candidate is then tested rigorously and released if it passes this intense round of testing. This process is shown in Fig. 2.6.

Since the contributors form a community rather than a hierarchically arranged team, the organization of the team is also different. Developers, users, and users-turned-developers form the community for a OSS project. Typical roles in an OSS community include a project leader or an overall administrator, core developers, contributors (active developers, peripheral developers, bug fixers, bug reporters, and readers), and passive users. However, each OSS has a unique community structure, and the roles may also evolve with time. Studies have been done to understand the practices and structure of the community in different OSS projects (e.g. [8,12,13]).

2.2 Project Management

Once a development process has been selected for the project, its application on the project at hand has to be carefully planned and then managed. Project management is done very differently in traditional process models (waterfall, iterative) as compared to open source processes. In this section we will first discuss the planning in traditional projects, followed by project management in open source projects, and briefly touch upon project management in agile.

2.2.1 Traditional Planning and Management

Traditional development projects often start with a contract, which, besides the scope of the project, also often specifies cost and schedule for the delivery. Thus, cost estimation and schedule planning become extremely important tasks in such projects. Because contractual obligations are significant, these projects also include quality plans and risk management plans. Once plans for executing the project are established, one must actively monitor the execution to ensure that the plan is being followed and met, and take suitable actions if there are deviations. We briefly discuss each of these here.

Effort Estimation

Software effort (or cost) estimation has been a very active area of research for decades. In commercial contract-based software development, it still plays an important role. Here we will briefly discuss two approaches that have been used for estimating effort—top down and bottom up.

Although the effort for a project is a function of many parameters, it is generally agreed that the primary factor that controls the effort is the size of the project. That is, the larger the project, the greater the effort requirement. The top down approach utilizes this and considers effort as a function of *project size*. Note that to use this approach, we need to first determine the nature of the function, and then to apply the function, we need to estimate the size of the project for which effort is to be estimated.

If productivity on similar projects is known from past, it can be used as the estimation function to determine effort from the size. If productivity is P KLOC/PM, then the effort estimate for the project will be SIZE/P person-months. Note that since productivity itself depends on the size of the project (larger projects often have lower productivity), this approach works only if the size and type of the new project are similar to the projects from which productivity P was derived (and a similar productivity can be achieved by following a process similar to what was used in earlier projects).

A more general function for determining effort from size that is commonly used is of the form:

$$EFFORT = a \times SIZE^b,$$

where a and b are constants, and project size is generally in KLOC (size could also be in another size measure called function points which can be determined from requirements.) Values for these constants for an organization are determined through regression analysis, which is applied to data from past projects. For example, in the COnstructive COst MOdel (COCOMO) [14,15], for the initial estimate (also called *nominal estimate*) the equation is $E = 3.9(SIZE)^{0.91}$.

Though size is the primary factor affecting cost, other factors also have some effect. Adjustment for these factors can be made to the estimate obtained from size. For example, in the COCOMO model, after determining the initial estimate, some other factors are incorporated for obtaining the final estimate. For doing this, COCOMO uses a set of 15 different attributes of a project called *cost driver attributes*. Examples of the attributes are required software reliability, product complexity, analyst capability, application experience, use of modern tools, and required development schedule. Each cost driver has a rating scale, and for each rating, a multiplying factor is provided. rating scale factors range from very low (0.75), low (0.88), nominal (1.00), high (1.15) and high (1.40). If the reliability requirement for the project is judged to be low then the multiplying factor is 0.75; if it is very high, the factor is 1.40.

Clearly, for top-down estimation to work well, it is important that good estimates for the size of the software be obtained. There is no known "simple" method for estimating the size accurately. The best way may be to get as much detail as possible about the software to be developed, identify the main components, and estimate the size of each component, and then sum them up.

A somewhat different approach for effort estimation is the *bottom-up approach*. In this approach, the project is divided into tasks and then estimates for each task are obtained first. This approach is also called activity-based estimation. Essentially, the size and complexity of the project is captured in the set of tasks the project needs to perform.

One difficulty in this approach is that to get the overall estimate, all the tasks have to be enumerated. To avoid the tedious task of listing all tasks for estimation, a more practical way is sometimes used in commercial organizations [16]. In this approach, the major programs (or units or modules) in the software being built are first determined. Each program unit is then classified as simple, medium, or complex based on certain criteria. For each category, an average effort for coding (and unit testing) is decided, based on past data from similar projects.

Once the number of units in each category of complexity is known, the total coding effort for the project can be determined. From the total coding effort, the effort required for other activities is determined as a percentage of coding effort, using effort distribution information from past projects. From these estimates, the total effort for the project is obtained.

Quality Planning

A project will have a high-level quality plan identifying the set of quality related activities that the project plans to undertake to achieve its quality goals. Let us first understand the defect injection and removal cycle, as it is defects that determine the quality of the final delivered software.

Software development is a highly people-oriented activity and hence is error-prone. In a software project, we start with no defects (there is no software to contain defects). Defects are injected into the software during the different phases of the project. That is, during the transformation from user needs to software to satisfy those needs, defects are injected in the transformation activities undertaken. These injection stages primarily include the requirements specification, high-level design, detailed design, and coding. To ensure that high quality software is delivered, these defects are removed through the quality control (QC) activities. The QC activities

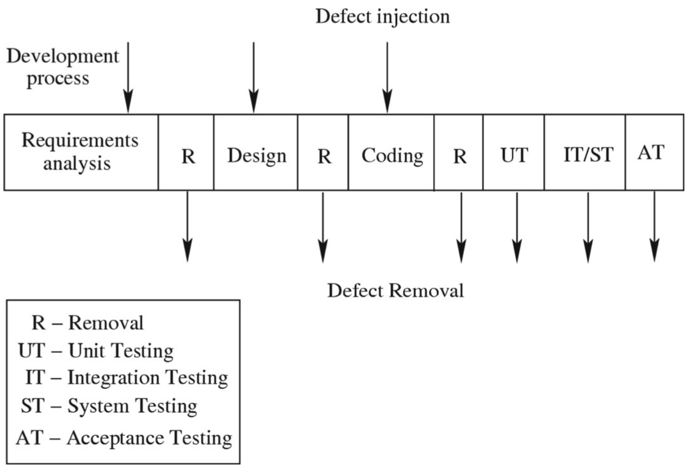


Fig. 2.7 Defect injection and removal cycle

for defect removal include requirements reviews, design reviews, code reviews, unit testing, integration testing, system testing, acceptance testing, etc. Figure 2.7 shows the process of defect injection and removal.

As the final goal is to deliver software with low defect density, ensuring quality in a project revolves around two main themes: reducing the defects being injected, and increasing the defects being removed. The first is often done through standards, methodologies, following of good processes, etc., which help reduce the chances of errors by the project personnel. (There are also specific techniques for defect prevention also.)

To remove injected defects, reviews and testing are two most common QC activities utilized in a project, as shown in the figure. Reviews are structured, human-oriented processes, whereas testing is the process of executing software (or parts of it) in an attempt to identify defects. The most common approach to quality planning in a project is to specify the QC activities to be performed, and have suitable guidelines for each QC task, to increase the chances of meeting quality goals. During project execution, these activities are scheduled, carried out in accordance to the defined procedures, and monitored.

Risk Management

Risk is defined as exposure to the chance of injury or loss. That is, risk implies a possibility that something negative may happen. In the context of software projects, negative implies an adverse effect on cost, quality, or schedule. *Risk management* tries to minimize the impact of risks on these aspects.

Risk management can be considered as dealing with the possibility and actual occurrence of events that are not "regular" or commonly expected, that is, they are probabilistic. The commonly expected events, such as people going on leave or some requirements changing, are handled by normal project management. So, in a sense, risk management begins where normal project management ends. It deals with events that are infrequent, somewhat beyond the control of the project management, and can significantly impact the project.

Most projects encounter risk. The idea of risk management is to minimize the likelihood of risks materializing, if possible, or to minimize their effects if they do. For this, a common approach is to identify the top few risk items and then focus on them. Once a project manager has identified and prioritized the risks, the question then becomes what to do about them. Knowing the risks is of value only if you can prepare a plan so that their consequences are minimal—that is the basic goal of risk management. This is achieved through risk mitigation steps.

One obvious strategy is risk avoidance, which entails taking actions to avoid the risk altogether. For example, the risk associated with use of an emerging data management product can be avoided by opting for an alternate, well established, database product. For some risks, avoidance might be possible.

For most risks, the strategy is to perform the actions that will either reduce the probability of the risk materializing or reduce the loss if it does. These are called risk mitigation steps. To plan what mitigation steps to take, a list of commonly used risk mitigation steps for various risks is very useful in this context. Using these tests, the risk mitigation steps to be executed as part of the project can be identified.

Project Schedule and Project Management Plan

After establishing the effort estimate, we need to establish an overall high-level delivery schedule—when the project plans to deliver the application and when major milestones will be reached. For each milestone, a detailed schedule must be planned with different tasks assigned to different people at different times, such that the milestone can be met. This detailed schedule is often called the project management plan. It is often the most live document controlling the project execution, establishing not only the goals but also listing tasks to be executed (which change as the project proceeds) successfully. We will first discuss the overall scheduling and then the detailed scheduling.

With the effort estimate (in person-months), it may be tempting to pick any project duration based on convenience and then fixing a suitable team size to ensure that the total effort with the team size and duration matches the estimated effort. However, as is well known, people and months are not fully interchangeable in a software project [7].

Still, for a project with some estimated effort, multiple schedules (or project duration) are indeed possible. For example, for a project whose effort estimate is 56 person-months, a total schedule of 8 months is possible with 7 people. A schedule of 7 months with 8 people is also possible, as is a schedule of approximately 9 months with 6 people. (But a schedule of 1 month with 56 people is not possible. Similarly, no one would execute the project in 28 months with 2 people.) In other words, once the effort is fixed, there is some flexibility in setting the schedule by appropriately staffing the project, but this flexibility is not unlimited.

A method for estimating an initial schedule for medium-sized projects is to use the rule of thumb called the *square root check* [16]. This check suggests that the proposed schedule can be around the square root of the total effort in person-months. This schedule can be met if suitable resources are assigned to the project. For example, if the effort estimate is 50 person-months, a schedule of about 7–8 months will be suitable with about 7–8 people. This estimate can be refined based on other project properties.

The effort and overall schedule estimates are inputs for developing a detailed plan or schedule which can then be followed in the project. The detailed plan assigns work items to individual members of the team.

For the detailed schedule, the major milestones identified during effort and schedule estimation, are broken into small schedulable tasks. For each detailed task, the project manager estimates the time required to complete it and assigns it to a team member (resource) so that the overall schedule is met, and the overall effort also matches. In addition to the engineering tasks that are the outcome of the development process, the QC tasks identified in the quality plan, the monitoring activities defined in the monitoring plan, and the risk mitigation activities should also be scheduled.

Generally, the project manager refines the tasks to a level so that the lowest-level activity can be scheduled to occupy no more than a few days from a single person. Activities related to tasks such as project management, coordination, database management, and configuration management may also be listed in the schedule, even though these activities are ongoing tasks.

Rarely will a project manager complete the detailed schedule of the entire project all at once. Once the overall schedule is fixed, detailing for a phase may only be done at the start of that phase.

For detailed scheduling, tools like Microsoft Project or a spreadsheet can be very useful. For each lowest-level activity, the project manager specifies the effort, duration, start date, end date, and the person it is assigned to. Dependencies between activities, due either to an inherent dependency (for example, you can conduct a unit test plan for a program only after it has been coded) or to a resource-related dependency (the same resource is assigned two tasks) may also be specified. An example of parts of the detailed schedule of a project is shown in Table 2.1 [16] (this project finally had a total of about 325 schedulable tasks.)

A detailed project schedule is never static. Changes may be needed because the actual progress in the project may be different from what was planned, due to newer tasks added in response to change requests, or because of other unforeseen situations. Changes are made as and when the need arises.

The final schedule is often the most “live” project plan document. During the project, if plans must be changed and additional activities added, after the decision is made, the changes must be reflected in the detailed schedule, as this reflects the tasks actually planned to be performed. Hence, the detailed schedule becomes the main document that tracks the activities and schedule. An example of such a plan and how it is obtained from estimates is given in [16].

Table 2.1 Portion of the detailed schedule

Module	Task	Effort (days)	Start date	End date	% done	Person
–	Requirements	1.33	7/10	7/21	100	bb,bj
–	Design review	0.9	7/11	7/12	100	bb,bj,sb
–	Rework	0.8	7/12	7/13	100	bj, sb
History	Coding	1.87	7/10	7/12	100	hp
History	Review UC17	0.27	7/14	7/14	100	bj,dd
History	Review UC19	0.27	7/14	7/14	100	bj,dd
History	Rework	2.49	7/17	7/17	100	dd,sb,hp
History	Test UC17	0.62	7/18	7/18	100	sb
History	Test UC19	0.62	7/18	7/18	100	hp
History	Rework	0.71	7/18	7/18	100	bj,sb,hp
Config.	Reconciliation	2.49	7/19	7/19	100	bj,sb,hp
Mgmt.	Tracking	2.13	7/10	7/19	100	bb
Quality	Analysis	0.62	7/19	7/19	100	bb

Project Monitoring

A project management plan is merely a document, detailing what tasks are to be done and when, to guide the execution of a project. Even a good plan is useless unless it is properly executed. And execution cannot be properly driven by the plan unless it is monitored carefully and the actual performance is tracked against the plan.

The main goal of project managers for monitoring a project is to gain visibility into the project execution to determine whether any action needs to be taken to ensure that the project goals are met. As project goals are in terms of effort, schedule, and quality, the focus of monitoring is typically on these aspects. Some common monitoring activities are:

Activity-level monitoring ensures that each activity in the detailed schedule has been done properly and within time. In other words, scheduled tasks are being completed as per the plan. This type of monitoring may be done daily in project team meetings or by the project manager checking the status of tasks scheduled to be completed that day. A completed task is often marked as 100% complete in detailed schedule—this is used by tools to track the percentage completion of a milestone or the overall project.

Defect tracking and quality monitoring. Because defects have a direct relationship to software quality, tracking of defects is critical for ensuring quality. A large software project may have thousands of defects that are found by different people at different stages. To keep a track of the defects found and their status, they must be logged and their closure tracked. If defects found are being logged, monitoring can focus on how many defects have been found, what percentage of defects are still open, and other issues. Defect tracking is considered one of the best practices for managing a project and is done in all large projects.

The *milestone analysis* is done at each milestone or after an iteration is complete. Analysis of actual versus estimated for effort and schedule is often included in the milestone analysis. Significant deviation may indicate that the project would run into trouble and might not meet its objectives. This situation calls for project managers to understand the reasons for the variation and apply corrective and preventive actions necessary. Defects found by different quality control tasks, and the number of defects fixed so far may also be analyzed.

Risk monitoring is another important activity that is done during project execution. It essentially tracks all identified risks and the activities needed to mitigate them. If some risks do materialize, then suitable actions must be added to the project management plan for that event.

2.2.2 Open Source Development

The open source development process has been explained above—it focuses on projects which have an existing code base and a community. Here we will briefly discuss how to utilize open source processes for developing and managing a software application development project, which may or may not later be made into an open source software with a community. The discussion here is based on GitHub platform, which is now being widely used by programmers for all types of projects. Managing the coding activity in a software development, which is quite involved, is discussed later in the coding chapter—here we will focus only on project management.

Planning and Starting a Project

In open source process, as discussed, an initial code base is assumed to be present. On GitHub, a project can be initiated by creating a project repository. When a repository is created, say by the project team lead, other members of the team are added as contributors. The licence type of the project is also specified.

All projects should explain the purpose of the project and the organization by having a README file. This provides a way to document the purpose of the project and some other matters that are of interest to the team, particularly to any new team members. In addition to this, for projects with documents like SRS, architecture, design, etc., it is desirable to create a "Documents" folder to store these files.

Scheduling and Planning Milestones

In open source, the entire project management revolves around making changes to the code in a controlled manner and periodically releasing the product. On GitHub, the primary mechanism to plan for timely release of an application is milestones. In a repository for an application development, multiple milestones can be created. One simple way to use this mechanism is to define main project milestones, with each release also being a milestone. This is a simple milestone-level schedule planning, with no provision for assigning deadlines within a milestone, hence all deadlines are controlled by establishing milestones.

In an open source project, and on GitHub, issues are the only project related items that can be assigned to a team member. Thus, all work items which need to be assigned to team members must be recorded as issues. Issues can be included in milestones, providing a way to combine work items for a common goal. Each issue can be tagged with labels, which allows different types of issues to be managed well (as issues can be filtered using these labels.) Labels are often used to classify the nature of different issues.

GitHub offers a standard set of labels which can be changed—it is best to have the labels that are most suitable for a project. For example, for a simple application being developed by a small team, only a few labels representing key work items to be done (e.g. feature, enhancement, testing, bug fixing, documentation) might suffice. Studies have shown that the common labels used are: enhancement, bug, feature, question and documentation [17].

How to use issues based project management for an application development project? Initially all the requirements can be added as feature issues in the repository, with those to be implemented for the first milestone assigned accordingly. When planning for a new milestone, all issues that are features/requirements are reviewed and decision are made about which ones to include in the next milestone.

In a project one may want to plan for when some work item may be started and when it should finish. In GitHub, this control is fairly basic, keeping in spirit with the open source practices. An issue is assigned to someone with only the end date in terms of milestone specified. If some task depends on another task in the same milestone, then this has to be managed through communication and collaboration.

Project Tracking and Release

Milestones serve as the primary mechanism for project tracking in GitHub. If all issues related to a milestone are assigned to this milestone, it’s easy to see the number of open issues and those that have been completed at any given time. For more visibility and control, project management tools can be employed—GitHub also provides additional capabilities in this regard.

A release typically will be planned as a milestone, and then managed. A better approach may be to treat it as two milestones—one for completing the development (i.e. adding the features and fixing the bugs assigned for this release), and another for releasing the application after a period of testing and bug fixing. The features to be added and bugs to be fixed can be tracked using the regular milestone tracking. The second milestone focuses on special testing and may also restrict addition of features. This can be achieved by not approving any requests to merge code for new features into the main code and only allowing those changes which have been done successfully to fix the bugs that need fixing to be merged.

With this type of managing, the most live document for project management is the list of issues (it is like the project management plan in traditional management) and how their completion is helping progress towards achieving the milestones.

2.3 Team-Working

Software is developed in teams. Hence, besides having good technical capability, software developers must possess team-working and related skills (e.g. communication). Surveys show that professionals rate the ability to work effectively in teams as a critical skill. In this section we will briefly discuss some basic concepts related to effective team-working.

2.3.1 Effective Teams

Professional coders always work in teams. There is virtually no software project without a team. Software development should be viewed as a team sport, not an individual one. The effectiveness of a team is directly correlated with the output, productivity, and quality of the project. Most projects succeed only due to team effort. Genius programmers is a myth [18]—even super individual programmers cannot build large software projects in a reasonable time without an effective team. For developing software, the goal is to have effective teams.

The two key stakeholders in a project are the users or sponsors and the development team. The users/sponsors are satisfied with a quality product, while the team members are satisfied with good teamwork and motivation. The goal of an effective team is to achieve both of these objectives—happy users and happy team [19].

Any team working towards a goal is always subject to external constraints; for example, the delivery deadlines. However, how the team works is decided by the team itself. Effective teams usually require a good degree of autonomy regarding the different aspects of teamwork, such as task allocation, culture, quality, etc. We will assume that within some larger constraints, the team is self-managing and has this autonomy.

No team can function effectively without a responsible team lead. So, teams must elect or appoint a team lead. The team lead is generally responsible for driving the team, building consensus, maintaining adequate and transparent communication and coordination, communicating with external stakeholders, helping resolve conflicts, etc.

The main goal of a team leader is to help the team to become and remain effective. A team lead is not a "boss"; rather is a member of the team with some additional responsibilities. The team lead must set clear and realistic goals, be a catalyst for solving issues rather than always being the solver, support team members and keep the team motivated. While carrying the team along, a team lead must not ignore non-performers or non-contributors, as this compromises the project outcomes and the spirit of teamwork. A lead must also not try to keep everyone happy—this is not the goal of the team, a successful project is the goal.

2.3.2 Pillars of Teamwork

Teams are not about individuals in them; they are about togetherness. Three essential building blocks make teamwork effective: Humility, Respect, and Trust (HRT) [18, 20]. If each member of the team is guided by HRT, the team will be an effective one. For a team member these can mean [18,20]:

- **Humility (with Self-Confidence).** Humility involves recognizing that you are not always correct (i.e. not infallible). There is always a scope for improvement, growth, and learning from others, even if you are a good programmer. It also means that you not identify yourself with your code or artefact, and should not be defensive about bugs, defects, or issues in your code. Recognize that even though you remain the author, the code belongs to the team/project. Hence there is no need to distinguish between code written by any peer in the team and your code. You must also believe that every team member is equal—and you are equal to others in the team (even if you feel you are more knowledgeable). However, it is also necessary to be confident in your own abilities and not feel the desire to hide your work due to feeling of inadequacy or insecurity. You should take constructive criticism from others as opportunities for self-improvement and for improving your code. On the flip side, you should be ready to help others.
- **Respect.** Even if you are a better programmer than your team members by some measure, it is imperative to have a sense of respect and genuine care for others in the team. You must appreciate others' abilities and help them where you can, and take help where needed. You must treat every team member with respect, regardless of their capabilities, and be respectful and gentle when criticising the work of peers—remember to only criticise the work/output, never the person. You must also listen to others' opinions about your own work, even if they know less about your aspect of the project. You should not dominate discussions and give everyone the time and opportunity to share their views.
- **Trust.** Trust among team members is about delivering work on time and with decent quality, giving constructive criticisms on code with respect for the coder, not shirking from volunteering for work to keep the workload evenly distributed, ensuring fairness and transparency, and caring for other's individual's interests (as long as they align with the team's goal). You must trust other team members and believe that they will complete their jobs correctly and make the right collaborative decisions for the project. On the other hand, you must realise that others also trust you, and so you must do the right thing as well to maintain trust.

The working style of a team, which should be based on the three pillars mentioned above, is its team culture. Different groups have different styles of working. "Culture" is an umbrella term covering shared values, goals, processes of working, and the social/behavioral practices of an environment. Team culture is built by active steps of starting/seeding the culture, following it, and strengthening it over time, motivating others to abide by it.

Effective and efficient communication is an integral element of a sound team culture. Both synchronous and asynchronous communication must be used - involving meeting in groups and individually. Synchronous communication refers to regular short and focused stand-up meetings. Using informal channels to discuss ideas in the early stages of work and using issue trackers provides visible and transparent communication to the entire team.

2.3.3 Teamwork in Practice

While the HRT principles should guide the actions of each team member to foster effective teamwork, some steps can be taken to ensure a good team culture is established based on these principles. Here we outline a few which could be useful for a small team working together for a few months on a project. It is assumed that the team has appointed / elected a team lead, who is a team member but with some additional responsibilities.

- Establish a few team practices (easiest to borrow some from other's experience) and clearly articulate them after due discussions and concurrence within the team. One essential practice should generally be holding regular meetings. Another is to have on some practices (e.g. going around and asking for everyone's view) that ensure that all team members are heard and are engaged as equals.
- Establish team's standard for common tasks like coding, commenting, reviewing code, testing, etc. (best to borrow some and adapt them for the project) and follow the standards. Deviations by a team member can be pointed out by referring to the standard, not as accusations.
- Each team member should complete assigned work with the quality expected and on time—there should be no excuses for compromising these as others depend on the tasks to be done properly and in time and violation of this trust can lead to a loss of respect within the team.
- Take reviews seriously as helping the project and the team member whose code (or any other output) is being reviewed. Complete reviews promptly (generally within a day) as the author may be waiting for feedback, and without its completion, the code may not be merged. Review comments should be focused on the code and be never about the programmer and should be phrased gently as suggestions (e.g. so there should never be a comment like:"you did X, you should do Y"; instead it should be of type: "instead of doing X, a better way might be Y")
- Address each review comment properly to respect the effort made by the team member, and view review comments as opportunities to improve code and yourself.
- Take decisions about the project in consultation with the team and communicate them clearly to all team members.

- Maintain transparency and good communication throughout the project. Each team member should be aware of what is happening in the project, understand the rationale behind decisions, and be informed of any hurdles, situations or setbacks, etc. This clarity fosters a cohesive and well informed team environment.

2.4 Summary

- A development process is the set of activities performed in some order during a project. The process followed impacts the productivity and quality achieved. Process models specify general process structures which, if followed, can result in achieving desired goals of high productivity and quality in some types of projects. Many process models have been proposed including waterfall, prototyping, iterative and agile. Open source development projects often follow a process model which is somewhat different from the traditional models—it is more lightweight and flexible and relies on team members to self assign the tasks that need to be done.
- Traditional project planning requires the overall effort for the project to be first estimated using various estimation models. From the effort estimate, overall schedule is estimated and key milestones are identified. For execution, the overall schedule is broken into a detailed schedule of tasks to be done so as to meet the milestones. These tasks are assigned to specific team members, with identified start and end dates. Tasks planned for quality, monitoring, and risk management are also scheduled in the detailed schedule. It is the most live project planning document and any changes in the project plan are reflected suitably in the detailed schedule. It is also the basis of monitoring the project.
- The project management approach of open source projects is different and is based on a list of issues in the project. Issues are suitably tagged (e.g. coding, testing, documentation, refactoring, bug fixing) to identify different types of tasks, and closed when done. No formal effort estimation is done. The schedule is decided based on what is desired in the next release, and for that some milestones are identified, the final being the release of this version. Issues (tasks) in the issue list are mapped to milestones for monitoring and issues are self-assigned by members of the team.
- Effective teamwork is essential for a software project—software development is a team sport, not an individual one. For a project's success it is important to have effective teams. The three pillars on which effective team working rests are: humility, respect, and trust (HRT). Humility is about each individual recognizing that he/she is not infallible and can learn from other team members. Respect is about respecting other team members as equal as far as the team and project is concerned, and appreciate their abilities and contributions. Respect also implies that while reviewing a team member's work, only the work is criticised and never the team member. Trust is about believing in the other team members to do what is right for the project and that they will deliver their assigned work in time and

with quality, which also implies that each member must do the same so as to earn other members' trust. Based on HRT, teams should establish a few practices for the team to follow to maintain a cohesive and productive working environment.

Self Assessment Exercises

1. What is the relationship between a process model and the process for a project?
2. What are the key outputs during an iteration in a project following an iterative development model.
3. Which of the development process models discussed in this chapter would you follow for the following projects?

   a. A simple data processing project.
   b. A data entry system for office staff that has never used computers before. The user interface and user-friendliness are extremely important.
   c. A spreadsheet system that has some basic features and many other desirable features that use these basic features.
   d. A web-based system for a start up where requirements are changing fast and where an in-house development team is available for all aspects of the project.
   e. A Web site for an on-line store which has a long list of desired features it wants to add, and it wants a new release with new features do be done very frequently.

4. There are two stages in an OSS development—developing the initial software and then open sourcing it. (i) What development model best describes the process likely to be used for the first stage. (ii) Briefly describe the steps in the 2nd stage.
5. In a traditional project, what are the key tasks that are done during project planning? How is project monitoring usually done?
6. In a project explain where defects are likely to be injected and what activities can be used for their removal?
7. In a project using open source process, briefly describe the important project planning and monitoring tasks that are to be done during an iteration leading to a release.
8. Give some attributes of an effective team—relate each attribute to the key pillars of teamwork.
9. Suppose you are to lead a project which has some developers who are new to your organization. Suggest a few actions you will take to ensure that an effective team is formed.

References

1. IEEE, IEEE standard glossary of software engineering terminology. Technical Report (1990)
2. W.W. Royce, Managing the development of large software systems, in *Proc. 9th International Conference on Software Engineering (ICSE-9); originally in IEEE Wescon, Aug 1970* (IEEE, 1987), pp. 328–338
3. H. Gomma, D.B.H. Scott, Prototyping as a tool in the specification of user requirements, in *Fifth International Conference on Software Engineering* (1981), pp. 333–341
4. V.R. Basili, A. Turner, Iterative enhancement, a practical technique for software development. IEEE Trans. Softw. Engineering SE-1 **4** (1975)

5. P. Jalote, A. Palit, P. Kurien, V.T. Peethamber, Timeboxing: a process model for iterative software development. J. Syst. Softw. **70**, 117–127 (2003)
6. P. Jalote, A. Palit, P. Kurien, The timeboxing process model for iterative software development, in *Advances in Computers*, vol. 62 (Academic, 2004), pp. 67–103
7. F. Brooks, *The Mytical Man Month* (Addison-Wesley, 1975)
8. K. Crowston, K. Wei, J. Howison, A. Wiggins, Free/libre open-source software development: what we know and what we do not know. ACM Comput. Surv. (CSUR) **44**(2), 1–35 (2008)
9. A. Mockus, R.T. Fielding, J.D. Herbsleb, Two case studies of open source software development: Apache and mozilla. ACM Trans. Softw. Eng. Methodol. (TOSEM) **11**(3), 309–346 (2002)
10. E. Raymond, The cathedral and the bazaar. Knowl. Technol. & Policy **12**(3), 23–49 (1999)
11. W. Scacchi, J. Feller, B. Fitzgerald, S. Hissam, K. Lakhani, Understanding free/open source software development processes. Softw. Process: Improv. Pract. **11**(2), 95–105 (2006)
12. G. Von Krogh, S. Spaeth, K.R. Lakhani, Community, joining, and specialization in open source software innovation: a case study. Res. Pol. **32**(7), 1217–1241 (2003)
13. N. Ducheneaut, Socialization in an open source software community: a socio-technical analysis. Comput. Supported Coop. Work (CSCW) **14**, 323–368 (2005)
14. B.W. Boehm, *Software Engineering Economics* (Prentice Hall, Englewood Cliffs, NJ, 1981)
15. B.W. Boehm, Software engineering economics. IEEE Trans. Softw. Eng. **10**(1), 135–152 (1984)
16. P. Jalote, *Software Project Management in Practice* (Addison-Wesley, 2002)
17. J. Cabot, J.L.C. Izquierdo, V. Cosentino, B. Rolandi, Exploring the use of labels to categorize issues in open-source software projects, in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (IEEE, 2015), pp. 550–554
18. B. Fitzpatrick, B. Collins-Sussman, *Debugging Teams: Better Productivity Through Collaboration* (O'Reilly Media, Inc., 2015)
19. C. Verwijs, D. Russo, A theory of scrum team effectiveness. ACM Trans. Softw. Eng. Methodol. **32**(3), 1–51 (2023)
20. B. Fitzpatrick, How to work well on teams (2024)

3 Software Requirements Analysis and Specification

IEEE defines a requirement as "(1) A condition of capability needed by a user to solve a problem or achieve an objective; (2) A condition or a capability that must be met or possessed by a system ... to satisfy a contract, standard, specification, or other formally imposed document" [1]. Note that in software requirements we deal with the requirements of the proposed system, that is, the capabilities that the system, yet to be developed, should have.

As we have seen, most software development processes require requirements to be specified. The goal of the requirements activity is to produce the Software Requirements Specification (SRS), which describes *what* the proposed software should do without detailing *how* it will do it. In iterative development, which is now the preferred approach for software development, the list of requirements with a high level description may be the initial SRS, with details getting developed in each iteration for the requirements implemented in that iteration. New requirements may also emerge and be added after each iteration. In this chapter we will focus on developing the initial SRS for an application. The approaches discussed can be suitably adapted for modifying the SRS as new requirements emerge.

In this chapter we will discuss:

- The importance of requirements in an application development project and main activities in the process for developing the desired SRS.
- The desired characteristics of an SRS, the structure of an SRS document, and its key components, with an example SRS for an application.
- The concept of minimum viable product (MVP).
- The use case approach for analyzing and specifying functional requirements, and how use cases can be developed.
- How LLMs can be used effectively to help in requirements specification and validation.

P. Jalote, *A Concise Introduction to Software Engineering*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-74318-4_3

3.1 Requirement Process

The origin of most software systems lies in the needs of clients. These systems are created by developers, and ultimately will be used by the end users. Thus, there are three major parties interested in a new system: the client or sponsor, the developer, and the users. The requirements for the system which will satisfy the needs of the clients and the concerns of the users must be communicated to the developer. Often the client may not understand software or the software development process, and the developer may not understand the client's business and application area. This causes a communication gap between the parties involved in the development project. Specifying the requirements aims to bridge this communication gap so all parties share a common vision of the software being built.

The requirements process typically consists of three basic tasks: problem or requirement analysis, requirements specification, and requirements validation. The process is not linear; there is considerable overlap and feedback between these activities. The overall requirement process is shown in Fig. 3.1. As shown in the figure, the specification activity often requires reverting back to the analysis activity. This happens frequently because some parts of the problem are analyzed and then specified before others. Furthermore, the process of specification frequently reveals gaps in the understanding of the problem, necessitating further analysis. Once the specification is complete, it undergoes the validation activity. This activity may uncover issues in the specification itself, requiring a return to the specification activity, or reveal shortcomings in the understanding of the problem, requiring a return to the analysis activity.

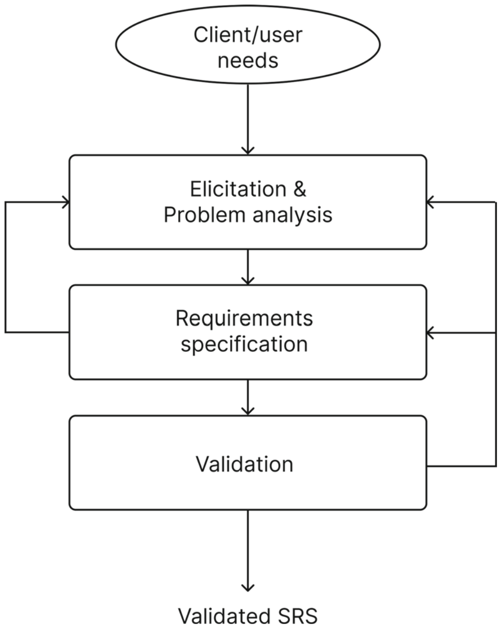


Fig. 3.1 The requirement process

3.1.1 Problem Analysis

Problem analysis often begin with a high-level "problem statement." The aim of is to gain a clear understanding of the needs of the clients and the users, what exactly is desired from the software, and the constraints on the solution. Frequently the client and the users do not fully understand or know all their needs because they may not fully appreciate the potential of the new system. Analysts must ensure that the real needs of the clients and the users are uncovered, even if they aren't clearly articulated. That is, the analysts don't just collect and organize information about the client's organization and its processes, but they also act as *consultants* who play an *active* role in helping clients and users identify their needs.

During analysis, the problem domain and environment are modeled to understand system behavior, constraints on the system, inputs, outputs, etc. The goal is to thoroughly understand what the software needs to provide. This involves a series of meetings between the analyst and clients and users. In the early meetings, the clients and end users explain their work, their environment, and their needs as they perceive them. The analyst is primarily listening and absorbing information. As understanding develops, the analyst seeks clarification in the subsequent meetings, documents information or build suitable models, and may brainstorm about what the system should do. In the final few meetings, the analyst essentially explains to the client what the system proposed should do and uses the meetings as a means of verifying if the proposed system meets the objectives.

For understanding the functional requirements, often the method used for specification (e.g. use cases) can help in analysis, by clarifying who expects what to be done by the application. We will discuss this further in the next section.

For modeling the data aspects (all applications work on data provided or collected), data modeling approaches such as Data flow diagrams (DFD), also called *data flow graphs*, may be used. A DFD shows the flow of data through a system. It views a system as a function that transforms the inputs into desired outputs through a set of transformations, and aims to capture these transformations that take place within a system to the input data so that eventually the output data is produced. In a DFD, data flows are identified by unique names that convey some meaning about what the data is. For specifying the precise structure of data flows, a *data dictionary* is often used. The associated data dictionary states precisely the structure of each data flow in the DFD.

Entity relationship (ER) diagrams are another approach that has been used for years for modeling the data aspects of a system. ER diagrams have two main concepts and notations to represent them. These are entities and relationships. Entities are a primary information holders or concepts, and are represented as boxes in the diagram. An entity is effectively a table in a database or a sheet in a spreadsheet, with each row representing an instance of this entity. Entities may have attributes, which are properties of the concept being represented, and can be viewed as the columns of the database table. Relationships between two entities are represented by a line connecting the boxes representing the entities. A relationship can also be given a name by labeling the line. An ER diagram specifies some properties of the relationship as well. The common ones include one-to-one (that one element of an entity is related to exactly one element of the other entity), one-to-many or many-to-one (that one element is related to many elements of the other entity), and many-to-many (that one element of entity A is related to many elements of entity B and one element of entity B is related to many elements of entity A). There are various notations to express the nature of relationships. Hence, the ER diagram, allows one to determine the initial logical structure of the tables easily.

An analyst may use models, or other approaches for developing a clear understanding of the application, enabling accurate requirement specification. Models aid the analyst in this understanding.

3.1.2 Specification

The understanding obtained from problem analysis forms the basis of *requirements specification*, where the focus is on clearly documenting the requirements. Issues such as representation, specification languages, and tools are addressed during this activity. Since analysis produces large amounts of information and knowledge with possible redundancies, organizing and describing the requirements properly is a key goal.

Much of this chapter focuses on specifying requirements, which we will discuss in more detail in subsequent sections.

3.1.3 Validation

It is crucial that the requirements specification accurately reflects the client's needs, as errors will lead to errors in the application. Furthermore, the longer an error remains undetected, the greater the cost of correcting it, making it extremely desirable to validate the requirements and detect any errors before its implementation. *Requirements validation* focuses on ensuring that the SRS is free of errors and is of high quality.

Requirements specification errors (besides clerical errors) can be classified in four types: omission, inconsistency, incorrect fact, and ambiguity. *Omission* is a common error in requirements. In omission, some user requirement is simply not included in the SRS; the omitted requirement may be related to the behavior of the system, its performance, constraints, or any other factor. Another common form of error in requirements is *inconsistency*. Inconsistency can be due to contradictions within the requirements themselves or to incompatibility of the stated requirements with the client's actual requirements or with the environment in which the system will operate. The third common requirement error is *incorrect fact*. Incorrect fact errors occur when some fact recorded in the SRS is incorrect. The fourth common error type is *ambiguity*. Errors of this type occur when there are some requirements that have multiple meanings, that is, their interpretation is not unique.

Error data from studies [2,3] are shown below—of over 250 and 80 errors found respectively, the percentage of different types of errors was:

Omission	Incorrect Fact	Inconsistency	Ambiguity
26%	10%	38%	26%
32%	49%	13%	5%

Though the distribution of errors is different in these two cases, they suggest that the major issues of omission, incorrect fact, inconsistency, and ambiguity are important in both. This implies that besides improving the quality of the SRS itself (e.g., no clerical errors, language is precise), the validation should focus on uncovering these types of errors.

Since requirements are generally textual that cannot be executed, requirements reviews are the most common method of validation.

Requirements review is a review by a group of people examining the SRS to find errors and point out other matters of concern. Because requirements specification specifies something that originally existed informally in people's minds, the requirements review team should include client and user representatives as well. Although the primary goal of the review process is to reveal any errors in the requirements, the review process is also used to consider factors affecting quality, such as testability and readability.

Requirements reviews are probably the most effective means for detecting requirement errors. The data in [3] about the A-7 project shows that about 33% of the total requirement errors were detected through reviews, and about 45% were found during the design phase, where the requirement document is used as a reference for design.

Though requirements reviews remain the most commonly used and viable means for validation, other possibilities arise if some special-purpose languages are used for requirements specification. In such situations, it is possible to have tools to verify some of the properties like consistency and correctness. However, as tools cannot generally check for completeness, reviews are still needed even if the formal notations or tools are used.

3.2 Requirements Specification

As mentioned, a primary purpose of the SRS is to bridge the communication gap between developers, sponsors and users, ensuring they share a common vision of the software application being built. Hence, one of the key advantages of a good SRS is that it establishes the basis for agreement between the client and the supplier on what the software will do. This basis for agreement also often implies that an SRS provides a reference for validation of the final product—it helps the client determine if the software meets the requirements.

Providing the basis of agreement and validation are strong reasons for both the client and the developer to have a high quality SRS. The quality of SRS also impacts the goal of achieving high quality and productivity. An error in the SRS will manifest itself as an error in the final system implementing the SRS. If we want a high-quality application with few errors, we must begin with a high-quality SRS. It is also known that the cost of fixing an error increases almost exponentially over time [4]. Therefore, improving the quality of requirements can result in significant future savings by reducing costly defect removals. In other words, a high-quality SRS lowers the development cost.

So, for a host of reasons, having a good quality SRS specification is highly desirable. Let's start the discussion with the desired characteristics of the SRS.

3.2.1 Desirable Characteristics of an SRS

To meet its goals, an SRS should have certain properties and include different types of requirements. Some desirable characteristics of an SRS are [1]:

1. Correct
2. Complete
3. Unambiguous
4. Verifiable
5. Consistent
6. Ranked for importance

An SRS is *correct* if every requirement represents something needed in the final system. It is *complete* if it specifies everything the software is supposed to do and the responses of the software to all classes of input data. In iterative development, completeness means that the requirements for the next iteration are complete. It is *unambiguous* if and only if every requirement stated has one and only one interpretation. Since requirements are often written in natural language, which is inherently ambiguous, the SRS writer has to be careful and ensure that there are no ambiguities.

An SRS is *verifiable* if and only if every stated requirement is verifiable. A requirement is verifiable if there exists some cost-effective process that can check whether the final software meets that requirement. It is *consistent* if no requirement conflicts with another. Terminology can cause inconsistencies; for example, different requirements may use different terms to refer to the same object. There may also be logical or temporal conflict between requirements that causes inconsistencies.

Generally, not all requirements are of equal importance. Some are critical, others are important but not critical, and there are some which are desirable but not very important. Similarly, some requirements are "core" requirements unlikely to change over time, while others are more dependent on time. Some provide more value to the users than others. An SRS is ranked for importance if each requirement's significance is indicated. This understanding of value that each requirement provides is essential for iterative development as it guides the selection of requirements for each iteration.

3.2.2 Functional and Non-functional Requirements

All applications have functional and some non-functional requirements, both of which must be understood and specified.

Functional requirements detail the expected behavior of the system—which outputs should be produced from the given inputs. These requirements outline the various functions the application must provide, specifying the relationship between input and output. Therefore, for each functional requirement, a description of all the data inputs and their sources, the units of measure, and the range of valid inputs should be clearly defined.

Similarly, all operations performed on the input data to produce the output must be specified. This includes validity checks on the input and output data, parameters affected by the operation, and equations or other logical operations that must be used to transform the inputs into corresponding outputs. For example, any formulas for computing outputs should be included. The functional requirement must also clearly state how the system should handle abnormal or exceptional situations like incorrect input. In short, the system behavior for all foreseen inputs and system states must be specified.

Besides performing the functions it is supposed to perform, a useful application must support other properties such as performance, security, privacy, user interface quality, etc. Such requirements for an application form its non-functional requirements. There can be many types of non-functional requirements, but the most common ones are about:

- Performance and other quality attributes
- Design constraints including security and privacy requirements
- External interfaces.

The *performance requirements* part of an SRS specifies the constraints on the software system performance. These may be capacity requirements like how many concurrent users or the load it should be able to handle. More commonly, they outline constraints on the execution behavior of the system, like the response time and throughput. All these requirements must be stated in measurable terms. Vague statements such as “response time should be good” or the system must be able to “process all the transactions quickly” are not acceptable because they are imprecise and not verifiable. Instead, statements like “the response time of command *x* should be less than one second 90% of the times” or “a transaction should be processed in less than one second 98% of the times” should be employed for clarity.

Performance is the quality attribute often emphasized in the non-functional requirements as acceptance of applications depends on it. However, there can be also requirements for other quality attributes such as usability, portability, reliability, portability, etc.

There are a number of factors in the client’s environment that may restrict the choices of a designer leading to *design constraints*. Such factors include standards that must be followed, resource limits, operating environment, reliability and security requirements, and policies that may have an impact on the design of the system. Some examples of these are:

Standards Compliance: This specifies the requirements for the standards the system must follow. The standards may include the report format and accounting procedures. There may be audit requirements which may require logging of operations.

Hardware Limitations: The software may have to operate on some existing or predetermined hardware, thus imposing restrictions on the design. Hardware limitations can include the restrictions imposed by existing or predetermined hardware; including the type of machines, operating system available on the system, supported languages, and limits on primary and secondary storage.

Tech Stack and Other Services: Though the choice of technology to be used should be a part of the design process, there are sometimes constraints on an application regarding what technologies it can use, often termed as the tech-stack. This specifies the languages, libraries and frameworks to be used for the front-end and for the back-end, and what database may be used (whether hosted on a server or a hosted service). There may also be requirements on technologies or specific services to be used for different functions of the application like authentication, payment, etc.

Reliability and Fault Tolerance: Fault tolerance requirements can place a major constraint on how the system is to be designed, as they make the system more complex and expensive. Recovery requirements are often an integral part here, detailing what the system should do if some failure occurs to ensure certain properties.

Security and Privacy: Security requirements are becoming increasingly important. These requirements place restrictions on authenticating users, provide different kinds of access for different people, require the use of passwords and cryptography techniques, and maintain a log of activities in the system, etc. They may also require proper assessment of security threats, proper programming techniques, and use of tools to detect flaws like buffer overflow.

All software interactions with people, hardware and other software should be clearly specified in the *external interface* specification part. The characteristics of each user interface of the software product should be specified. User interface is becoming increasingly important and must be given proper attention. A preliminary user manual may be created with all user commands, screen formats, an explanation of how the system will appear to the user, and feedback and error messages.

For hardware interface requirements, the SRS should specify the logical characteristics of each interface between the software product and the hardware components. If the software is to execute on existing hardware or predetermined hardware, all the hardware characteristics, including memory restrictions, should be specified. In addition, the current hardware's use and load characteristics should be given.

The interface requirement should also specify the interface with other software the system will use or that will use the system. This includes the interface with the operating system and other applications including the external services the application must use.

3.2.3 Structure of a Requirements Document

All requirements for a system should be specified in a clear and concise document. For this, it is essential to properly organize the SRS document. IEEE provides various ways of specifying requirements [1], understanding that no single method that suits all projects. Drawing from these structures, we propose a simple structure that is suitable for concisely specifying requirements for modest-sized projects when employing use cases (discussed in next section) for functional requirements specification. The general structure of an SRS is given in Fig. 3.2.

The title should be descriptive providing a quick vision of what the application is about. The sponsor is the organization/person for whom the application is being built. The introduction section contains the purpose and goals, scope, etc. of the application, as well as the categories of users which will use the application. The key aspect here is to clarify the motivation and business objectives that are driving the project and provide clarity on the different types of users who will use the application.

The next section provides a summary of the functional requirements of the application—when using use cases, it includes a table listing the use case for each category of user with brief descriptions of entry conditions (when the use case may be executed), and possible exceptions. In addition, the priority of each use case should be noted—to aid in prioritizing implementation of requirements in iterative development. This summary table gives a good overview of the application and its use. For agile methods, as well as situations where details can be obtained as needed, this may suffice for the initial requirements phase, as these approaches prefer to do the detailed requirements when the requirement is to be implemented.

The non-functional requirements section specifies all such requirements the application should support. For performance, both static (i.e. capacity related) and dynamic (e.g. response time) requirements may be mentioned. For interfaces, all the interfaces of the software: to people, other software, hardware, and other systems, should be mentioned. User interfaces are clearly a very important component, and the SRS may provide some general guidelines about the screen formats, provide contents/components of each screen/form etc. (in the detailed requirements section detailed screen layouts can be provided, if desired). Design constraints specify all the

1. Title - An expressive title for the application being built
2. Sponsor - organization names, contacts, etc
3. Introduction - purpose and basic objectives of the application,
   user categories, scope, definitions/acronyms being used, etc.
4. Functional Requirement Overview - the list of use-cases the application should
   provide for each user category.
5. Non-Functional Requirements - performance requirements, user interface requirements,
   design constraints, etc
6. Further Details - detailed description of use cases, of non-functional
   requirements

Fig. 3.2 A structure of an SRS

constraints imposed on design (e.g., security, fault tolerance, standards compliance, tech-stack, external services to be used, etc.).

The detailed requirements section describes the specifics of the requirements. Here, details of each use case can be provided. If detailed requirements are documented, this can be the largest part of the document. Often, detailed requirements may be done during the iteration when the requirement is implemented.

3.2.4 Minimum Viable Product

Minimum Viable Product(MVP) is a relatively new concept which has been promoted more for entrepreneurship and lean startups [5]. In projects or startups where a new product is conceived, the requirements may be decided by the “founders” or sponsors based on their idea of what the users want. In such situations there is tendency to envision fancier or advanced features that the users may not initially need, leading to requirement bloating. And for a new product, since the success depends on a host of factors, beyond just its features, it is crucial to minimize risk while still providing a useful product. This allows the sponsors to judge the market and the use by users, and enhance the product, if needed, based on feedback from actual users of the initial release.

In this dynamic and risky landscape of product development, the concept of MVP has emerged as a guiding philosophy, particularly embraced by lean startups seeking to innovate efficiently. The essence of MVP lies in the strategic introduction of a new product with basic features, designed to capture user attention and deliver immediate value. This approach recognizes the importance of adapting and refining a product based on real-world feedback, minimizing waste and maximizing the potential for success.

The MVP concept suggests quick launch of a new product by a startup utilizing minimum resources and time, keeping in mind that adding features can delay the product release and increase costs. The idea is first to build a working product quickly and then expand it based on user feedback. An MVP is minimum (has minimum features) and it is viable (can provide good value to the users with the features included). The initial release may be a full release or a beta for early adopters and testers. Feedback from initial users helps the team validate the product idea, its features, and get inputs on the functionality to be added in the future.

The minimum features in the MVP should be such that they provide value and demonstrate the future potential of an application. The product must also have a good interface (no one wants to use an application with poor user interface, even if it is a useful application), and its design should also be extensible to allow for future enhancements.

As mentioned earlier, an MVP is to facilitate a quick release, and is usually built and released in a few months to attract early users. It validates the product idea and assumptions about users with real-life data, which is used to build future releases. It also provides insight into user behaviour to shape the direction of product initiatives and marketing strategy. Hence, besides including the minimum features needed to provide value, an MVP must also collect data about application use which can be analyzed to get a sense of what users like what they don't like, and what may be missing.

An MVP minimizes the resources needed for software development, allowing resources to be spent on other activities of a venture. It supports the philosophy of failing fast—if the product is unviable or not needed, it is better to realise it early and move on.

An often-quoted example of the use of MVP is that of Airbnb. Airbnb aimed to create a new marketplace offering personal housing for rent on a peer-to-peer basis. This new idea had some critical assumptions: that some people would be willing to stay on a stranger's private property and pay for wifi, a bed, and breakfast, and that others would be willing to rent their properties for short duration to strangers. It was a risky venture as how well these assumptions hold and how users will respond to them were not known. They first built an MVP, which was a minimal website with photos and information, only offering their own apartment. They learned what the users wanted with the feedback from paying guests. There are many other examples of (now) big companies that use MVPs, e.g. Dropbox, Uber, Meta (Facebook), and Instagram.

Though MVP has been discussed in the context of new application launches, generally by startups or new companies, the idea of MVP is helpful in any application development where an iterative development approach is used (which is most common today). The MVP approach provides a way to prioritize requirements particularly when it is a new application or service is being developed for the users. Instead of loading the initial application itself with all the features users may need, the MVP concept can be used to start by including the important features that provide high value to the users and that provide good data to the developers about user behaviour and preferences. This can avoid feature bloating by not developing features that are thought to be helpful by designers but are not desired by the users.

If requirements specification contains prioritization of the requirements, deciding what to include in the MVP becomes easier—the MVP will implement the high priority requirements (and those that are needed to implement them) and some features that will provide useful data to analyze the use of features and envision additional features that may be desired. Future iterations can deliver newer features that will provide additional value or make the current features easier or better to use.

3.2.5 Requirements in Open Source Process

The requirements process discussed above is tailored for developing software applications from scratch. As discussed in the previous chapter, the open source development process typically starts with an existing application and the process is about iteratively enhancing the application, and often does not require an SRS document. However when starting a fresh application development project using an open source process, adapting the SRS to fit this process can be beneficial. We discuss one approach of how this can be done.

In the previous chapter we have discussed how to manage a project being executed using an open source platform like GitHub. The SRS can be put into the documents folder for the project. However, that does not help in assigning tasks and getting the project done. For that, the requirements will have to be listed as issues which can then be assigned and tracked. This can be done easily by adding each requirement for the upcoming milestone (or the next few milestones) as an issue labeled as a "feature". If use cases or user stories have been used for requirements specification, each one can be converted into an issue (or it may be broken into multiple issues each assigned separately, e.g., a use case may be divided into an issue for the front end and another issue for the back end).

After initial iterations, as new requirements emerge, they should be added as issues in the project. If desired, they may be also appended in the initial SRS to keep the document complete. In general, while initial SRS is crucial for starting and providing a planned start to the project, after a few iterations, its use becomes limited in an open source development process, as most work items are managed through issues, and the set of issues tagged as "features" effectively represents the evolving requirements.

3.3 Functional Specification with Use Cases

Functional requirements often form the core of a requirements document. The traditional approach to specify functionality is to detail each function that the system should provide. However, use cases specify the functionality of a system by describing the behavior of the system, captured as interactions between the users and the system. The discussion of use cases here is based on the concepts and processes discussed in [6].

3.3.1 Basics

A software application may be used by various users, or other systems. In use case terminology, an *actor* is a person or a system that uses the system to achieve some goal. An actor represents a group of users or a user category/type who behave in a similar manner. Different actors represent groups with different goals.

A *primary actor* is the main actor that initiates a use case (UC) to achieve a goal, and whose goal satisfaction is the main objective of the use case. However, the use case must also fulfill goals that other stakeholders might have for it. That is, the main goal of a use case is to describe behavior of the system that results in satisfaction of the goals of all the stakeholders, although the use case may be driven by the goals of the primary actor. For example, the use case "Withdraw money from the ATM" has a customer as its primary actor and will normally describe the entire interaction of the customer with the ATM. However, the bank is also a stakeholder of the ATM system and its goals may include that all steps are logged, money is given only if there are sufficient funds in the account, and no more than some amount is given at a time, etc. Satisfaction of these goals should also be shown by the above use case.

For describing interaction, use cases utilize scenarios. A *scenario* describes a sequence of steps performed to achieve a goal under specified conditions. Each step in a scenario is a logically complete action performed either by the actor or the system. Generally, a step is some action by the actor (e.g., enter information), some logical step that the system performs to progress toward achieving its goals (e.g., validate information, deliver information), or an internal state change by the system to satisfy some goals (e.g., log the transaction, update the record).

A use case has a *main success scenario*, which describes the interaction if nothing fails and all steps succeed. Though the use case aims to achieve its goals, different situations can arise while the system and the actor are interacting which may not permit the system to achieve the goal fully. For these situations, a use case has *extension scenarios* which describe the system behavior if some of the steps in the main scenario do not complete successfully. Sometimes they are also called *exception scenarios*. A use case is a collection of all the success and extension scenarios related to the goal. The terminology of use cases is summarized in Table 3.1.

It should be evident that the basic system model assumed by the use case is that a system primarily responds to requests from actors who use the system. By describing the interaction between actors and the system, the system behavior is specified, which in turn specifies its functionality. A key advantage of this approach is that use cases focus on external behavior, thereby cleanly avoiding internal design during requirements, something that is desired but is a challenge with many modeling approaches.

Use cases are naturally textual descriptions representing the behavioral requirements of the system. This behavior specification captures most of the functional requirements of the system.

Table 3.1 Use case terms

Term	Definition
Actor	A person or a system which uses the system being built for achieving some goal
Primary actor	The main actor for whom a use case is initiated and whose goal satisfaction is the main objective
Scenario	A set of actions that are performed to achieve a goal under specified conditions
Main success scenario	Describes the interaction if nothing fails and all steps succeed
Extension scenario	Describes the system behavior if some steps in the main scenario do not succeed

3.3.2 Examples

Let us illustrate these concepts with a few use cases, which we will also use to explain other related concepts. Let us consider that a small on-line auction system is to be built for a university community, called the University Auction System, through which different members of the university can sell and buy goods. We assume a separate financial subsystem for processing payments with each buyer and seller having account in it.

In this system, even though the same people might be buying and selling, we identify "buyers" and "sellers" as separate logical actors, because they have different goals. Besides these, the auction system itself and the financial system are also stakeholders and actors. Let us first consider the main use cases of this system—"put an item for auction," "make a bid," and "complete an auction." These use cases are given in Fig. 3.3.

The use cases are self-explanatory. This is the great value of use cases—they are natural and story-like which makes them easy to understand by both an analyst and a layman. This helps considerably in minimizing the communication gap between the developers and other stakeholders.

Some points about the use case are worth discussing. The use cases are generally numbered for reference purposes. The name of the use case specifies the goal of the primary actor (hence there is no separate line specifying the goal). The primary actor can be a person or a system—for UC1 and UC2, they are persons but for UC3, it is a system. The primary actor can also be another software requesting a service. The *precondition* of a use case specifies what the system must ensure before allowing the use case to be initiated. Common preconditions are "user is logged in," "input data exists in files or other data structures," etc. For an operation like delete it may be that "item exists," or for a tracking use case it may be that the "tracking number is valid."

It is worth noting that the use case description list contains some actions that are not necessarily tied to the goals of the primary actor. For example, the last step in UC2 is to update the bid price of other bidders. This action is clearly not needed by the current bidder for their goal. However, as the system and other bidders are also stakeholders for this use case, the use case must ensure that their goals are also satisfied. Similar is the case with the last item of UC1.

The exception situations are also fairly clear. We have listed only the most obvious ones. There can be many more, depending on the goals of the organization. For example, there could be one "user does not complete the transaction," which is a failure condition that can occur anywhere. What should be done in this case must then be specified (e.g., all the records are cleaned).

A use case can employ other use cases to perform some of its work. For example, in UC2 actions like "block the necessary funds" or "debit bidder's account and credit seller's" are actions that need to be performed for the use case to succeed. However, they are not performed in this use case, but are treated as use cases themselves whose behavior has to be described elsewhere. If these use cases are also part of the system being built, then they must be described in the requirements document. If they belong to another system, then proper specifications about them must be obtained.

- ***UC1***: **Put an item for auction**
  Primary Actor: Seller
  Precondition: Seller has logged in
  Main Success Scenario:

  1. Seller posts an item (its category, description, picture, etc.) for auction
  2. System shows past prices of similar items to seller
  3. Seller specifies the starting bid price and a date when auction will close
  4. System accepts the item and posts it

  Exception Scenarios:

  – 2 a) There are no past items of this category
    · System informs the seller this situation

- ***UC2***: **Make a bid**
  Primary Actor: Buyer
  Precondition: The buyer has logged in
  Main Success Scenario:

  1. Buyer searches or browses and selects an item
  2. System shows the rating of the seller, the starting bid, the current bids, and the highest bid; asks buyer to make a bid
  3. Buyer specifies a bid price
  4. System accepts the bid; Blocks funds in bidders account
  5. System updates the max bid price, informs other users, and updates the records for the item

  Exception Scenarios:

  – 3 a) The bid price is lower than the current highest bid
    · System informs the bidder and asks then to rebid
  – 4 a) The bidder does not have enough funds in their account
    · System cancels the bid, asks the user to add more funds

- ***UC3***: **Complete auction of an item**
  Primary Actor: Auction System
  Precondition: The last date for bidding has been reached
  Main Success Scenario:

  1. Select highest bidder; send an email to selected bidder and seller informing them of final bid price; send an email to other bidders as well
  2. Debit the bidder's account and credit the seller's
  3. Unblock all other bidders funds
  4. Transfer from seller's acct. commission amt. to organization's acct.
  5. Remove item from the site; update records

  Exception Scenarios: None

Fig. 3.3 Main use cases in an auction system

The financial actions may easily be outside the scope of the auction system, so need not be described in the SRS. However, actions like "search" and "browse" are most likely part of this system and must be described in the SRS.

3.3.3 Developing Use Cases

UCs can be developed in a stepwise refinement manner with each step adding more details. This approach allows UCs to be presented at various levels of abstraction. In the initial stages, UCs can be used for requirements elicitation and problem analysis, as they are easy to use for discussions with sponsors and users. Though any number of levels of abstraction are possible, four natural levels emerge:

- **Actors and goals.** The actor-goal list enumerates the use cases and specifies the actors for each goal. (The name of the use case generally represents the goal.) This table may be extended by giving a brief description of each use case. At this level, the use cases together specify the scope of the system and give an overall view of what it does. Completeness of functionality can be assessed fairly well by reviewing these.
- **Main success scenarios.** For each of the use cases, the main success scenarios are provided at this level. With the main scenarios, the system behavior for each use case is specified. Reviewing them ensures that interests of all the stakeholders are met and that the use case delivers the desired behavior.
- **Failure conditions.** Once the main success scenarios are listed, all the possible failure conditions can be identified. At this level, for each step in the main success scenario, the different ways in which a step can fail form the failure conditions. Before deciding what should be done in these failure conditions (which is done at the next level), it is better to enumerate the failure conditions and review for completeness.
- **Failure handling.** This is perhaps the most tricky and difficult part of writing a use case. Often the focus is so much on the main functionality that people do not pay attention to how failures should be handled. Determining what should be the behavior under different failure conditions will often identify new business rules or new actors.

These levels serve different purposes. Actor and goal-level description is very useful for discussion on overall functionality or capabilities of the system. Failure conditions, on the other hand, help understand and extract detailed requirements and business rules under special cases.

These four levels can also guide the analysis and specification activity when using use cases. First actors (user categories) and their goals should be identified and validated. This can be followed by specifying the steps in the success scenarios for each use case. The success scenarios will highlight possible failure conditions, which can be enumerated and handled.

What should be the level of detail in a use case? There is no one answer to a question like this; the answer always depends on the project and the situation. Generally it is good to have sufficient details which are not overwhelming but are sufficient to build the system and meet its quality goals. For example, if there is a small collocated team building the system, it is quite likely that use cases which list the main exception conditions and give a few key steps for the scenarios will suffice. On the other hand, for a project whose development is to be subcontracted to some other organization, it is better to have more detailed use cases.

When writing use cases, apply general technical writing rules. Use simple grammar, clearly specify who is performing the step, and keep the overall scenario as simple as possible. For simplicity, combine steps logically where it makes sense, such as combining "user enters his name," "user enters his SSN," and "user enters his address" into one step "user enters personal information."

3.3.4 User Stories

User stories are a common way of specifying functional requirements in agile development [7]. They are similar to use cases (UC) in that they also specify operational requirements from a user perspective and the goals the user wants to achieve from the application. So, while they are mentioned as a separate method for specifying functionality in agile developments, they can be viewed as a form of use case. It is a lightweight approach for capturing the who, what and why of a feature/functionality/requirement. Each user story can be viewed as a feature or functionality needed by the system and can be assigned to a programmer in the team. Though used often in agile development, they can be used in any iterative development approach.

A user story typically follows the format: As a [type of user], I want [some action] so that [some reason/benefit]. There are three parts to a user story:

- Type of user: Identifies the specific type of user or stakeholder of the system. (Actor in UC)
- Action: Describes the desired action or functionality (main success scenario UC)
- Benefit: Explains the reason or benefit behind the user's goal. (Goal in UC)

A user story succinctly captures a functional requirement of a user. Often, a user story may be stated on a card. This specification may also contain, besides the user story, a number and id/name for the story, and it may also specify the acceptance criteria for the user story. As we can see, a user story is similar to a use case in many ways, though it does not specify the detailed steps during the execution of the functionality. Hence, we consider them as a variation of use cases. Some examples of user stories are:

- For an e-commerce application: As a *customer*, I want a *shopping cart* so *I can combine all my proposed purchases*
- For an auction system: (1) As a *seller*, I want to *put an item for auction* so that *I can sell it to the highest bidder*. (2) As a *buyer*, I want to *search for an item and make a bid* so *I can purchase it*
- For an application for conducting quizzes in a lecture: (1) As an *instructor*, I want to *start a timed quiz of different types* so *I can give the exercise to students*. (2) As a *student*, I want to *participate in the quiz*, so *I can give my replies*.
- An application for supporting environment projects: (1) As a *project coordinator*, I want to *float a new environment project* so *I can have volunteers join the project*. (2) As a *volunteer*, I *want to join a project*, so *I can contribute as a volunteer*.

When the user categories are known, then for each user category, the functionality the user expects from the application can be stated as user stories and suitably prioritized—much the same way as discussed above for use cases.

User stories, as they are brief, will need detailing before implementing them—detailing can be done in the iteration in which the user story is to be developed through interaction with the users and developers (called communication in user stories). When detailing, acceptance criteria are often added to the story, which can be used to test the implementation. For example, during the elaboration of the first user story for an auction system, the following criteria/details may get added: (1) When I choose to list an item for auction, I should be prompted to provide details such as category, description, and a picture of the item. (2) When I enter the item details, the system should display past prices of similar items to assist me in setting a competitive starting bid price. (3) When I specify the starting bid price and choose a closing date for the auction, the system should validate the information and accept the item for auction. (4) When I submit the item, the system should post it, making it available for potential buyers to place bids.

User stories will evolve throughout the application development—each iteration will implement some user stories, and new functional requirements may emerge, which will be stated as user stories.

3.4 An Example

We give here key portions of the requirements specification for a case study example application. This application has been fully developed by a group of advanced computer science students, and has been tested and deployed. We will use this case study in some other chapters as well. While in the book we will discuss only some aspects, details about this case study are given on the website for the book, including its code, test scripts, design document, etc. For functional requirements, we will give here the summary of the use cases—details of the use cases can be found on the website.

SRS for Student Clubs Event Management Platform

Problem Background

The primary purpose of this platform is to facilitate student club coordinators to post requests for scheduling events within an academic institute, which are subject to approval by an events coordinator. Once approved, the event is added to the platform allowing students to register. Additionally, the platform provides dedicated pages for clubs, which are managed by the respective club's coordinators.

Stakeholders/Users

The primary users of the platform are: (i) students of the university, (ii) coordinators of various clubs, (iii) Admin for the platform.

Functional Requirements

The use cases for the different user groups are listed below (Fig. 3.4). (Details of the use cases are given in the full SRS on the website.)

Performance Requirements

1. The system shall be designed to support up to 1000 concurrent users without degradation in performance, ensuring a response time of less than 300 ms for 95% of requests under this load.
2. The platform should be responsive and have an average response time of 300 ms or less under normal load.

Design Constraints

1. The platform should use users' institute email address for login and authentication.
2. The tech stack should be free and open source.

External Interfaces

- The platform should be compatible with all major browsers (Safari, Chrome, Firefox, Brave, Microsoft Edge, etc.)
- The platform should be able to send email notifications to students and other users (optional).

Name of the Use Case	Actor	Purpose	Precondition
Explore Upcoming Events (essential)	Students	For users to explore upcoming events organised by various clubs	Student logged in
Explore Clubs (essential)	Students	For users to explore and obtain information about the various registered clubs	Student logged in
Log in (essential)	Students	To log in the platform	None
Register for Events (essential)	Students	To register for events	Student logged in
View the Profile (desirable)	Students	Allow students to view and edit their profile	Student logged in
Propose a new club (essential)	Students	Allow students to propose a new club	Student logged in
Propose an Event (essential)	Club Coordinator	To propose a new event	Club coordinator logged in
Edit an event's information	Club Coordinator	Edit the event's information	Club coordinator logged in
Approve an event (essential)	Admin	Approve an event	Event has been created; coordinator logged
Approve a new club (essential)	Admin	Approve	Club created; admin logged in
Change the events coordinator (optional)	Admin	Change the coordinator or his/her info	Coordinator is logged in

Fig. 3.4 Summary of use cases in the students event management application

Security/Privacy Requirements

- Only authorised users should be able to execute the different use cases; the landing page is visible to all.
- The platform shall implement security measures including but not limited to input validation and output encoding (Optional).

3.5 Using LLMs for Requirements

We have discussed the overall requirements analysis and specification process. Given the power of LLMs, it seems natural to leverage them to produce the desired SRS for a project. LLMs can also be used to evaluate SRS for various properties. Prompting approaches suitable for software tasks have been discussed earlier in Chap. 1. Here, we will briefly discuss some good prompting approaches for effectively using LLMs for developing requirements and for validating them. This is based on our experience [8] as well as some published works on this topic (e.g. [9]).

Generating the SRS

An LLM does not work in vacuum—it has to be given sufficient background information or context to help it produce a useful output. Additionally, what is desired from it has to be specified clearly through prompts—the more clearly stated the prompt is, the better the result will likely be [9]. For SRS development, we must decide on what background information to provide, and what command to give for the expected output.

Any application development project will start with having the title of the project, which itself gives some information about what the project is about. For understanding the requirements, initial discussions with the sponsors and/or users about the purpose, scope, key functionalities, etc., for the application provide some information for these. Some constraints also become evident. In other words, during the initial discussions with the sponsors, a fair amount of information is gathered and may be compiled as points and notes. Though this information is likely to be incomplete (from the requirements perspective), it is the starting point of developing the SRS. When using an LLM, this initial information compiled about the application can provide context to LLMs to help develop the SRS.

In addition, as we want the SRS to be in a specific format, we also need to provide the desired SRS structure (i.e. the output format). With these two contextual inputs, an LLM like ChatGPT can be asked to "produce an SRS for this application in the provided format."

We can use the one-prompt approach, in which we provide the background information as well as the command to generate the full SRS in one prompt. Generating the SRS in one go is likely to result in the individual sections being shorter and lacking in detail. Experience suggests that it may be better to take help from ChatGPT to generate portions of the SRS separately, and then combine them. That is, using the prompt chaining technique by dividing the task of SRS creation into sub tasks to develop the different sections of the SRS. These allow the LLM to provide more detailed outputs in each section, which can be combined with subsequent prompts.

For example, a prompt can be of the following form (based on the experiments reported in [8]):

> Prompt: I have to generate a software requirement specification, SRS, document for an application for <Task>. The SRS has to have the following structure: <the structure of the SRS document>.
>
> Please generate the user categories and the table of use cases for this application.

In the next prompt, it can be asked to generate the non-functional requirements for this application—in much the same manner as above, except that the context need not be provided as it is already there from earlier replies. Depending on the output, if further details of the use cases or other requirements are desired, the LLM can be prompted to do so—it will be better if the details being sought are clearly explained to the LLM. After generating all the required components, the sections can be merged suitably (or the LLM may be asked to consolidate them to create a unified SRS document.)

A variation of this approach can also be used using chain of thought (CoT) and role prompting. After giving the context, the LLM is prompted to methodically follow through with the SRS-creation procedure. We can prompt the LLM to do something like:

> Think step by step for this problem. Identify the user categories for the application, identify their needs, create requirements, and draft an SRS. Think like an experienced product manager during this.

Sometimes the LLM may produce a generic type of document and fail to tailor requirements for the specific project. The non-deterministic generation of LLMs also significantly impacts the generation quality, and multiple runs with the same prompt configuration could provide vastly different results. Using self-consistency prompting, which involves re-running prompts to reduce the impact of the probabilistic generation procedure with LLMs, can help here.

SRS Validation

Once the SRS document has been produced, it should be evaluated for some key properties of correctness, completeness, consistency, and unambiguity—as discussed earlier in the chapter. LLMs can be used for this task also.

There are two ways of evaluating the SRS with the help of an LLM. We can either prompt the LLM to evaluate the SRS on all the parameters in one prompt, or have it evaluate the SRS on one parameter at a time using prompt chaining. Given that each parameter is quite different, it is best to do it one at a time for the most coherent response. For example, it can be asked to check for correctness, then for consistency, and then for unambiguity, etc. Care must be taken to provide the definition of the parameter being evaluated to make sure that the LLM only responds for that particular parameter.

For validation, we can group the attributes into two categories—those that can be evaluated on a requirement-by-requirement basis, and those that depend on the entire SRS and the context outside the document. In the former are attributes like ambiguity, correctness, verifiability. Completeness and consistency would be examples of the latter.

For per requirement quality attributes, based on experiments [8], some learnings are:

- Providing examples of quality attributes (i.e. one or few shot prompting) generally helps the LLM do a better job of evaluating the parameter and identify the problems more accurately.

- Chain-of-thought is also quite helpful. Asking the LLM to think step-by-step generally gives better performance.
- Besides identifying errors, we also want to remove them. For this we can ask the LLM to suggest improvements or suggest alternatives for the requirement. Experience suggests that the latter approach results is better and more precise results—the former can end up generating general guidelines.

With these, for validating an SRS, the prompt can contain: information about the task, background about the SRS, definition of the attribute, and some examples—in each example giving the requirements with the error and the corrected requirements. An example prompt for checking an SRS for ambiguity is:

> Prompt: I have a software requirement specification for a club event management system. Some background about the system: .... The following is the definition of unambiguousness: A requirement is unambiguous if only one possible interpretation exists.
> Eg1. Ambiguous: ... text....
> Unambiguous: ... text....
> Eg2. Ambiguous: ... text....
> Unambiguous: ... text....
> Here is the SRS: .... SRS of the system ....
> Please determine which requirements are ambiguous.
> Think step by step.

It may be better to ask it to identify ambiguous requirements first in functional and then in non-functional requirements. We can then ask it to tabulate the findings in a table. Following this, we can now ask it to fix the ambiguity errors for some/all of the identified errors.

For verifying a property like inconsistency, since inconsistency is a document-wide issue, asking the LLM to consider the whole SRS at once will help identify more points of inconsistency. We can explicitly prompt the LLM to gauge all sections of the SRS to check for inconsistency. Experiments suggest that providing examples does not help the LLM much in identifying these types of errors. A zero shot prompting can be appropriate here. The LLM can be asked to "consider the whole SRS together". Once it generates the inconsistencies, it can be asked to summarise it in a table. Experiments indicate that LLM is able to identify many inconsistencies.

Checking for completeness is a different challenge as compared to the rest of the SRS attributes as completeness depends on the context of the application and the needs of the user. When asking for missing requirements, it is best to ask the LLM to think like a stakeholder involved in the project and ask it to identify missing requirements. Zero shot prompting is sufficient here. The prompt may contain the background information and the task, as follows:

> Background information:
> I have a SRS for a club management system. Some info: ...
> An SRS is complete if all the requirements have been specified.
> Here is the SRS: ... SRS...
> Task: Find out the missing requirements for this SRS. Think step by step, acting like each of the stakeholders

Once the LLM does the analysis, we can prompt it to generate additional functional and non functional requirements.

3.6 Summary

- The main goal of the requirements process is to produce the software requirements specification (SRS) which accurately captures the client's requirements and which forms the basis of software development and validation.
- There are three basic activities in the requirements process—problem analysis, specification, and validation. The goal of analysis is to understand the different aspects of the problem, its context, and how it fits within the client's organization. In requirements specification the understood problem is specified or written, producing the SRS. Requirements validation is done to ensure that the requirements specified in the SRS do not have errors. Omission, incorrect fact, inconsistency, and ambiguity are the most common errors in an SRS. For validation, the most commonly used method is doing a structured group review of the requirements.
- The key desirable characteristics of an SRS are: correctness, completeness, consistency, unambiguousness, verifiability, and ranked for importance.
- A good SRS should specify all the functions the software needs to support, performance requirements of the system, the design constraints that exist, and all the external interfaces.
- Minimum viable product (MVP) is a useful concept to keep in mind while developing requirements. For a new application which a start-up may be depending on has many risks regarding user acceptance, what users really want, time to market, etc. In such a situation, it is best to build an MVP in the first release, which only implements those requirements that are needed for providing essential value to the users, and those which can help gain an understanding about what may be needed in the future. Based on data from the use of the MVP in the market, features for future releases can be selected.
- Use cases are a popular approach for specifying functional requirements. Each use case specifies the interaction of the system with the primary actor, who initiates the use case for achieving some goal. A use case has a precondition, a normal scenario, as well as many exceptional scenarios, thereby providing the complete behavior of the system. For developing use cases, first the actors and goal should be identified, then the main success scenarios, then the failure conditions, and finally the failure handling.

- LLMs can help in the requirements development as well as validating the requirements. Given the nature of the requirements development, prompt chaining is a suitable approach for specifying requirements, while providing proper context and establishing expectations in the initial prompts. For requirements validation, it is desirable to ask the LLM to verify for each of the desired property separately.

Self-assessment Exercises

1. What are the main components of an SRS? And what are the main criteria for evaluating the quality of an SRS?
2. What are the main error types for requirements, and what are some methods of detecting them?
3. What is an MVP for an application. When selecting from the requirements what features to implement in an MVP, briefly describe two criteria that you will use for including the requirements.
4. You have to develop a web-based application for giving multiple-choice in-class quizzes. The instructor starts a quiz and shares the URL. Students then can submit their responses in the form/screen shown to them. After some time, the instructor ends the quiz, after which students cannot submit their responses. After submitting, students can ask to see the summary of the responses. List the main use cases for this application.
5. Take an on-line social networking site of your choice. List the major use cases for this system, along with the goals, preconditions, and exception scenarios.
6. Do the same exercise for a conference management site which allows authors to submit papers, program chairs to assign reviewers and do the final paper selection based on reviews, and reviewers to enter the review.
7. Write user stories instead of use cases for these projects.
8. For the above examples (or any other), try generating requirements using an LLM by giving it a general description of the application, and the structure of the requirements document. Try a single prompt approach, and a prompt chaining approach in which different parts of the requirements are prompted for separately. See how the two requirements compare.
9. Take some SRS and remove some requirements, and add some inconsistencies in it. Use an LLM to find the errors and see how many of the injected errors it is able to find, how many it misses, and how many other errors it finds.

References

1. IEEE, IEEE recommended practice for software requirements specifications. Technical Report (1998)
2. J.S. Davis, Identification of errors in software requirements through use of automated requirements tools. Inf. Softw. Technol. **31**(9), 472–476 (1989)
3. V.R. Basili, D.M. Weiss, Evaluation of a software requirements document by analysis of change data, in *5th International Conference on Software Engineering* (IEEE, 1981), pp. 314–323
4. B.W. Boehm, *Software Engineering Economics* (Prentice Hall, Englewood Cliffs, NJ, 1981)
5. E. Ries, *The Lean Startup* (Portfolio Penguin, 2011)
6. A. Cockburn, *Writing Effective Use Cases* (Addison-Wesley, 2001)

7. G. Lucassen, F. Dalpiaz, J. M.E. Van Der Werf, S. Brinkkemper, Forging high-quality user stories: towards a discipline for agile requirements, in *2015 IEEE 23rd International Requirements Engineering Conference (RE)* (IEEE, 2015), pp. 126–135
8. M. Krishna, B. Gaur, A. Verma, P. Jalote, Using LLMs in software requirements specifications: an empirical evaluation, in *To appear in Proceedings of RE2024* (IEEE, 2024)
9. C. Arora, J. Grundy, M. Abdelrazek, Advancing requirements engineering through generative AI: assessing the role of LLMs (2023). arXiv:2310.13976

Software Architecture

4

Any complex system is composed of subsystems that interact under the control of system design to provide the expected behavior. When designing such a system, therefore, the logical approach is to identify the sub-systems, their interfaces, and the rules for their interaction. This is what software architecture aims to do.

As the software applications become more distributed and more complex, architecture becomes a crucial step in building the application. With a wide range of options now available for how an application may be configured and connected, carefully designing the architecture becomes very important. It is during the architecture design phase that choices about middleware, back-end databases, servers and security components are made. Architecture is also the earliest stage where properties like reliability, performance and scalability can be evaluated.

In this chapter, we will discuss:

- The key roles architecture plays in a software project, and the architectural views to specify different structural aspects of the system.
- The component and connector architecture of a system, and how it can be expressed.
- Different architectural styles for component and connector view to design the architecture of an application.
- Different architecture decisions that need to be taken and how the architecture of an application should be documented and evaluated.
- How LLMs can be used to help in architecture design decisions.

P. Jalote, *A Concise Introduction to Software Engineering*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-74318-4_4

4.1 Architecture Views

What is architecture? Generally, the architecture of a system provides a very high-level view of the parts of the system and how they are related to form the whole system. That is, architecture partitions the system into logical parts such that each part can be comprehended independently, and then describes the system in terms of these parts and their relationship.

A widely accepted definition of software architecture is that *the software architecture of a system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them* [1]. This definition implies that we only focus on those abstractions that specify the properties that other elements can assume to exist and are needed to establish relationships, not on the details of how these properties are supported. This is an important capability that allows architecture descriptions to represent complex systems in a succinct form that can be easily comprehended.

An architecture description therefore describes the different structures of the system. Some important roles of software architecture descriptions play are [1–3]:

1. *Understanding and communication*. An architecture description primarily communicates the architecture to various stakeholders, including the users who will use the system, the clients who commissioned the system, the builders who will build the system, and, of course, the architects. It helps the stakeholders gain an understanding of the macro properties of the system and how it intends to fulfill the functional and quality requirements. As the description provides a common language between stakeholders, it also becomes the vehicle for negotiation and agreement among the stakeholders, who may have conflicting goals.
2. *Reuse*. If one wants to build a software product in which existing components may be reused, the architecture must be designed in a manner such that the components which have to be reused can fit properly with newly developed components—it is very hard to achieve this type of reuse later if the architecture is not suitably designed.
3. *Construction and Evolution*. Architecture partitions the system into relatively independent parts. These parts can naturally be used for constructing the system, which also requires that the system be broken into parts such that different teams (or individuals) can separately work on different parts.
4. *Analysis*. An architecture enables the analysis or prediction of properties like reliability, performance, scalability, etc. of the application. Such an analysis in early stages of an application development helps determine whether the system will meet quality and performance requirements, and identify necessary adjustments.

Different views of the application architecture facilitate various types of analyses or focus on specific aspects. A view represents the system as composed of certain *elements* and *relationships* between them. Which elements are used by a view, depend on what the view wants to highlight. Many types of views have been proposed, most important being these three types [1,2]:

- Module
- Component and connector
- Allocation.

In a module view, the system is viewed as a collection of code units, each implementing some part of the system functionality. The main elements in this view are modules. These views are code-based and do not explicitly represent any runtime structure of the system. An examples of modules is packages, which are essentially related code being provided as a component to applications.

In a component and connector (C&C) view, the system is viewed as a collection of runtime entities called components. A component is a unit which has an identity in the executing system. Examples of components include objects, a collection of objects, and processes. While executing, components interact with others to support the system services. Connectors provide means for this interaction. Examples of connectors are pipes, sockets, http, remote procedure call, middleware, shared data, etc.

An allocation view focuses on how the different software units are allocated to resources like the hardware, file systems, and people. It specifies the relationship between software elements and elements of the environments where the software system is executed. They expose structural properties like which processes run on which processor, and how system files are organized on a file system.

Note that these different views are not unrelated. They all represent the same application and have relationships between elements in different views. These relationships may be simple or may be complex. For example, the relationship between components and modules may be one to many where one component is implemented by multiple modules. On the other hand, it may be complex where a module is used by multiple components. The designers must be aware of these relationships when creating different views.

Of these views, the C&C view has become the de facto primary view, almost always prepared when an architecture is designed (some definitions even view architecture only in terms of C&C views). In the rest of the chapter, we will focus primarily on the C&C view.

4.2 Component and Connector Architecture

The C&C architecture view of a system has two main elements—components and connectors. Components are usually computational elements or data stores present during system execution. Connectors define the means of interaction between these components. A C&C view defines the components and specifies which components are connected and through which connector. It describes a runtime structure of the system—what components exist during execution and how they interact.

The C&C view is the most common view of architecture, often represented by box-and-line diagrams. Most discussions about the architecture, refer to the C&C view and most architecture description languages focus on it as well.

4.2.1 Components

Components are units of computation or data stores in the system. Each component has a name that is generally chosen to represent its role or function, providing it a unique identity, necessary for referencing details in supporting documents, as a C&C diagram will only show component names.

A component is of a component type, where the type represents a generic component, defining the general computation and the interfaces a component of that type must have. Note that though a component has a type, in the C&C architecture view, we refer to actual instances and not types. Examples include clients, servers, filters, etc. Different domains may have other types like controllers, actuators, and sensors (for a control system domain).

In a C&C diagram, it is desirable to have a different representation for different component types, so the different types can be identified visually. While rectangular boxes are often used for all components, using distinct symbols for each type is more effective. This approach helps readers identify component types without needing additional descriptions.

Some of the common symbols used for representing commonly found component types are shown in Fig. 4.1.

4.2.2 Connectors

The different components of a system are likely to interact while the application is in operation to provide the expected services. After all, components exist to provide parts of the services and features, and these must be combined to deliver the overall application functionality. For composing an application from its components, information about the interaction between components is necessary.

Interaction between components may be through a simple means supported by the underlying process execution infrastructure of the operating system. For example, a component may interact with another using the procedure call mechanism (a connector), which is provided by the runtime environment for the programming language. However, the interaction may involve more complex mechanisms as well. Examples of such mechanisms are remote procedure call, TCP/IP ports, and a protocol like HTTP. These mechanisms require a fair amount of underlying runtime infrastructure, as well as special programming within the components to use the infrastructure. Consequently, it is extremely important to identify and explicitly represent these connectors. Specification of connectors help identify the suitable infrastructure needed to implement an architecture, as well as clarify the programming needs for components using them.

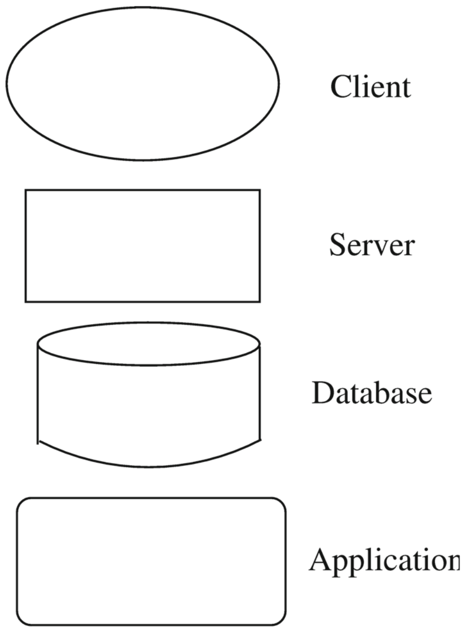


Fig. 4.1 Component examples

Note that connectors need not be binary and may provide an n-way communication between multiple components. For example, a broadcast bus may be used as a connector, which allows a component to broadcast its message to all others. Some of the common symbols used for representing various connector types are shown in Fig. 4.2.

A connector also has a name describing the nature of interaction it supports and type, which is a generic description of the interaction, specifying properties like whether it is a binary or n-way, types of interfaces it supports, etc.

It is worth pointing out that the implementation of a connector may be quite complex. However, the connectors are provided by the underlying system or as libraries, which components use as per the connector specifications. If, however, the underlying system does not provide a connector used in the architecture, it must be implemented as part of the project. Generally, while creating an architecture, it is wise to use connectors available on the systems where the software will be deployed.

4.2.3 An Example

Consider designing and building a simple application for taking an on-line survey of students on a campus. with a set of multiple-choice questions. The system will provide the survey form to the student, who can fill and submit it online. We also want that when the user submits the form, they are also shown the current result of the survey, that is, what percentage of students so far have filled which options for the different questions.

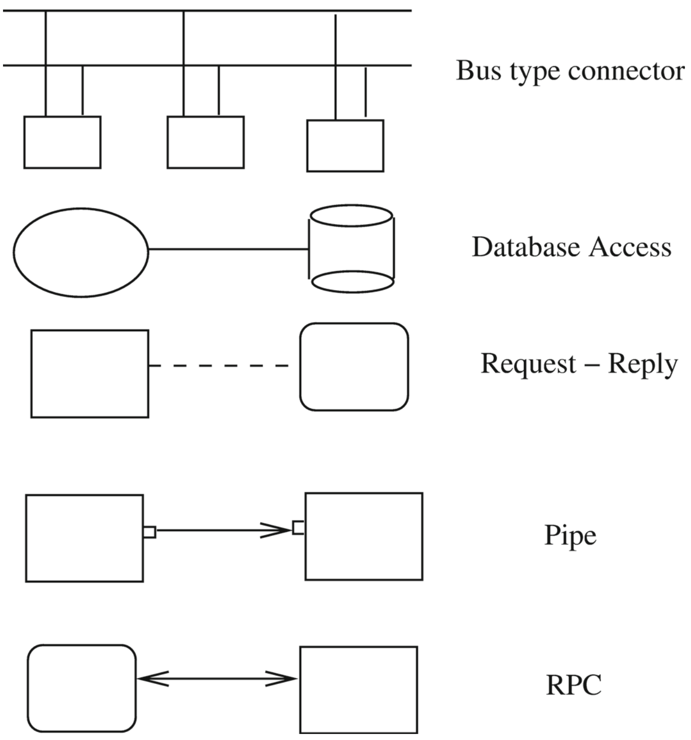


Fig. 4.2 Connector examples

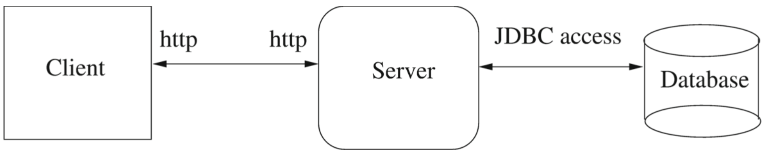


Fig. 4.3 Architecture of the survey system

This application is best built as a web application. For this simple application, a traditional 3-tier architecture is proposed. Consisting of a client, server, and database, the client displays the form and results, the server processes the submitted data and saves it to the database, which stores that data. The server also queries the database to get the survey results and sends them in proper format (HTML) back to the client, which then displays the result. The C&C view of this architecture is shown in Fig. 4.3.

Note that the client, server, and database are shown as different types of components. Note also that the connectors between the components are also of different types. The diagram makes the different types clear, making itself stand alone and easy to comprehend. Also note that at the architecture level, further details, such as the URL of the survey set, the modules or language used are unnecessary.

Note also that the connector between the client and the server explicitly specifies that HTTP is to be used. And the diagram also says that it is a Web client. This implies the necessity of a web browser running on the machines from which the student will take the survey. Having HTTP as the connector also means that there must be a proper HTTP server running, with the system's server suitably attached to it to allow access by clients. In other words, the entire infrastructure of browser and the HTTP server, for the purposes of this application, primarily provides the connector between the client and the server (and a virtual machine to run the client of the application).

The initial architecture has no security and a student can take the survey as many times as he wishes. Furthermore, even a non-student can take the survey. Now suppose the Dean of Students wants that this system be open only to registered students, and that each student is allowed to take the survey at most once. To identify the students, it was explained that each student has an account, and their account information is available from the main server of the institute.

The architecture needs significant modifications. The proposed architecture introduces a separate login form for the user, and a separate server component which does the validation. For validation, it queries the proxy for checking the provided login and password. If valid, the server returns a 'cookie' to the client (which stores it as per the cookie protocol). When the student completes the survey form, the cookie information validates the user, and the server checks if this student has already completed the survey. The architecture for this system is shown in Fig. 4.4.

Note that even though the connection between the client and the server is that of HTTP, it is somewhat different from the connection in the earlier architecture. In the first architecture, plain HTTP is sufficient. In this modified architecture, as cookies are also needed, the connector is really HTTP + cookies. If the user disables cookies, the required connector is not available and this architecture will not work.

Now suppose we want the system to be extended in a differently. It was found that the database server is somewhat unreliable, and is frequently down. Additionally, when the student is given the result of the survey upon submitting the form, slightly

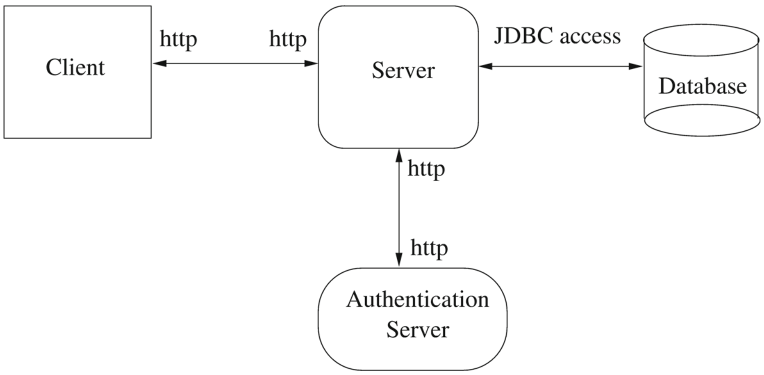


Fig. 4.4 Architecture for the survey system with authentication

outdated results are acceptable, as they are statistical data and a little inaccuracy is permissible. We assume that the survey result can be outdated by about 5 data points (i.e., even if it does not include data of 5 surveys, it is OK). The Dean wanted to enhance the system's reliability, and provide some facility for completing the survey even when the database is down.

To make the system more reliable, the following strategy was proposed. When the student submits the survey, the server interacts with the database as before. The results of the survey, however, are also stored in the cache by the server. If the database is down or unavailable, the survey data is stored locally in a cache component, and the cached results are used to provide feedback to the student. (This can be done for up to 5 requests, after which the survey cannot be completed.) Consequently, we have another component in the server called the cache manager. And there is a connection between the server and this new component of the call/return type. This architecture is shown in Fig. 4.5. A variation of this is to have the cache manager component between the server and the database.

Using the cache clearly improves the availability of the system. The cache also has an impact on performance. These extensions demonstrate how architecture affects both availability and performance, and how properly selecting or tuning the architecture can help meet the quality goals (or just improve the quality of the system).

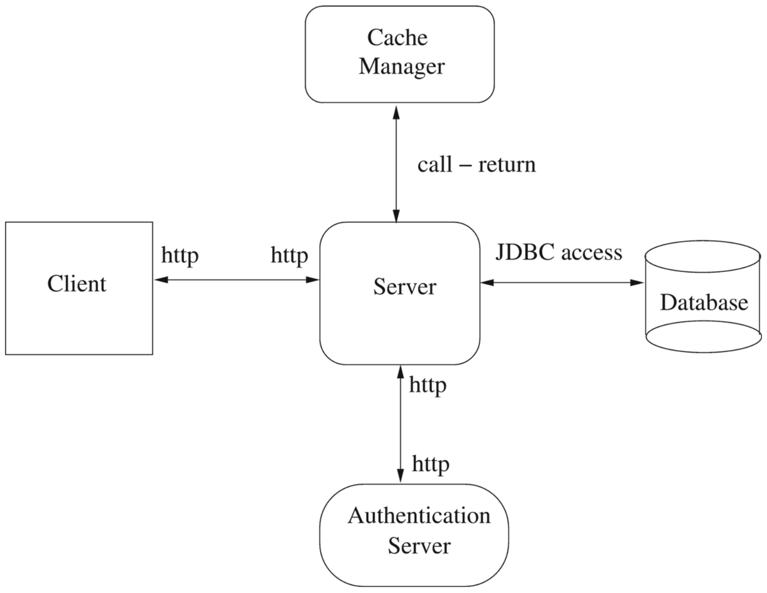


Fig. 4.5 Architecture for the survey system with cache

4.3 Architecture Styles

It should be clear that different applications may have different architectures. Some general architectures observed in many applications represent general structures useful for architecture of a class of applications. These are called *architectural styles*. A style defines a family of architectures that satisfy the constraints of that style [1,4]. In this section we discuss a few common styles for the C&C view useful for a broad set of problems. These styles can provide ideas for creating an architecture view for the problem at hand. A good compilation of a wide range of architectural styles can be found in [1,5].

An application may be a stand-alone monolith or distributed. In a stand-alone application the entire code for the application runs on one machine, often as a single process. A distributed application, on the other hand, has code which is distributed and portions of it run on different machines. Most of the architectural styles discussed here apply to both monolith and distributed applications—the main difference is that in a monolith the connector is often call-return type connector while in a distributed application the connector may be an http based connector.

4.3.1 Pipe and Filter

The Pipe-and-filter style of architecture is well suited for systems primarily performing data transformation whereby some input data is received and the goal of the system is to produce some output data by suitably transforming the input data. A system using pipe-and-filter architecture achieves the desired transformation by applying a network of smaller transformations and composing them in a manner such that together, the overall desired transformation is achieved.

The pipe-and-filter style has only one component type called the filter. It also has only one connector type, called the pipe. A filter performs a data transformation and sends the transformed data to other filters for further processing through a pipe connector. In other words, a filter receives the data it needs from some defined input pipes, performs the data transformation, and then sends the output data to other filters via defined output pipes. Filters can have multiple inputs and outputs operating as independent and asynchronous entities. They are concerned only with the data arriving on the pipe. A filter need not know the identity of the filters that sent the input data or one that will consume the data they produce.

The pipe is a unidirectional channel that conveys streams of data received from one end to the other. A pipe does not change the data in any manner but merely transports it to the filter on the receiver end in the order in which the data elements are received. As filters can be asynchronous and should work without the knowledge of the identity of the producer or the consumer, proper buffering and synchronization needs to be ensured by the pipe. The filters merely consume and produce data.

This style imposes some constraints. First, as mentioned above, the filters should work without knowing the identity of the consumer or the producer; they should only require the data elements they need. Second, a pipe, which is a two-way connector, must connect an output port of a filter to an input port of another.

A pure pipe-and-filter structure also generally requires each filter to have an independent thread of control to process data as it arrives. Implementing this necessitates an infrastructure that supports a pipe mechanism, which buffers the data and does the needed synchronization (for example, blocking the producer when the buffer is full and blocking the consumer filter when the buffer is empty). For using this pipe, the filter builder must be fully aware of the properties of the pipe, particularly with regard to buffering and synchronization, input and output mechanisms, and the symbols for the end of data.

However, there could be situations in which the constraint of real time data processing may not be required. Without this constraint, filters can produce the data completely before passing it on, or start their processing only after complete input is available. In such systems, filters cannot operate concurrently, and the system is like a batch-processing system. However, it can considerably simplify the pipes and easier mechanisms can be used for supporting them.

Let us consider an application for analyzing web server logs to detect anomalies and generate reports. This type of processing can be broken into a series of processors (filters) that each perform some focused processing to obtain the data needed by the next filter, and pass it on to that filter. The architecture for this application includes the following filters:

- **Read Log.** Reads raw log files from the web server and stores the log entries in a suitable format or data structure for processing. Input: Raw log files. Output: Organized log entries.
- **Parse Log.** Parses each log entry to extract relevant information such as timestamps, request types, URLs, response codes, and user agents, and prepares a structured data file (or data structure) with this data. Input: Raw log data. Output: Structured log data.
- **Analyze Log.** Analyzes the structured log data for patterns, such as detecting unusual traffic spikes or identifying the most frequently accessed pages. Input: Structured log data. Output: Analyzed results, such as traffic reports or anomaly alerts.
- **Report.** Generates reports based on the analyzed data. Input: Analyzed results. Output: Human-readable reports, graphs, or alerts.

Pipes in this architecture could pass files (through the file system) from one filter to the next, or use some type of in-memory queue. The architecture for this application is shown in Fig. 4.6.

In this architecture, the Read Log component reads a web server's log file, and passes it on to the Parse Log component. The Parse Log filter extracts fields such as IP address, timestamp, request type, and status code from each log entry and organizes them and passes the data to the Analyze Log component. The Analyze Log filter detects anomalies (e.g., a sudden increase in 404 errors) and other patterns and then passes this information to the Generate Report component. The Generate Report

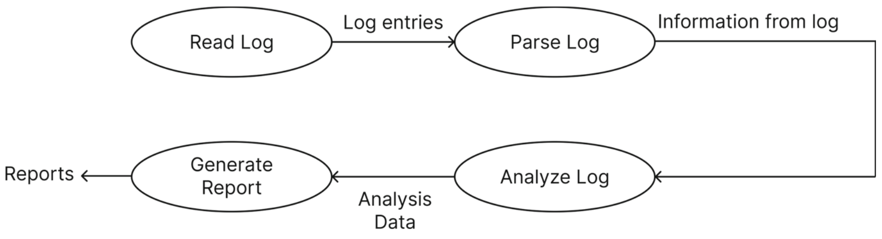


Fig. 4.6 Pipe-and-filter example

filter generates a summary report in a human readable form highlighting anomalies and the frequently accessed pages, which is then emailed to the web admin team.

This example illustrates the flexibility and modularity of the pipe and filter architecture style, allowing each filter to be independently developed, tested, and maintained while providing a clear and manageable workflow for complex data processing tasks.

The pipe and filter architecture is ideal for data processing pipelines, compilers, and data transformation applications. It promotes modularity, as each filter is an independent processing unit, and reusability, as filters can be easily reused in different pipelines. However, it introduces performance overhead due to data passing between filters and complicates error handling across multiple filters. It is not well suited for interactive applications.

4.3.2 Shared-Data

In the shared-data architectural style, there are two types of components—data repositories and data accessors. Components of data repository type are where the system stores shared data—these could be file systems or databases. These components provide a reliable and permanent storage, handle any synchronization needs for concurrent access, and provide data access support. Components of data accessors type access data from repositories, perform computation on the data obtained, and if they want to share the results with other components, put the results back in the repository. In other words, the accessors are computational elements that receive their data from the repository and save their data in the repository as well. These components do not directly communicate with each other—the data repository components are the means of communication and data transfer between them.

There are two possible variations of this style. In the Blackboard style, when data is posted to the data repository, all accessor components that need to know about it are informed. In other words, the shared data source acts as an active agent informing the components about the arrival of interesting data, or triggering the execution of components that need to act on the new data. In databases, this form of style is often supported through triggers. The other is the Repository style, in which the data repository is just a passive repository providing permanent storage and related controls for data accessing. The components access the repository as and when needed.

As can be imagined, many database-based applications use this architectural style. Databases, though originally more like repositories, now act both as repositories as well as blackboards as they provide triggers and efficient data storage. Many programming environments also use a similar organization: A common repository is used to store program artifacts that can be accessed by various tools to perform the desired translations or information retrieval.

Note that the different computation components do not need to communicate with each other and do not even need to know about each other's presence.

As an example of an application using this style of architecture, consider a student registration system in a University with a central repository containing information about courses, students, pre-requisites, etc. It has an `Administrator` component that sets up the repository, assigns rights to different people, etc. The `Registration` component allows students to register and updates the information for students and courses. The `Approvals` component is for granting approvals for courses requiring the instructor's consent. The `Reports` component produces reports on the students registered in different courses at the end of the registration period. The component `Course Feedback` is used for taking feedback from students at the end of a course. This architecture is shown in Fig. 4.7.

As different computation components do not communicate with each other, adding new components or modifying one becomes easy. To automate course scheduling based on registration data, a new scheduling component can be added without informing or modifying any existing components.

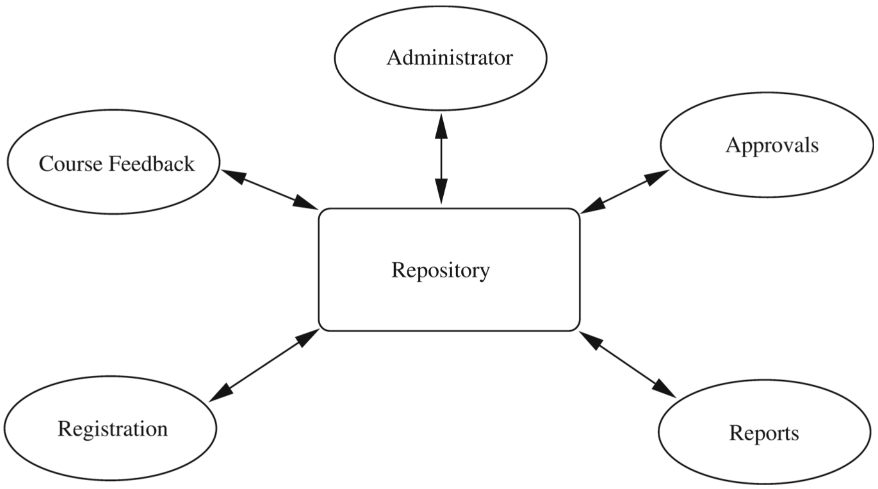


Fig. 4.7 Shared data example

There is really only one connector type in this style—read/write. Note, however, in specific architectures, this general connector type can take more precise forms. For example, though a database can be viewed as supporting read and updates, for a program interacting with it, the database system may provide transaction services as well. Connectors using this transaction service allow complete transactions (which may involve multiple reads and writes and preserve atomicity) to be performed by an application.

Note also that as in many other cases, these connectors require a considerable amount of underlying infrastructure. For example, read and writes to a file system involves a fair amount of file system software involving issues like directories, buffering, locking, and synchronization. Similarly, a considerable amount of software goes in databases to support the type of connections it provides for query, update, and transactions.

Shared-data architecture is best suited for data-centric applications, collaborative tools, and systems with central data storage requirements, such as databases and content management systems. It simplifies data management by centralizing data storage, ensuring data consistency and integrity. This centralization allows for easier access to data by different components, facilitating integration and data sharing across the application. However, as the system scales, the central repository can become a bottleneck, leading to potential performance issues. It also introduces a single point of failure, making the system more vulnerable. Managing concurrent access to shared data also adds complexity to the application.

4.3.3 Client-Server

The client-server architectural style is widely used to build applications today and is built upon the paradigm of Client-server computing, one of the basic paradigms of distributed computing.

In this style, there are two component types—clients and servers. A constraint of this style is that a client can only communicate with the server, not with other clients. The communication is initiated by the client when the client sends a request for some service that the server supports. The server receives the request at its defined port, performs the service, and then returns the results of the computation to the client.

There is one connector type in this style—the request/reply type. A connector connects a client to a server. This type of connector is asymmetric—the client end can only make requests (and receive the replies), while the server end can only send replies in response to the received requests. The communication is typically synchronous—the client waits for the server to return the results before proceeding. That is, the client is blocked at the request, until it gets the reply.

Most often, in a client-server architecture, the client and the server component reside on different machines. Even if they reside on the same machine, they are designed in a manner such that they can exist on different machines. Hence, the connector between the client and the server is expected to support the request/result type of connection across different machines. Consequently, these connectors are internally quite complex and involve a fair amount of networking to support.

Most network based remote services use this architecture style—e.g. webservers, file servers, domain name servers, and email servers. It is particularly effective for services which are stateless—that is, they don't carry state information from one request to another. Each request is processed independently and has no impact on the internal state of the server after it has been executed. It is well suited for applications where a clear separation between client-side and server-side responsibilities is necessary.

Client-server architecture promotes separation of concerns, allowing the client and server to evolve independently. It supports scalability, as servers can handle multiple clients simultaneously, and simplifies maintenance since server logic updates do not directly impact clients. However, it relies on reliable network communication, making it susceptible to network issues. Servers can become overloaded with too many client requests, leading to potential performance bottlenecks. Additionally, network latency can affect the responsiveness of the application.

4.3.4 3-Tier

The 3-tier architecture is an extension of the client server style. If the server in a client-server type application needs to access and manipulate data from a database, then the database for the application (providing a host of data related operations for the application), becomes the third tier—the data tier.

In the 3-tier architecture, the client tier is called the presentation tier. The code in this tier interacts with the users, collects the inputs from the user, and displays the information. The middle tier is generally called the business or application tier, which is the heart of the application performing much of the logic related to the application. The application tier receives inputs and commands from the presentation tier, and contains the code to perform the processing requested by the user. Whatever data it needs from the databases, it obtains by requesting the data tier, and saves data the results in the data tier, again by requesting it. The architecture style is shown in Fig. 4.8.

The 3-tier architecture is suitable for enterprise applications and web applications with significant business logic, and is perhaps the most common architecture style for building web applications. It enhances separation of concerns, allowing each layer to be developed and maintained independently. This separation improves maintainability and supports scalability through load balancing across different layers. However, the architecture increases overall complexity, requiring careful coordination across layers. Inter-layer communication can introduce latency, potentially affecting performance and deployment and updates can be challenging due to dependencies between layers.

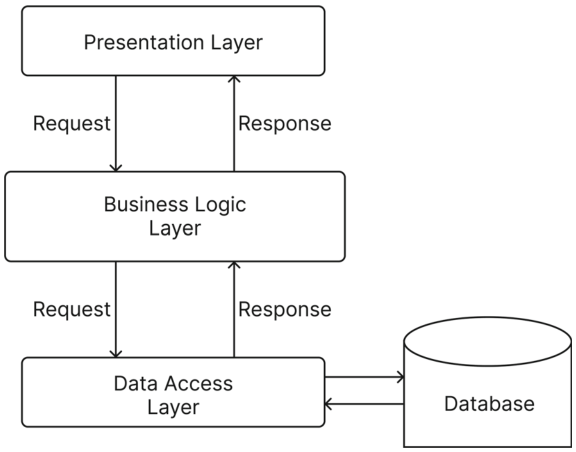


Fig. 4.8 3-Tier architecture style

4.3.5 Model View Template and Model View Controller

With the emergence of web development frameworks like Django, which is Python-based, the Model View Template (MVT) architecture style has become very popular for building web applications. For discussing this style, we will use the example of Django [6].

In a web application, the front-end is the browser through which the user interacts with the application hosted on the internet. To execute an operation in the application, the user provides a URL in the browser (by typing or by clicking a link in the application or a button), which is then executed by the back-end. The response is sent back by the back-end as an HTML page, rendered by the browser. In such applications, effectively, for any operations performed by the application there is an URL which the user provides.

In this style, an application, has the following components:

View: When a user types a URL related to this application, the webserver identifies what code is to be executed for this URL so as to return the desired output. This is managed by an index provided by the application designer (In Django this is specified in urls.py, where for each acceptable URL for the application, what function is to be executed is specified) to decide what function to execute. These functions together form the view component of the application (in Django they are kept in views.py). The functions in view work with the arguments in the URL request and then obtain the necessary data using functions in the models component, which may interact with the databases. The results to be sent to the user are then passed to a suitable template in the template component, which forms a suitable HTML page from this data provided. The view function sends the HTML page with all the information back to the requestor as response to the request. So, the functions in the view component perform all the business logic—they obtain the necessary data based on the user's request, do the necessary processing, and then send the desired data for rendering to the user through a template.

Model: Most user requests require data from some database. As discussed previously, databases typically run as separate processes (perhaps on a separate server), accept requests in some format (e.g. SQL), and return the result in some format (e.g. JSON). However, functions in the view component can only work with data structures provided by the language (e.g., lists, dictionaries, etc. for Python). So, some processing is needed to take the input information from a views function, convert it to suitable database commands to get the data from the database and convert it back to suitable data structures. This operation is quite complex as it requires establishing a connection with the database, interacting with it, and converting the data between the language data types and the data structures of databases. The functions in the model component provide this service. Using functions in the model component (in Django, they are kept in models.py folder), the caller functions from view do not have to worry about these complexities and can seamlessly work with familiar data types. In other words, the models component provides a layer of functions that provide for all the interactions to and from databases the application uses.

Template: After a user request has been processed and the necessary information requested by the user has been obtained by the function in views, this information has to be shown to the user. In a web application, this means that the information has to be embedded suitably in an HTML page, which has to be sent to the user for rendering by the browser. For forming this HTML page with suitable information, the function in views uses a suitable template of the application. Hence, the template forms the third component—the templates to be used to send back information to the user as a response to the user request (in Django, these templates are kept in the templates folder).

The MVT architecture pattern is shown in Fig. 4.9. This architectural style is feasible only if a framework is available to use these components in the manner described. For example, the framework must provide support for calling the appropriate function in the views component for a request based on the mapping provided by the application designer (e.g. in urls.py). The framework must also provide the functions in the models component support for connecting with different databases (typically is provided as a library with suitable functions). Similarly, when a template is used by a views function, the framework should interpret it suitably and populate the template

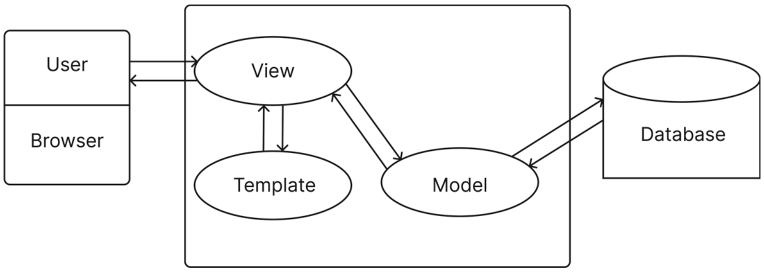


Fig. 4.9 Model-View-Template (MVT) architecture style

and prepare a suitable HTML page to be returned. Frameworks like Django provide all these, and a lot more, to the application designer.

A similar approach is taken in the Model-View-Controller (MVC) style, which is supported by frameworks like NodeJS and Rails. In this style, the back-end consists of three main components. The controller, the model and the view. The controller is the central component which has the back-end APIs. The front-end code for the application calls the back-end APIs. A request for an API comes to the controller component and the function for that API is executed. The API logic will interact with the models component to get the required data (for which the models component connects with the database) and perform the desired business logic. From the data returned from the models component, the controller may compute the final result that is to be returned to the user. For returning the result, the controller uses the view component, which has the functions that decide how the data is to be shown.

The views component functions prepare the html that is to be sent back to the user placing the data provided by the controller in suitable places in the html page (and for which it may use libraries, templates, etc.). The view component can be viewed as a set of html templates in which the controller provided data (which was requested by the user) can be easily embedded in suitable places in the html page. The controller then sends the page back to the user through the webserver.

So, in this architecture the controller services the requests for which it interacts with the models component and the view component. The controller coordinates between the two—it invokes functions in the models component to get the necessary work done and obtain the data, and invokes functions in views to compose and present results to the user. The models and view components do not interact directly.

A variation of MVC style uses more modern rendering techniques to improve upon the user experience. In this style, the front-end code essentially becomes a set of static files with JavaScript code embedded in them. The JavaScript code in a front-end page may provide link to get another page, or may make an API call in JavaScript code embedded in them. Unlike earlier approach where the back-end returns an html page, in this the API returns the results of the computation as JSON objects. Using these objects, the front-end code updates parts of the currently rendered page with new data from JSON objects. This approach makes the user experience smoother as the entire web page does not have to be reloaded—only those components of the page that need to change due to the results from the API need to be changed.

MVT is particularly suitable for web applications, especially those using the Django framework. It facilitates rapid development with Django's built-in features and promotes separation of concerns, making it easier to manage changes in the UI and logic independently. It also enhances maintainability by isolating different aspects of the application. However, the MVT framework has a learning curve for developers unfamiliar with it, potentially slowing down the initial development. Additionally, the framework can introduce some performance overhead due to its abstraction layers.

MVC is suitable for both web and desktop applications with rich user interfaces. It promotes separation of concerns, making it easier to manage application complexity. It enhances reusability, as components can be reused across different parts of the application, and improves testability by allowing independent testing of components. However, the architecture can add complexity, especially for smaller projects, due to the need to coordinate between the model, view, and controller. It may introduce performance overhead due to the additional layers and interactions required.

4.3.6 Microservices

In many of the styles discussed previously (3-tier, MVT, MVC), a single database layer contains all the data related to the application and the different APIs in the back-end make requests to it through a models layer. In web applications, the users are distributed across the globe, i.e. the front-end is running in parallel on many machines making requests to the back-end of the application. This situation can create a bottleneck at the back-end if the arrival rate of requests is too high. To handle large volume of requests, the back-end can be replicated on many servers with load-balancers ensuring that the API requests are sent to different servers, often in a round-robin manner. All these different servers connect to the same database for shared information.

This approach works well for most applications, particularly if the request load is not too high. However, if the requests rate is extremely high, while the back-end processing can be parallelized, the database becomes the bottleneck. It is not only a bottleneck for performance, but also for reliability—if the database fails, no requests can be serviced. If the database provides a backup service, it can take care of the reliability issue, but cannot still address the issue of slow response time in times of high load, which can result in lost customers.

To alleviate this situation, the microservices architecture style has been suggested [5]. In this style, the back-end is divided into small services, each providing a small set of related functions all using only a few data tables. The services and the data they need are combined together in a microservice. In other words, the back-end service and database layers are both partitioned and parts are combined to provide microservices. Each microservice is a stand-alone complete application with its own databases, allowing it to run independently on a server. With this architecture, the request load automatically gets divided into different microservices. However, for directing the requests to correct microservice requires suitable routers and load balancers.

In other words, in the microservices architectural style, an application is structured as a collection of small, independent services. Each microservice is self-contained and responsible for a specific business capability and can be developed, deployed, and scaled independently.

If a microservice needs some data or functionality which another microservice provides, it makes suitable API requests to that microservice. However, the partitioning in microservices should be such that most of the requests can be serviced by a microservice by itself—if for servicing a request a microservice has to routinely make API calls to other services, it will defeat the main purpose of microservice—to have independent services which can be deployed and scaled separately. It should be noted that this style provides resilience against failures, as failures in one service are isolated and do not bring down the entire system. There are other advantages of this style particularly for web applications with huge traffic.

E-commerce platforms often use this style. Such a platform may have microservices like product catalog, user authentication, shopping cart, order processing, and payment. Each service manages its own information (database), and provides suitable API endpoints for its services to be accessed. For example, the Product Catalog Service manages product information, categories, and inventory, which is kept in its own database. It may provide API to retrieve product information and update inventory. The Shopping Cart Service, manages users' shopping carts—stores cart items and user session data in its own databases, and provides APIs for adding, removing, and updating items in the cart. As we can see the databases as well as the services for the two are completely distinct. Similarly, the other services are also such that they work on limited data which they can own. Some services, for example, Order Processing, may require to use some functionality from other services (e.g. update the inventory, initiate payment), for which it will make API calls to those services. The architecture of such an application is shown in Fig. 4.10.

Microservices architecture is ideal for large, complex applications that require high scalability and consist of multiple independent components. It is suitable for environments where different parts of the application need to be developed, deployed, and scaled independently. It supports independent scaling of components, enhancing flexibility and fault isolation, as failures in one service do not impact the entire system. Different microservices can use various technologies, allowing for the best tool to be chosen for each task. However, this architecture increases complexity in managing inter-service communication and ensuring data consistency. It requires robust

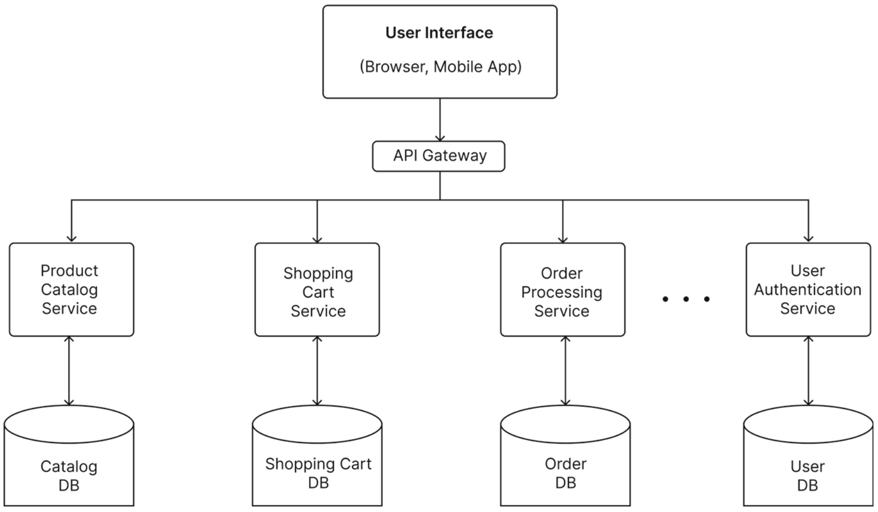


Fig. 4.10 An example of microservices style

pipelines for frequent updates to the application and introduces resource overhead, as each microservice may need its own infrastructure and management resources.

4.3.7 Some Other Styles

Publish-Subscribe

In this style, there are two main types of components. One type of component subscribes to a set of defined events. Other types of components generate or publish events. In response to these events, the components that have published their intent to process the event, are invoked. This type of style is prevalent in user interface frameworks, where many events are defined (like mouse click) and components are assigned to these events. When that event occurs, the associated component is executed. As is the case with most connectors, it is the task of the runtime infrastructure to ensure that this type of connector (i.e., publish-subscribe) is supported. This style can be seen as a special case of the blackboard style, except that the repository aspect is not being used.

Peer-to-Peer, or Object-Oriented

If we take a client-server style, and generalize each component to be a client as well as a server, then we have this style. In this style, components are peers and any component can request a service from any other component. The object-oriented computation model represents this style well. If we view components as objects, and connectors as method invocations, then we have this style. This model is the one that is primarily supported through middleware connectors like CORBA or .NET.

Communicating Processes

Perhaps the oldest model of distributed computing is that of communicating processes. This style tries to capture this model of computing. The components in this model are processes or threads, which communicate with each other either with message passing or through shared memory. This style is used in some form in many complex systems which use multiple threads or processes.

4.4 Architecture Design Decisions

When designing an architecture for an application, some key decisions must be made to develop the most suitable architecture [7]. While architecture impacts both functional and non-functional properties of an implementation, the latter are significantly more affected.Therefore, non-functional requirements often drive the choices made during architecture design. In this section, we will discuss three fundamental sets of decisions: Architectural Style, Technology Stack, and Other Considerations such as Security and Scalability. We will discusses the main issues, their importance, and how to approach these decisions. We will then briefly discuss how LLMs can be employed to help in these decisions.

Once the decisions are made, the architecture must be documented. The architecture design document should show the detailed component architecture, with brief description of each component. It should also document the decisions regarding technology stack etc., and provide suitable explanations about their suitability. We will briefly discuss documenting architecture and decisions.

4.4.1 Architectural Style

We have discussed above many different architectural styles. A key decision while designing an architecture for an application is which style should be adopted. Application requirements guide this decision. For example, when building a stand-alone application with a single well-defined task, the pipe-and-filter approach may be ideal. For developing a platform where new tools can be plugged in, the black-board style may be most suitable. For a web-application, generally a 3-tier type of style will be most suited. If the application needs to support rapid scaling and independent deployment of components, a microservices architecture might be appropriate.

Another factor in selecting the style is complexity of implementation and management. Monolithic architectures are simpler to implement but may become challenging to maintain as the application grows. Microservices offer flexibility but require robust inter-service communication and management.

A practical consideration is team expertise. It's often more effective to choose a style that aligns with the team's strengths.

The choice can be made through discussions and examining case studies of similar applications. If necessary, small prototypes can be built using different architectural styles to understand their practical implications.

4.4.2 Technology Stack and Open-Source Choices

The technology stack encompasses the programming languages, frameworks, libraries, and tools used for both front-end and back-end development including the database. The technology stack directly affects development speed, performance, compatibility, and the ease of finding skilled developers. Selecting a suitable stack is a key decision during architecture design. Sometimes the technologies to be used are specified in requirements. If not, it is desirable to make the choice during architecture design.

Open source technology stacks are very useful here—multiple open source technology stack choices are available. For our purposes here, we consider all technology choices available for free and without restriction on its use as open source (i.e. even those whose source code is not available). There are situations where a restricted version of a technology/tool may be available for free, but advanced features are paid. In such cases, the sponsor's input should be taken regarding the choice.

When making choices for technology, the team must ensure that the chosen technologies meet the application's requirements and are well suited for the chosen architectural style. Another important consideration is future-proofing. The application being developed may last for many years. Hence, the technology and tools should be such that they are likely to remain available for a long time, have community support for improving it and keeping it updated. Established technologies are generally more reliable, but emerging technologies might offer innovative solutions and better performance.

For most commonly used styles, some choices have emerged as suitable over time. Some common open-source technology choices for a few architectural styles are mentioned here.

- **Pipe and Filter.** (i) With its rich ecosystem of open-source libraries and frameworks, including those for workflow automation, Python is a suitable choice. It is also easy to learn. However, Python may not be the fastest language for performance-intensive tasks. (ii) Java, with its frameworks and libraries like Apache Camel, Spring Integration, etc., is also commonly used. It has robust integration capabilities and strong performance but may have higher complexity and a steeper learning curve.
- **Shared-Data or Repository-Based.** (i) JavaScript/TypeScript with frameworks and libraries like Node.js and Express.js are suitable. For common data store, MongoDB is appropriate as it facilitates storage and access of non-tabular data. This combination provides asynchronous I/O handling and ease of development. However, it may not permit multiple threads, which can impact performance. (ii) Java with frameworks and libraries like Spring Boot and Hibernate, along with MySQL database, is also suitable. It offers a mature ecosystem with strong performance and support for relational databases.
- **Client-Server, 3-Tier.** (i) One commonly used technology stack is: HTML, CSS, JavaScript (React) for the presentation layer; Node.js, Express.js for the business logic layer, and MongoDB for the data layer. Often called the MERN stack, it supports clear separation of concerns, reusable business logic, and scalability. (ii) Another popular stack is HTML, CSS, and Angular for the client, Java, Spring Boot for the server, and PostgreSQL for the data layer. (iii) Another stack, often called the MEAN stack, is a combination of Mongo DB, Express JS, Angular, and Node JS.
- **Model View Template (MVT).** Commonly used with Django framework with back-end code written in Python. The database can be PostgreSQL or another suitable database.
- **Model View Controller (MVC).** A common stack is: HTML, CSS, JavaScript for presentation layer; Node.js, Express.js for business logic layer; and MongoDB for data layer.

- **Microservices.** The 3-tier technology stacks can be used for each of the services. In addition, suitable options for gateway and load balancing have to be selected—Open source choices are available (e.g. NGINIX).

Various libraries and frameworks are available for different types of applications. For example, an application that is to work with images can use open-source libraries like OpenCV.

4.4.3 Other Considerations (Security, Scalability, etc.)

Beyond architectural style and technology stack, other critical considerations during architecture design includes security, scalability, performance, and maintainability. Suitable decisions on these will ensure that the application remains robust, responsive, and secure as it grows and evolves. Here we briefly discuss a few of these.

- **Security.** To protect user data, allow only authorized users to access the application, prevent breaches, etc. Employ best practices such as encryption, authentication, and regular security audits. Choose technologies and frameworks with built-in security features and strong community support for addressing vulnerabilities. For authentication, standard services which have been proven can be used, e.g. google auth.
- **Scalability.** Scalability ensures the application can handle increasing loads without degrading performance. This is of importance only for applications that may have a high load. Use load balancing and distributed architectures to manage traffic efficiently and distribute the load.
- **Performance.** Performance impacts user experience and satisfaction. For high performance, the development team needs to optimize code, use caching mechanisms, and choose high performance technologies. Performance monitoring tools should be employed to identify and address bottlenecks.
- **Hosting.** For a web application, whether to host it on an organization server or a cloud is a major decision. Though strictly not an architecture issue, this is often decided during the architecture design, as the type of server needed or the resources that will be needed in the cloud will be impacted by performance and other requirements and the architectural choices made.
- **Maintainability.** As applications are long lasting and evolve with time, ease of maintaining the code becomes important. Maintainability reduces the overall cost of ownership of an application over its life and ensures that the application can be easily updated or modified. Suitable architectural style should be used that supports modularity and decoupling of different parts of the application, and suitable technologies/tools for version control systems, test automation, documentation, etc. should be selected.

4.4.4 Using LLMs for Architectural Decisions

LLMs can be useful for making the architectural design choices. Given the multitude of decisions required and the numerous options for each, a one-prompt approach is unlikely to be effective. It is best to use a multi-prompt approach using the tree-of-thought type of prompting, where the LLM is provided with the requirements and then prompted for each architectural decision to provide multiple choices with their advantages and disadvantages. From the choices, one may be selected. This process can be repeated for subsequent decisions. For example, in the first prompt (or in the context portion of the prompt) the SRS can be given, specifying that questions will be asked about suitable architecture for this. We illustrate this by the series of prompts that were given for the example SRS given in the previous chapter.

> Prompt 1: Given below is the software requirements specifications for an application. In subsequent prompts we will ask you some questions about suitable architecture for implementing this application.
> ...SRS...

> Prompt 2: Please suggest two most suitable architectural styles for this application and give the advantages and disadvantages of each.

ChatGPT suggested two architecture styles—MVC and Microservices. For each it provided some advantages and disadvantages, and in the end summarized that MVC Architecture might be more suitable if the application is expected to have a more straightforward structure with clear separation between the user interface, data handling, and business logic.

> Prompt 3: We would like to use the MVC architecture. Please suggest a few technology choices for each of the layers as well as the database for this application with pros and cons for each.

The LLM suggested various choices, eventually recommending MongoDB for the Model Layer, React JS for the View layer, and Express.js with Node.js for the Controller layer.

> Prompt 4: For authentication, we would like to use some reliable external service. Please suggest some with their strengths and weaknesses.

It suggested choices like Google Identity Platform, Auth0, Amazon, and Cognito, summarizing that Google Identity Platform seems to be the most appropriate choice for this application.

As we can see an LLM it is able to generate reasonable suggestions for various decisions, often including the desirable choice. In this example where the architecture choices were already made, we found most of the decisions we took were included in the suggestions. Overall, exploring options using an LLM can be a good way to make architecture design decisions.

4.4.5 Documenting and Evaluating Architecture

The architecture has to be properly documented and communicated to all stakeholders for discussion and agreement. Properly documenting an architecture is as important as creating one, and templates have been proposed for their documentation (e.g. [1–3]).

We know that an architecture for a system is driven by the system objectives and needs of the stakeholders. Therefore, an architecture document should provide the context for the architecture design— the stakeholders and their concerns, key requirements that the architecture has to support including the non-functional requirements, etc.

With the context defined, the document can proceed with describing the architectural view(s). The description of the architecture will almost always contain a pictorial representation of the view. As discussed earlier, in any view diagram it is desirable to have different symbols for different element types and provide a key for the different types, such that the type of the different components (represented using the symbols) is clear to a reader. It is, of course, highly desirable to keep the diagram simple and uncluttered (and omit some minor aspects which can be described separately.)

However, a pictorial representation is not a complete description of the view. It gives an intuitive idea of the design, but is not sufficient for providing the details. For example, the purpose and functionality of a component is indicated only by its name which is not sufficient. Hence, the document should also explain each of the component's purpose and its interface.

Besides selecting the architecture style and the high level view, as mentioned above, other architectural decisions must also be taken. It is important that these decisions be documented with an explanation of the reasoning for their selection. Documenting architecture decisions is considered a good practice and guidelines have been proposed on how to document these (e.g. [5]). These decisions include the choice of technologies, tools, etc., as well as how desired attributes like performance, security, etc. are being met.

It is desirable to have an evaluation of the architecture to show that the choices made support the requirements for the application. How should a proposed architecture be evaluated for these attributes—we will very briefly discuss this here.

For some attributes like performance and reliability, it is possible to build formal models using techniques like queuing networks and use them for assessing the value of the attribute. However, these models require information beyond the architecture description, generally in forms of execution times, and reliability of each component.

A more common approach is to to subjectively evaluate the impact of the architecture on some of the attributes. In this approach, attributes that are impacted by the architecture are listed, and reasoning is provided on why the decisions taken will achieve the desired level for the attribute. This can be done by evaluating the level that may be achieved for each attribute (e.g., good, average, poor), or might simply mention whether it is satisfactory or not. Many techniques have been proposed for evaluation, and a survey of them is given in [8].

4.5 An Example

In the earlier chapter, we provided the requirements specification for an application to manage the students' events in a college. Here, we present the architecture design for this application along with the key architectural decisions. More details are available on the website for the book.

Architecture Design

The application will follow the variation of the MVC architecture style, where the front-end code consists of a set of HTML files with embedded JavaScript code that can call back-end APIs. The back-end API returns JSON objects, and the front-end code updates parts of the currently rendered page with the information in the JSON objects. Hence, in this architecture, the front-end component contains some files for rendering pages to the user, while the backend component contains the code for executing the APIs that provide information for parts of the front-end pages. The overall architecture is shown in Fig. 4.11.

Component Descriptions

Front-end: It essentially contains a set of HTML files with JavaScript code embedded in them. These are the screens the user sees, each with various components, some of which may be updated due to an API call made by the JavaScript code in the page.

Back-end: The Back end consists of many components.

- **Router.** The router is responsible for mapping incoming API requests to the appropriate controller actions and handling the routing logic, ensuring that requests are directed to the correct parts of the application.
- **Middleware.** A request passes through the middleware, where the user is authenticated (using Google Auth) and the user's privileges checked. Suitable flags are set.

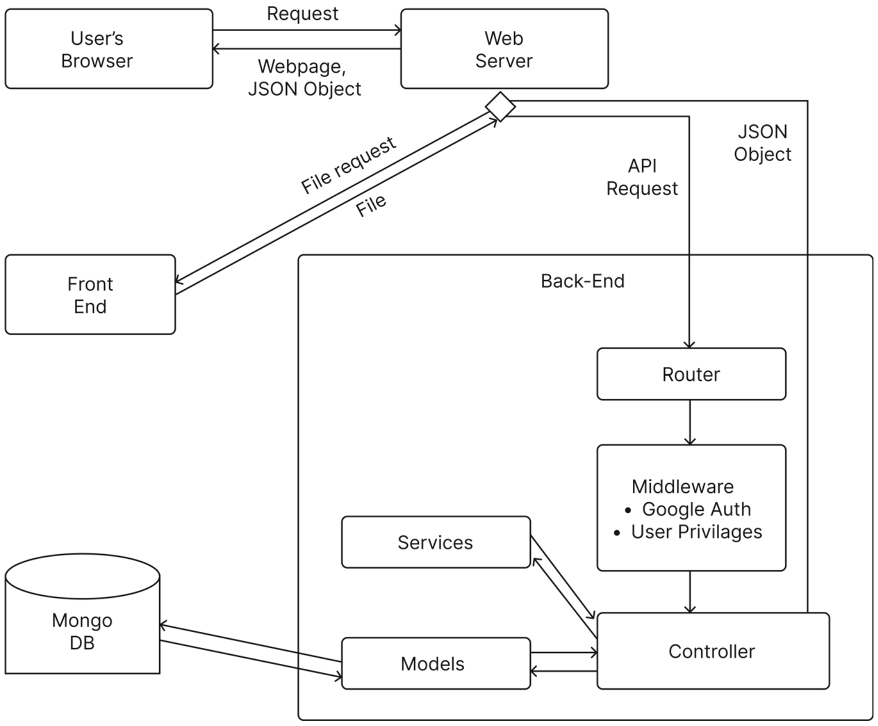


Fig. 4.11 Proposed architecture for the students events applications

- **Controller.** Contains the code for all the APIs. Upon an API request, code for that API is executed (by the application server). Controller modules interact with Models for database access, and use various services. Upon completion of an API request, it sends a JSON object back to the browser, which updates the webpage with the information.
- **Models.** Contains the functions to create the databases and establish connections. It passes the database tables to the controller as objects on which the controller can make method calls for retrieving data or executing the operations.
- **Services.** These are various services that the controller may use.

Technology Stack and Other Choices

The following technologies have been chosen. The criteria for selection was that the technology should be freely available, and should be able to support the functional as well as non-functional requirements of the application. Another factor in selection was familiarity with the technology of the team.

Frontend

- HTML (Hyper Text Markup Language) and CSS (Cascading Style Sheets): Using HTML and CSS as core technologies ensures compatibility across various web browsers and devices. They provide full control over the look and feel of the application's user interface.
- React.js: It is free and open source and its component-based structure and wide range of libraries facilitate building the application quickly with minimal code development, which will also promote maintainability.
- Redux: Redux is an open source and rich library that provides for managing application state, facilitating predictable data flow and updates. This is particularly useful for handling the state of event requests, user registrations, and notifications.

Backend

- TypeScript: TypeScript is used as the backend programming language. It is open source, enhances code maintainability and catches type-related errors during development, reducing runtime issues.
- Node.js: Node.js is one of the most popular open source JavaScript backend for developing web-based applications. It provides rich libraries, eases connections with databases, and has event-driven, non-blocking I/O architecture, which allows for good response time and high concurrency.
- Express.js: It is an open-source web application framework that provides functionality for routing, simplifying the process of defining API routes and handling HTTP requests using MVC (Model View Controller) architecture. It also provides support for development.

Database

MongoDB: Its NoSQL, document-oriented nature is well-suited for handling diverse data types, such as event details, user profiles, and club information. Its flexibility allows for efficient data retrieval and storage, which is essential for managing event requests and club information. It is open source and allows limited use for free.

Security

It was decided to use the Google Auth service for user authentication. This is a proven and reliable service and is available for free.

Hosting

The application will be hosted on a virtual server to be provided by the organization. A suitable configuration will be sought to ensure fast execution of the application. The option of cloud hosting was not chosen, as the organization has sufficient servers and expertise to manage them.

Evaluation

- The architecture allows for new features to be added to existing screens, by suitably adding new APIs and changing the front-end code to call the new APIs. Completely new screens in the front-end can also be easily added. Hence, the architecture supports ease of modifiability.
- Security and access control is ensured by Google auth. It also ensures the design constraint that institute email be used for login and authentication. (Some other security requirements like input validation will be supported during implementation.)
- Performance challenges have been minimized by selecting the modified MVC style and the chosen technology stack, which ensures that each request does not need to result in the full page being loaded. To ensure performance constraints are met, a suitable virtual host is to be allocated to host the application.
- As standard technologies (e.g. html, css, javascript, etc.) that are supported in all major browsers are used, compatibility requirements will be met.

4.6 Summary

- Architecture of a software system provides a very high-level view of the system in terms of parts of the system and how they are related to form the whole system. Architecture impacts properties like performance of the application. Depending on how the system is partitioned, we get a different architectural views of the system. Consequently, the architecture of a software system is defined as the structures of the system which comprise software elements, their externally visible properties, and relationships among them.
- There are three main architectural views of a system—module, component and connector, and allocation. In a module view, the system is viewed as a structure of programming modules like packages, classes, functions, etc. In a component and connector (C&C) view, the system is a collection of runtime entities called components, which interact with each other through the connectors. An allocation view describes how the different software units are allocated to hardware resources in the system.
- C&C view is most common, and is often the centerpiece of the architecture description. This view is frequently described by block diagrams specifying the different components and the different connectors between the components.

- Some common architecture styles for building applications are are pipe and filter, shared data, client-server, 3-Tier, Model-View-Template, and microservices. Each of these styles describes the types of components and connectors that exist and the constraints on how they are used.
  - The pipe and filter has one type of component (filter) and one type of connector (pipe) and components can be connected through the pipe.
  - In shared data style the two component types are repository and data accessors. Data accessors read/write the repository and share information among themselves through the repository.
  - The client-server style has two types of components (client and server) and there is one connector (request/reply). A client can only communicate with the server, and an interaction is initiated by a client.
  - The 3-tier style is an extension of client-server by adding data store as the third layer behind the server. 3-Tier architecture is commonly used for developing web-applications with front-end, back-end, and database layers to implement the application.
  - The model-view-template (MVT) type of style is used with web application development frameworks like Django. The back-end is divided into model, view, and template layers—the model layer provides code to interact with the database, the view provides the business logic and also uses the modules in template to form responses to requests from the font-end.
  - In microservices style, a service and its associated database is combined into a microservice. A microservice can be developed, deployed, and scaled independently.
- Designing architecture entails making architectural decisions on what architectural style to use, what technology stack to use, and what open-source choices exist for them. It also includes deciding how to ensure properties like security, scalability, performance, etc. Hosting options for the application may also be considered. All these decisions should be documented, and an evaluation should be done to ensure that the decisions taken support the application requirements.
- LLMs can be quite useful for making these architectural decisions. An exploratory approach using prompt chaining can work well. The requirements are first provided, and then in each prompt, choices for a decision are asked for, and then one is chosen before exploring the options for other decisions.

Self-Assessment Exercises

1. For stand-alone applications and for web-applications, suggest two architectural styles that you think will work for a range of applications? Clearly explain the nature of the connectors between components and how are they provided (e.g. by the programming language, a library, operating system, etc.)

2. Consider an interactive website which provides many different features to perform various tasks. Show that the architecture for this can be represented as a shared-data style as well as client-server style. Which one will you prefer and why?
3. Think of an application that you have to build. List the key architectural decisions you will need to take while designing the architecture for it.
4. Take two different technology stacks for a web application. List the conditions under which you will use one over the other.
5. Consider an application using an MVT architecture. List the events that occur in this application when a user requests a service.
6. Suggest how you will evaluate a proposed architecture from a modifiability perspective.

References

1. L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, 2nd edn. (Addison-Wesley Professional, 2003)
2. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford, *Documenting Software Architectures: Views and Beyond* (Addison-Wesley, 2003)
3. IEEE, IEEE recommended practice for architectural description of software-intensive systems. Technical Report 1471-2000 (2000)
4. M. Shaw, D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline* (Prentice Hall, 1996)
5. M. Richards, N. Ford, *Fundamentals of Software Architecture: An Engineering Approach* (O'Reilly Media, 2020)
6. w3schools. Django introduction (2024)
7. A. Jansen, J. Bosch, Software architecture as a set of architectural design decisions, in *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)* (IEEE, 2005), pp. 109–120
8. L. Dobrica, E. Niemela, A survey on software architecture analysis methods. IEEE Trans. Softw. Eng. **28**(7), 638–653 (2002)

5 Design

The design activity begins when the architecture has been defined. During design we refine the architecture and identify the code modules to be developed. Often, design of each component is done separately and we can view the design exercise as determining the module structure of the components. However, this simple mapping of components and modules may not always hold, and some modules may cut across components.

The design process often has two levels. At the first level, the modules required for the system, their specifications, and how they are interconnected are designed. This is called the module-level or high-level design. In the second level, often called detailed design or logic design, we determine how the specifications of the module can be implemented. In this chapter we focus on high-level design.

A *design methodology* is a systematic approach to creating a design by applying of a set of techniques and guidelines. Design methodologies focus on the module-level design, and provide guidelines for it, though they do not reduce the design activity to a sequence of steps that can be blindly followed.

In this chapter, we discuss:

- The key design concepts of modularity, cohesion, coupling, and open-closed principle.
- The structure chart notation for expressing the structure of a function-oriented system, and the structured design methodology for designing an application.
- Some key concepts in the Unified Modeling Language (UML) that can be used to express an object-oriented design, and a methodology for creating the object-oriented design for an application.
- An approach for designing an a 3-tier application.
- How to effectively use LLMs for designing.

5.1 Design Concepts

The design of a system is *correct* if a system built according to the design satisfies the requirements of that system. The goal of the design activity is not simply to produce a correct design, but to find the *best* possible design within the limitations imposed by the requirements and the physical and social environment in which the system will operate.

To assess a design, we must specify some evaluation criteria. We will focus on modularity of a system, which is decided mostly by design, as the main criterion for evaluation. A system is considered *modular* if it consists of discrete modules so that each module can be implemented separately, and a change to one module has minimal impact on other modules.

Modularity is clearly a desirable property. It helps in system debugging—isolating the system problem to a module is easier if the system is modular; in system repair—changing a part of the system is easy as it affects few other parts; and in system building—a modular system can be easily built by "putting its modules together."

For modularity, each module in a component needs to support a well-defined abstraction and have a clear interface through which it interacts with other modules. To produce modular designs, some criteria must be used to select modules so that the modules support well-defined abstractions and are solvable and modifiable separately. Coupling and cohesion are two modularization criteria, which are often used together. We also discuss the open-closed principle, which is another criterion for modularity.

5.1.1 Coupling

Two modules are considered independent if one can function completely without the other. Obviously, if two modules are independent, they are solvable and modifiable separately. However, all the modules in a system cannot be independent of each other, as they must interact to produce the desired external behavior of the system. The more connections between modules, the more dependent they are, in the sense that more knowledge about one module is required to understand or solve the other. Hence, the fewer and simpler the connections between modules, the easier it is to understand one without understanding the other. The notion of coupling [1,2] attempts to capture this concept of "how strongly" different modules are interconnected.

Coupling between modules is the strength of interconnections between modules, or a measure of interdependence among modules. In general, the more we must know about module A in order to understand module B, the more closely connected A is to B. "Highly coupled" modules are joined by strong interconnections, while "loosely coupled" modules have weak interconnections. Independent modules have no interconnections. In a good design, module should be loosely coupled with other modules. The choice of modules decides the coupling between them. Because the modules of the software system are created during system design, the coupling is largely decided during design.

Table 5.1 Factors affecting coupling

	Interface complexity	Type of connection	Type of communication
Low	Simple obvious	To module by name	Data
			Control
High	Complicated obscure	To internal elements	Hybrid

Coupling increases with the complexity and obscurity of the interface between modules. To keep coupling low, we should minimize the number of interfaces per module and the complexity of each interface. An interface of a module is used to pass information to and from other modules. Coupling is reduced if only the defined entry interface of a module is used by other modules, for example, passing information to and from a module exclusively through parameters. Coupling increases if a module is used by other modules via an indirect and obscure interface, like directly using the internals of a module or shared variables.

The complexity of the interface is another factor affecting coupling. The more complex the interface, the higher the degree of coupling. For example, complexity of the entry interface of a procedure depends on the number of items being passed as parameters and on the complexity of the items. Some level of complexity of interfaces is required to support the communication needed between modules. However, often more than this minimum is used. For example, if a field of a record is needed by a procedure, often the entire record is passed, rather than just the needed field. By passing the record we are increasing the coupling unnecessarily. Essentially, we should keep the interface of a module as simple and small as possible.

The type of information flow along the interfaces is the third major factor affecting coupling. There are two kinds of information that can flow along an interface: data or control. Passing or receiving control information means that the action of the module will depend on this control information, making it more difficult to understand the module and provide its abstraction. Transfer of data information means that a module passes some input data to another module and gets in return, some data as output. This allows a module to be treated as a simple input-output function that performs some transformation on the input data to produce the output data. In general, interfaces with only data communication result in the lowest degree of coupling, followed by interfaces that only transfer control data. Coupling is highest if the data is hybrid, meaning some data items and some control items are passed between modules. The effect of these three factors on coupling is summarized in Table 5.1 [1].

The manifestation of coupling in OO systems is somewhat different as objects are semantically richer than functions. In OO systems, three different types of coupling exist between modules [3]:

- Interaction coupling
- Component coupling
- Inheritance coupling.

Interaction coupling occurs when methods of a class invoke methods of other classes. Like with functions, the worst form of coupling occurs when methods directly access internal parts of other methods. Coupling is lowest if methods communicate directly through parameters, especially if only data is passed, but is higher if control information is passed since the invoked method impacts the execution sequence in the calling method. Additionally, coupling is higher if the amount of data being passed is increased. The least coupling situation happens when communication occurs with parameters only, with only necessary information being passed, and these parameters only passing data.

Component coupling refers to interactions between two classes where a class has variables of the other class. This occurs in three clear ways. A class C can be component coupled with another class C1, if C has an instance variable of type C1, or C has a method whose parameter is of type C1, or if C has a method which has a local variable of type C1. Component coupling is considered to be weakest (i.e. most desired) if in a class C, the variables of class C1 are either in the signatures of the methods of C, or are some attributes of C. If interaction is through local variables, then this interaction is not visible from outside, and therefore increases coupling.

Inheritance coupling arises from inheritance relationship between classes. Two classes are considered inheritance coupled if one class is a direct or indirect subclass of the other. Within inheritance coupling there are some situations that are worse than others. The worst form is when a subclass B1 modifies the signature of a method in B (or deletes the method). This situation can lead to runtime errors besides violating the open-closed principle (discussed below). Changing a method's implementation while preserving its signature violates the implied contract that objects of the parent class are also objects of the subclass. The least coupling scenario is when a subclass only adds instance variables and methods without modifying any inherited ones.

5.1.2 Cohesion

We have seen that coupling is reduced when the relationships among elements in different modules are minimized. That is, coupling is reduced when elements in different modules have little or no bonds between them. Another way to achieve this effect is by strengthening the bond between elements within the same module by maximizing their relationships. Cohesion is the concept that tries to capture this intramodule [1,2]. With cohesion, we are interested in determining how closely the elements of a module are related to each other.

Cohesion represents how tightly bound the internal elements of the module are to one another. Cohesion of a module gives the designer an idea about whether the different elements of a module belong together. Cohesion and coupling are clearly related. Generally, the greater the cohesion of each module in the system, the lower the coupling between modules is. This correlation is not perfect but has been observed in practice. There are several levels of cohesion:

- Coincidental
- Logical
- Temporal
- Procedural, Communicational, Sequential
- Functional.

Coincidental cohesion is the lowest level, and functional cohesion is the highest. Coincidental cohesion can happen when there is no meaningful relationship among the elements of a module. Coincidental cohesion can occur if an existing program is "modularized" by chopping it into pieces and making different pieces modules. If a module is created to save duplicate code by combining parts of the code that occur in many different places, that module is likely to have coincidental cohesion.

A module has logical cohesion if there is some logical relationship between its elements and these elements perform functions that fall in the same logical class. A typical example of this kind of cohesion is a module that performs all inputs or all outputs. In such a situation, if we want to input or output a particular record, we have to somehow convey this to the module, often, by passing some kind of special status flag, to determine what statements to execute. This results in hybrid information flow between modules, which generally leads to the worst form of coupling, and usually creates tricky and clumsy code. In general, logically cohesive modules should be avoided if possible.

Temporal cohesion is similar to logical cohesion, except the elements are also related in time and are executed together. Modules that perform activities like "initialization," "cleanup," and "termination" are usually temporally bound. Even though the elements in a temporally bound module are logically related, temporal cohesion is higher than logical cohesion, because the elements are all executed together.

A procedurally cohesive module contains elements that belong to a common procedural unit. For example, a loop or a sequence of decision statements in a module may be combined to form a separate module. A module with communicational cohesion has elements related by a reference to the same input or output data. That is, in a communicationally bound module, the elements are together because they operate on the same input or output data. Communicationally cohesive modules may perform more than one function. When the elements are together in a module because the output of one forms the input to another, we get sequential cohesion. If we have a sequence of elements in which the output of one forms the input to another, sequential cohesion does not provide any guidelines on how to combine them into modules.

Functional cohesion is the strongest cohesion. In a functionally bound module, all the elements are related to performing a single function. By function, we do not mean simply mathematical functions; modules accomplishing a single goal are also included. Functions like "compute square root" and "sort the array" are clear examples of functionally cohesive modules.

How does one determine the cohesion level of a module? There is no mathematical formula that can be used; We have to use our judgment for this. A useful technique is to write a sentence that describes, fully and accurately, the function or purpose of the module. Modules with functional cohesion can always be described by a simple sentence. If we cannot describe it using a simple sentence, the module likely does not have functional cohesion.

Cohesion in object-oriented systems has three aspects [3]:

- Method cohesion
- Class cohesion
- Inheritance cohesion.

Method cohesion is the same as cohesion in functional modules. It focuses on why the different code elements of a method are together within the method. The highest form of cohesion is if each method implements a clearly defined function, and all statements in the method contribute to implementing this function.

Class cohesion focuses on why different attributes and methods are together in a class. The goal is to have a class that implements a single concept or abstraction with all elements contributing toward supporting this concept. In general, whenever there are multiple concepts encapsulated within a class, the cohesion of the class is not as high as it could be. A designer should try to change the design to have each class encapsulate a single concept.

One symptom of a class having multiple abstractions is when the set of methods can be partitioned into two (or more) groups, each accessing a distinct subset of the attributes. That is, the set of methods and attributes can be partitioned into separate groups, each encapsulating a different concept. Clearly, in such a situation, having separate classes encapsulating separate concepts can improved cohesion.

Inheritance cohesion focuses on why classes are together in a hierarchy. The two main reasons for inheritance are to model generalization-specialization relationships, and for code reuse. Cohesion is considered high if the hierarchy supports generalization-specialization of some concept. It is considered lower if the hierarchy is primarily for sharing code with weak conceptual relationship between superclass and subclasses.

5.1.3 The Open-Closed Principle

This is a design concept which came into existence more in the OO context. Like cohesion and coupling, its primary goal is to promote building of easily modifiable systems. As modification and change happen frequently, a design that cannot easily accommodate these changes will result in systems that will not be able to easily adapt to the changing world and become obselete.

The basic principle, as stated by Bertrand Meyer, is “Software entities should be open for extension, but closed for modification” [4]. This means that while a module’s behavior can be extended to accommodate new demands placed on it due to changes in requirements and system functionality, the existing source code should not be altered when making enhancements.

This principle restricts the changes to modules to extension only, i.e. it allows addition of code, but disallows changing of existing code. If this can be done, clearly,

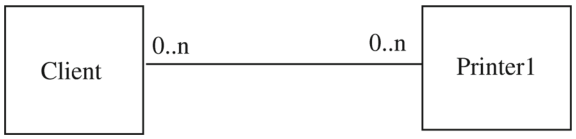


Fig. 5.1 Example without using subtyping

the value obtained is tremendous. Code changes involve heavy risk and to ensure that a change has not "broken" things that were working often requires a lot of regression testing. This risk is minimized if no changes are made to existing code.

In OO design, this principle can be satisfied through inheritance and polymorphism. Inheritance allows new classes to extend the behavior of existing classes without modifying the original class. And it is this property that can be used to support this principle. For example, consider an application where a client object (of type Client) interacts with a printer object (of class Printer1) and invokes the necessary methods for completing its printing needs. The class diagram for this will be as shown in Fig. 5.1.

In this design, the client directly calls the methods on the printer object. If we need to introduce a new printer 'Printer2', to be used by the client, the client code must be modified to accommodate this new printer. This design does not support the open-closed principle as the Client class is not closed against changes.

A better design which the Open-Closed principle, involves using an abstract class 'Printer' that defines the interface of a printer and specifies all the methods a printer object should support. Printer1 is implemented as a specialization of this class. In this design, when Printer2 needs to be added, it is created as another subclass of type Printer. The client does not need to be aware of this subtype as it interacts with objects of type Printer. That is, the client only deals with a generic Printer, and its interaction is the same whether the object is actually of type Printer1 or Printer2. The class diagram for this is shown in Fig. 5.2.

It is this inheritance property of OO that is leveraged to support the open-closed principle. The idea to encapsulate the abstraction of a concept within a class. If this abstraction needs to be extended, the extension is done by creating new subclasses, thereby keeping the existing code unchanged.

If inheritance hierarchies are built in this manner, they are said to satisfy the Liskov Substitution Principle [5]. According to this principle, if a program is using object o1 of a (base) class C, that program should remain unchanged if o1 is replaced by an object o2 of a class C1, where C1 is a subclass of C. By satisfying this principle and using hierarchies properly, the Open-Closed principle can be supported. It should also be noted that recommendations for both inheritance coupling and inheritance cohesion also support following this principle in class hierarchies.

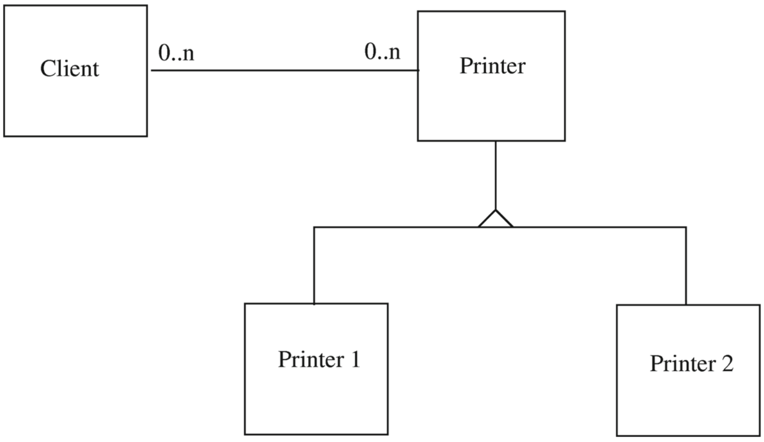


Fig. 5.2 Example using subtyping

5.2 Design Methodologies

During the design phase it is crucial to identify the code modules that need to be implemented for implementing the components of the application. These modules can be functions or classes. Traditionally design approaches have focused on using one of the other, though these days, applications in languages like Python often utilise both types of modules. Methodologies have been proposed for both. Here we will briefly discuss approaches for function-oriented design (i.e. the modules are functions) and object-oriented design (where modules are classes). These approaches are particularly useful when developing stand-alone applications. For a distributed application, other methods may be more useful—we will discuss one in the next section.

5.2.1 Function-Oriented Structured Design

Structured design methodology was one of the popular approaches for designing function-oriented software [1,2]. We first discuss some concepts for developing function-oriented designs. Then we will discuss the structured design methodology.

Structure Charts

The structure of a program is consists of the modules of that program together with the interconnections between them. Every computer program has a structure, and given a program its structure can be determined. The structure chart of a program is a graphic representation of its structure. In a structure chart a module is represented as a box with its name written inside. An arrow from module A to module B indicates that module A invokes module B. B is called the *subordinate* of A, and A is called the *superordinate* of B. The arrow is labeled by the parameters received by B as input and the ones returned by B as output, with the direction of flow of these input and output parameters represented by small arrows.

Generally, information about the logic within a function is not represented in a structure chart, and the focus is on representing the hierarchy of modules. Modules in a design can be categorized into few classes. There are modules that obtain information from their subordinates and then pass it to their superordinate. These kind of modules are *input modules*. Similarly, there are *output modules*, which take information from their superordinate and pass it on to its subordinates. As the names suggest, the input and output modules are typically used for input and output of data from and to the environment. Then there are modules that exist solely for the sake of transforming data into another form. Such modules are called *transform modules*. Most of the computationally intensive modules typically fall in this category. Finally, there are modules whose primary concern is managing the flow of data to and from different subordinates. Such modules are called *coordinate modules*. A module can perform functions characteristic of more than one type.

A structure chart effectively represents a design that uses functional abstraction, showing the modules and their call hierarchy, the interfaces between the modules, and information exchanged between them. So, for a software component, once its structure is decided, the modules, their interfaces, and dependencies get fixed.

Structured Design Methodology

The objective of the structured design methodology is to provide guidance on how to evolve the structure chart for an application so that the modules in the design have a hierarchical structure with functionally cohesive modules and minimal interconnections between them (low coupling).

The basic principle behind the structured design methodology, as with most other methodologies, is problem partitioning. Structured design methodology partitions the system at the very top level into various subsystems, one for managing each major input, one for managing each major output, and one for each major transformation. The modules performing the transformation deal with data at an abstract level, and hence can focus on the conceptual problem of performing the transformation without dealing with how to obtain clean inputs or how to present the output.

The rationale behind this partitioning is that in most applications, a significant portion of the code deals with managing the inputs and outputs. Modules dealing with inputs deal with issues related to screens, reading data, formats, errors, exceptions, completeness of information, structure of the information, etc. Similarly, modules dealing with outputs prepare the output in presentation formats, make charts, produce reports, etc. Hence, for many applications, the bulk of the software deals with inputs and outputs. The actual transformation in the system is often not very complex.

This partitioning is at the heart of the structured design methodology. The methodology starts with understanding the flow of data within the application—how various inputs are taken and then processed through stages, and how desired outputs are produced. For this understanding the methodology suggests the use of data flow diagrams. Most applications have basic transformations that perform the required operations. However, in most cases, the transformation cannot be easily applied to the actual physical input and produce the desired physical output. Instead, the input is first converted into a data structure that the transformation can easily handle. Similarly, the main transformation modules often produce outputs that need to be converted into the desired physical output.

In the next step, the central or main transformations performed on the data are identified, along with the inputs they need and outputs they produce. Generally, these transformations work with *the most abstract input data elements*—data elements that are obtained after cleaning, error checking, validating, etc. the actual input data, and suitably organized in appropriate data structures for processing. Similarly, the central transforms produce the *most abstract output data elements*—data produced by the computations and kept in internal data structures used for computation. Though this is the logical output of whatever computation the application was to perform, for output to the user, it may need reorganization and formatting. The *central transforms* perform the basic transformation, taking the most abstract input and transforming it into the most abstract output. This allows central transform modules to focus on performing the transformation without being concerned with converting the data into proper format, validating the data, and so forth.

Having identified the central transforms and the most abstract input and output data items, an initial structure chart design can be developed—a main module whose purpose is to invoke the subordinates, a subordinate input module for each of the most abstract input data items, a subordinate output module for each of the most abstract output data items, and a subordinate module for each central transform. These transform modules accept data from the main module, which obtains it from the input modules, and return the computed data back to the main module, which passes it to some output modules for eventual output. The main module is the overall control module, that invokes the input modules to get the most abstract data items, passes these to the appropriate transform modules, and delivers the results of the transform modules to other transform modules until the most abstract data items are obtained. These are then passed to the output modules.

The initial structure chart may have subordinate modules which have a lot of processing to do. These modules may then be factored into subordinate modules that will distribute the work of a module. Each of the input, output, and transformation modules must be considered for factoring. The process of factoring an input module is similar to the method discussed above—the main transformation being done to produce the most abstract input is identified, which becomes the transform sub-module. The inputs for this sub-module are obtained from input sub-modules. Essentially, through a hierarchy of transformations, the raw input is converted into the most abstract input which is used by the central transforms. The factoring of the output modules is symmetrical to the factoring of the input modules.

Factoring of transform modules can be done using step-wise refinement, identifying sub-transforms that can be composed together to perform the operations of the central transform.

An Example

Consider the problem of developing a component that will determine the number of different words in an input file (this may be part of an application to analyze text of classics). This problem has only one input data stream—the input file—while the desired output is the count of different words in the file. To transform the input to the desired output, the first step is to form a list of all the words in the file. It is best to then sort the list, as this makes identifying different words easier. This sorted list is then used to count the number of different words, and the output of this transform is the desired count, which is then printed. This sequence of data transformation is what we have in the data flow diagram.

The structure chart after the first-level factoring of the word counting problem is shown in Fig. 5.3.

In this structure, there is one input module, which returns the sorted word list to the main module. The output module takes from the main module the value of the count. There is only one central transform in this example, and a module is drawn for that. Note that the data items travelling to and from this transformation module are the same as the data items going in and out of the central transform.

The input module, get-sorted-list can be factored further by having an input module to get the list and a transform module to sort the list. The input module can be factored further into an input module for getting a word and another module for adding it to the list.

The central transform module count-the-number-of-different-words, can also be factored further by having a module to get a word repeatedly, a module to determine if it is the same as the previous word (for a sorted list, this check is sufficient to determine if the word is different from other words), and another to count the word if it is different.

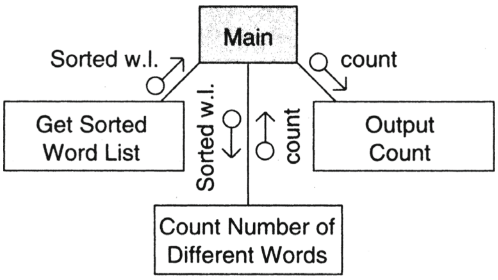


Fig. 5.3 First-level factoring

5.2.2 UML and Object-Oriented Design

Object-oriented (OO) approaches for software development have become extremely popular in recent years. The object-oriented design approach is different from the function-oriented design approaches primarily due to the different module abstraction used. In this discussion, we will assume that the reader is familiar with the basic OO concepts of classes and objects, object attributes and methods, and inheritance. In this section, we will briefly discuss some key aspects of the UML notation, which is widely used for OO design and is also useful for modeling other aspects (e.g. architecture, deployment). Then we will discuss a methodology for object-oriented design.

Unified modeling language (UML) is a graphical notation for expressing object-oriented designs [6]. For an OO design, a specification of the classes that exist in the system might suffice. However, during the design process, the designer may also try to understand how the different classes are related and how they interact to provide the desired functionality. Though UML has now evolved into a fairly comprehensive and large modeling notation, we will focus on a few central concepts and notations related to classes, their relationships, and interactions. For a more detailed discussion on UML, the reader is referred to [6].

Class Diagrams

The class diagram in UML serves as a pivotal element in system design or model. As the name suggests, these diagrams describe the classes within the design. As classes from the foundation of OO implementations, these diagrams establish a desired correlation with the final code. A class diagram defines

1. *Classes that exist in the application*—besides the class name, diagrams describe key fields and important methods of the classes.
2. *Associations between classes*—the types of associations that exist between different classes.
3. *Subtype, supertype relationship*—classes may form subtypes creating type hierarchies using polymorphism. Class diagrams can represent these hierarchies as well.

A class itself is represented as a rectangular box divided into three areas. The top section gives the class name. The middle section lists the key attributes or fields of the class, which are the state holders for the objects of the class. The bottom section lists the methods or operations of the class representing the behavior that the class can provide.

The divided-box notation describes the key features of a class as a stand alone entity. However, classes also have relationships between them, and objects of different classes interact. Therefore, to model an application, we must represent these relationship. One common relationship is the generalization-specialization relationship reflected as an inheritance hierarchy. In this hierarchy, properties of general significance are assigned to a more general class—the superclass—while properties which can specialize an object further are put in repetitive subclasses. All properties of the superclass are inherited by a subclass, so a subclass contains its own properties as well as those of the superclass.

The generalization-specialization relationship is shown with arrows coming from the subclass to the superclass, with an empty triangle shaped arrow-head touching the superclass. Often, when multiple subclasses exist, one arrow head on the superclass, may connect the lines leading to the subclasses. In this hierarchy, specialization often occurs based on a *discriminator*—a distinguishing property that is used to specialize superclass into different subclasses. In other words, by using the discriminator, objects of the superclass type are partitioned into sets of objects of different subclass types. The discriminator used for the generalization-specialization relationship can be labelled on the arrow. An example of how this relationship is modeled in UML is shown in Fig. 5.4.

In this example, the `IITKPerson` class represents all people at IITK. These are broadly divided into two sub-classes—`Student` and `Employee`, as they have different properties (some common ones also) and behaviors. Similarly, students have two different sub-classes, `UnderGraduate` and `PostGraduate`, each requiring different attributes and constraints. The `Employee` class has sub-types representing the faculty, staff, and research staff. (This hierarchy is from an actual working system developed for the Institute.)

Besides the generalization-specialization relationship, another common relationship is association, which allows objects to communicate with each other. An association between two classes means that an object of one class needs services from objects of the another class to perform its own service. The relationship is that of peers in that objects of both the classes can use services of the other. The association is shown by a line between the two classes and may include a name which can be

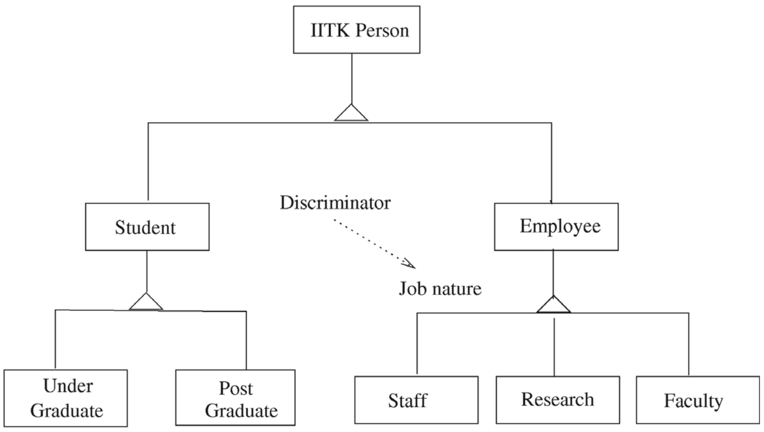


Fig. 5.4 A class hierarchy

specified by labeling the association line. (The association can also be assigned some attributes of its own.) And the roles of the two ends of the association can also be named. In an association, an end may also have multiplicity allowing relationships like 1 to 1, 1 to many, etc to be modeled. Where there is a fixed multiplicity, it is represented by putting a number at that end; a zero or many multiplicity is represented by a '*'.

Another type of relationship is the part-whole relationship which represents the situation when an object is composed of many parts, each part itself being an object. This situation represents containment or aggregation, i.e., object of a class is contained inside the object of another class. (Containment and aggregation can be treated separately and shown differently, but we will consider them as the same.) For representing this aggregation relationship, the class which represents the "whole" is shown at the top and a line emanates from a little diamond connecting it to classes which represent the parts. This relationship is often modelled as an association in implementations.

Sequence and Collaboration Diagrams

Class diagrams represent the static structure of the system, detailing the code structure and class relationships. However, they do not capture the dynamic behavior of the system. i.e how the system performs its functions. This dynamic behaviour is represented by *sequence diagrams* or *collaboration diagrams*, collectively called *interaction diagrams*. An interaction diagram typically captures the behavior of a use case and models how the different objects in the system collaborate to implement the use case. Let us discuss sequence diagrams, which is perhaps more common of the two interaction diagrams.

A sequence diagram depicts the series of messages exchanged between objects, and their temporal ordering, when objects collaborate to provide some desired system functionality (or implement a use case.) Sequence diagrams are typically drawn to model the interaction between objects for a particular use case. Note that, unlike class diagrams, sequence diagrams focus on objects, not classes as objects are the active participants during execution.

In a sequence diagram, participating objects (and their types) are shown at the top as boxes with object names. For each object, a vertical bar representing its lifeline is drawn downwards. Messages from one object to another are represented as arrows from the lifeline of one to the lifeline of the other. Each message is labeled with the message name, which typically should be the name of a method in the class of the target object. An object can also make a self call, which is shown as message starting and ending in the same object's lifeline. To clarify the sequence of messages and relative timing of each, time is represented as increasing as one moves farther away downwards from the object name in the object life. That is, time is represented by the y-axis, increasing downwards.

Each message has a return, which is when the operation finishes and returns the value (if any) to the invoking object. Though often this message can be implied, sometimes it may be desirable to show the return message explicitly. A sequence diagram

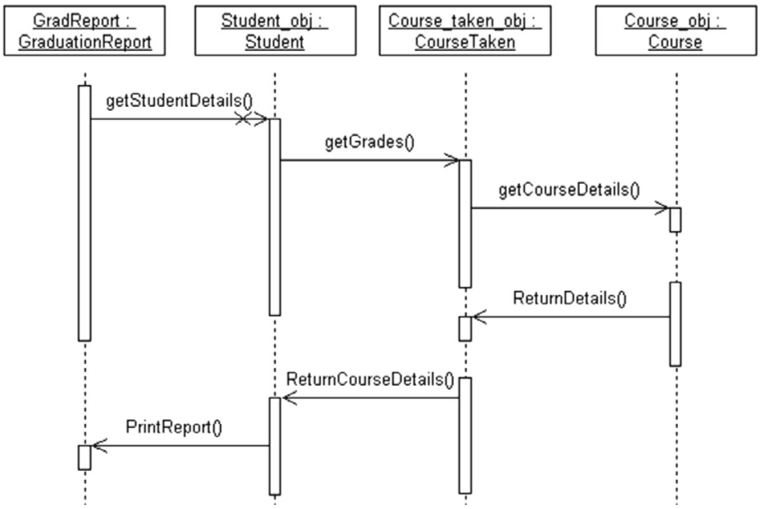


Fig. 5.5 Sequence diagram for printing a graduation report

for an example is shown in Fig. 5.5. This example is for printing the graduation report for students. The object for `GradReport` (which has the responsibility for printing the report) sends a message to the `Student` objects for the relevant information, which request the `CourseTaken` objects for the courses the students have taken. These objects get information about the courses from the `Course` objects.

Collaboration diagrams also show how objects communicate, but instead of using a time-line based representation that is used by sequence diagrams, it looks more like a state diagram. Each object is represented in the diagram, and the messages sent from one object to another are shown as *numbered* arrows from one object to the other. In other words, the chronological ordering of messages is captured by message numbering, in contrast to a sequence diagram where ordering of messages is shown pictorially. As should be clear, the two types of interaction diagrams are semantically equivalent and have the same representation power. However, over the years, sequence diagrams have become more popular, as people find the visual representation of sequencing quicker to grasp.

As we can see, Interaction diagrams model the internal dynamic behavior of the system, when it performs some function. The internal dynamics of the system is represented in terms of how the objects interact with each other. Through interaction diagrams, one can clearly see how a system internally implements an operation, and what messages are sent between different objects. If a convincing interaction diagram cannot be constructed for a system operation with the classes that have been identified in the class diagram, then it is safe to say that the system structure is not capable of supporting this operation and that it must be enhanced. Therefore, it is also used to validate if the system structure being designed through class diagrams is capable of providing the desired services.

Since a system has many functions, involving different objects in various ways, a dynamic model is needed for each of these functions or use cases. In other words, whereas one class diagram can capture the structure of the system's code, many diagrams are needed for the dynamic behavior. However, it may not be feasible or practical to draw the interaction diagram for each use case scenario. Typically, interaction diagrams for key use cases or functions are be drawn during design to ensure that the classes can indeed support the desired use cases, and to understand their dynamics.

Other Diagrams and Capabilities

In modeling and building systems, as we have seen, components may also be used. They encapsulate "larger" elements, and are semantically simpler than classes. Components often encapsulate sub-systems and provide clearly defined interfaces through which the other components can use them. As we have seen, components are very useful while designing an architecture. UML provides a notation for specifying a component. UML also offers separate notation for a sub-system, which is an identified part of a system. In a large system, many classes may be combined together to form packages—a collection of many elements, possibly of different types. UML also provides a notation to specify packages.

In the chapter on Architecture we briefly mentioned that there is a deployment view of the system, which may be quite different from the component or module view. The deployment view, focuses on what software element uses which hardware, that is, how is the system deployed. UML has notation for representing a deployment view. The main element is a *node*, represented as a named cube, which represents a computing resource like a CPU (which physically exists). The name of the cube identifies the resource as well as its type. Within the cube, the software elements it deploys (which can be components, packages, classes, etc.) are shown using their respective notation. If different nodes communicate with each other, it is shown by connecting the nodes with lines.

The notation for packages, deployment view etc, provide structural views of the system from different perspectives. UML also provides notation to express different types of behavior. A *state diagram* is a model in which the entity being modeled is viewed as a set of states, with transitions between the states taking place when an event occurs. A state is represented as a rectangle with rounded edges or as ellipses or circles; transitions are represented by arrows connecting two states. Details can also be attached to transitions. State diagrams are often used to model the behavior of objects of a class—the state represents the different states of the object and transition captures the performing of the different operations on that object. So, whereas interaction diagrams capture how objects collaborate, a state diagram models how an object itself evolves as operations are performed on it. This can help clearly understand and specify the behavior of a class.

Activity Diagrams is another diagram for modeling dynamic behavior. These diagrams aim to model a system by modeling the activities that take place in it when the system executes for performing some function. Each activity is represented as an oval, with the name of the activity within it. From the activity, the system proceeds to other activities. Often, which activity to perform next depends on some decision. This decision is shown as a diamond leading to multiple activities (which are the options for this decision.) Repeated execution of some activities can also be shown. These diagrams are like flow charts, but also have notation to specify parallel execution of activities in a system by specifying an activity splitting into multiple activities or many activities joining (synchronizing) after their completion.

Use case diagrams are also a part of the UML. We discussed use cases in an earlier chapter. In a use case diagram, each use case is shown as a node, and relationship between actors and use cases as arcs. They are mostly used to provide a high-level summary of use cases.

A Design Methodology

Many OO design and analysis methodologies have been proposed. As we stated earlier, a methodology uses concepts (of OO in this case) to provide guidelines for the design activity. We briefly discuss some aspects of one of the earlier methodologies [7].

The starting point of this methodology is identifying the classes and their relationships, which requires identification of object types in the problem domain, the structures between classes (both inheritance and aggregation), attributes of the different classes, associations between the different classes, and the services each class needs to provide to support the system. Basically, in this step we aim to define the initial class diagram of the design.

To identify classes, start by looking at entities mentioned in the use cases or in component descriptions and consider if a class definition is needed for each entity. Include an entity as a class if the system needs to remember something about it, or needs some services from it to perform its own services. To identify attributes, consider each class and see which attributes are needed by the problem domain.

To identify the class structure, consider the classes that have been identified as a generalization and see if there are other classes that can be considered as its specializations. Ensure that the specializations are meaningful for the problem domain.

To identify assembly structure, consider each object of a class as an assembly and identify its parts. Determine if the system needs to keep track of the parts as separate objects. If it does, then the parts must be reflected as objects; if not, then the parts should not be modeled as separate objects.

For associations we need to identify the relationship between instances of various classes. The associations between objects are derived from the problem domain directly once the objects have been identified.

The class diagram obtained gives the initial module-level design. This design will be further shaped by the events in the system, as the design must ensure that the expected behavior can be supported. Modeling the dynamic behavior of the system further helps in further refining the design.

The dynamic model of a component aims to specify how the state of various objects changes when events occur. A scenario is a sequence of events that occur in a particular execution of the component or system—generally when a use case is executed by the user. From the scenarios, the different events being performed on different objects can be identified, which are then used to identify services/methods on objects. If the design is such that it can support all the scenarios, we can be sure that the desired dynamic behavior of the system can be supported by the design. This is the basic reason for performing dynamic modeling.

Once the main use cases are modeled, various events on objects that are needed to support executions corresponding to the various scenarios are identified. This information is then used to expand our view of the classes in the design. These steps together can suffice to get a decent OO design for a component. However, for more complex systems the methodology suggests some more steps, like functional modeling, identifying internal classes, and optimizing the design. The reader is referred to books on OO modeling for further information on these.

Examples

Let us first consider the word counting problem discussed earlier as an example for Structured Design methodology. The initial analysis clearly shows that there is a `File` object, which is an aggregation of many `Word` objects. Further, one can consider that there is a `Counter` object, which keeps track of the number of different words. It is a matter of preference and opinion whether `Counter` should be an implemented as an object, or counting should be implemented as an operation. If counting is treated as an operation, we will ponder upon, to which object it belongs. As it does not belong "naturally" to either the class `Word` or the class `File`, it will have to be "forced" into one of the classes. For this reason, we keep `Counter` as a separate object. The basic problem statement finds only these three objects. However, further analysis reveals the need for some history mechanism to check if the word is unique. The class diagram obtained after doing the initial modeling is shown in Fig. 5.6.

Now let us consider the dynamic modeling for this problem. This is essentially a batch processing problem, where a file is given as input and some output is given by the system. Hence, the use case and scenario for this problem are straightforward. For example, the scenario for the "normal" case can be:

- System prompts for the file name; user enters the file name.
- System checks for existence of the file.
- System reads the words from the file.
- System prints the count.

From this simple scenario, no new operations are uncovered, and our class diagram remains unchanged. This design was specified and later implemented in C++. The conversion of design to code required only minor additions and modifications to the design. The final code was about 240 lines of C++ code (excluding comments and blank lines).

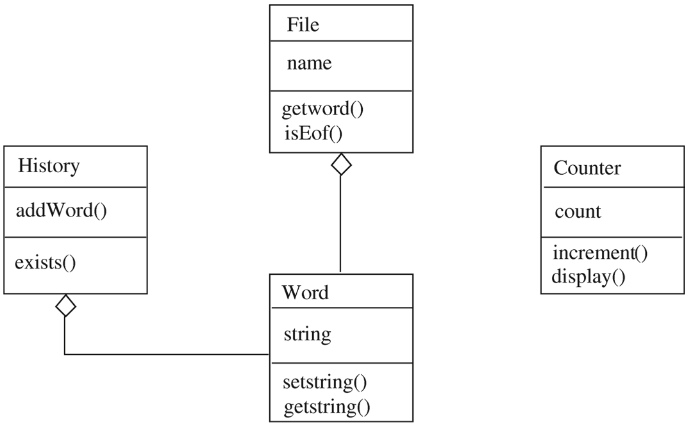


Fig. 5.6 Class diagram for the word counting problem

Let us consider a slightly larger problem: of determining the rate of returns on investments. An investor has made investments in several companies. For each investment, in a file, the name of the company, all the money he has invested (in the initial purchase as well as in subsequent purchases), and all the money he has withdrawn (through sale of shares or dividends) are given, along with the dates of each transaction. The current value of the investment is given at the end, along with the date. The goal is to find the rate of return for each investment, as well as for the entire portfolio, along with total amount invested initially, amounts invested subsequently, amounts withdrawn, and the current value of the portfolio.

This is a practical problem that is frequently encountered by investors, and a rate of returns calculator can easily form an important component of a larger financial management system. The computation of rate of return is not straightforward and cannot be easily done through spreadsheets. Hence, such an application can be of practical use.

Initial problem analysis easily throws up a few object classes of interest—`Portfolio, Investment`, and `Transaction`. A portfolio consists of many investments, and an investment consists of many transactions. Hence, the class `Portfolio` is an aggregation of many `Investments`, and an `Investment` is an aggregation of many `Transactions`. A transaction can be of `Withdrawal` type or `Deposit` type, resulting in a class hierarchy, with `Transaction` being the superclass and `Withdrawal` and `Deposit` subclasses.

For an object of class `Investment`, the major operation we need to perform is to find the rate of return. For the class `Portfolio` we need to have operations to compute rate of return, total initial investment, total withdrawal, and total current value of the portfolio. Hence, we need operations for these. The class diagram obtained from analysis of the problem is shown in Fig. 5.7.

In this problem, the interaction with the environment is minimal, so the dynamic model is not significant and can be omitted.

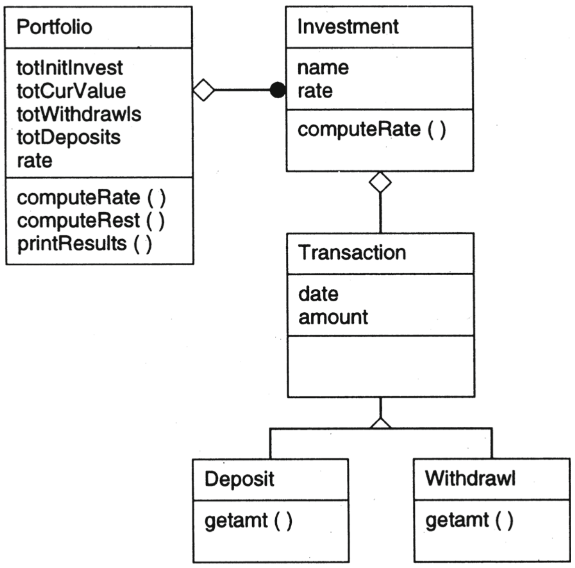


Fig. 5.7 Class diagram for rate of return problem

Now we can optimize the design from the point of view of implementation concerns. While considering the implementation of computation of total initial investment, computation of overall return rate, overall withdrawals, and so on, we notice that for all of these calculations, appropriate data from each investment is required. Hence, to the class `Investments`, appropriate operations need to be added. Further, we note that all the computations for total initial investment, total current value, and so on are all done together, and each of these is essentially adding values from various investments. Hence, we combine them in a single operation in `Portfolio` and a corresponding single operation in `Investment`. Studying the class hierarchy, we observe that the only difference in the two subclasses `Withdrawal` and `Deposit` is that in one case the amount is subtracted and in the other it is added. In such a situation, the two types can be easily considered a single type by keeping the amount as negative for a withdrawal and positive for a deposit. So we remove the subclasses, thereby simplifying the design for implementation.

The final design was implemented in C++ code, and we found that only minor details got added during the implementation, showing the correctness and completeness of the design. The final size of the program was about 470 lines of C++ code (counting noncomment and nonblank lines only).

5.3 Designing an Application

Most applications can be conceptualised as having a front-end component, which has code that interacts with the end users for getting inputs and for showing outputs, and a back-end component, which has code that is called by the front-end code to get some processing done and obtain the subsequent results. Distributed applications have very clearly demarcated and separate front-end and back-end code. But stand-alone applications also require code for input/output with users, to process those inputs, which should ideally be separated for clarity and modularity.

And if an application has persistent data, which is common in most applications, the back-end code must access and update this data to serve the requests of the front-end. These persistent data stores, typically databases, have defined methods for reading and writing/updating the data, and on reading, provide data in specific predefined formats, and for writing, require data to be in defined formats. Code modules are needed to interact with the persistent data store which the back-end code can call whenever data retrieval or updating is necessary. It is often beneficial to organize these modules into as a layer, sometimes referred to as the model layer.

Databases typically operate as a separate process containing many functions within themselves. This naturally makes the database a separate component. We will not delve into the design of databases here and the reader is referred to any standard text for it.

Therefore, for designing an application, we can divide the problem into designing the front-end, designing the back-end and designing the model layer. We discuss each of these further. For the discussion, we will assume a distributed application.

5.3.1 Front-End Design

Front-end of an application can be seen as a set of screens that the user interacts with. An input screen typically contains multiple components, each designed to gather or display specific information. Each of the screens can be viewed as a module in the front-end design. An input module collects the information from the user, and passes the information to some backend module for computing which returns the desired information. (In a stand-alone application the call to the backend module will be a direct function/method call; in a distributed application it involves invoking an API; we will refer to both as calling an API.) In most web applications, the API responds by sending an output screen (e.g. as a html page) to be displayed, or information in some format (e.g. JSON) for updating the displayed page.

A front-end module can be viewed as an input screen comprising various components (e.g. an image, a table, a box for getting the name, a box for getting telephone number, etc). Each front-end module may call some back-end APIs for performing the desired operation. Thus, the front-end design can be specified as a table of front end modules of the type shown in Fig. 5.8.

Fig. 5.8 Front-end design

No.	Name	Purpose	Main Components	APIs called
1				
2				
..				

Name should reflect the operation the user wants to perform, and the purpose describes it briefly. Main components list the components this screen contains, and the APIs this module calls specifies the functionality from backend this module expects.

Designing these screens from scratch can be cumbersome and time consuming. Fortunately numerous open source component libraries are available to expedite this process. Some examples are: MaterialUI, ChakraUI, TailwindCSS, HeadlessUI. These libraries provide components like boxes, grids, cards (to contain some content and actions), navigation drawer (for menus), rich functionality tables, autocomplete, etc., which can be used by a designer for designing the screens.

Users also need to navigate between the different screens within an application. While a simplistic design may feature a "home screen" from where any screen can be accessed, most graphical user interfaces (GUIs) allow users to move from one input screen to a limited set of screens. This navigation can be illustrated as a graph by having each screen as a node and a link from a node A to B if the screen for A has provision for the user to go to screen B. (There are special navigation libraries like React Router, Next Router, etc, which can make navigation more efficient by keeping some portions of the previous screen and only fetching the necessary portions.)

When the user moves from one input screen to another, in distributed applications like those based on REST design, there is no direct mechanism to pass information collected on the previous screen(s) to the next, as each new screen is supposed to be fresh. This can be cumbersome for many applications where information collected in previous screens need to be carried over and be available to the new screen (e.g. as pre populated components showing the information, or for using past information for validation, etc.). To facilitate sharing state information across screens, special state management libraries are available e.g. Redux, VueX, etc.

It is advisable to specify all these elements in the front-end design. Therefore, besides the list of modules in the front-end, the design should also specify the component libraries being used.

In a distributed web-based application, the front-end modules may be kept in a separate front-end server. This decouples the front-end from the back-end, optimizing performance for the front-end.

5.3.2 API-Based Back-End Design

The front-end modules get the desired inputs from the user for performing the requested operation. To carry out the operation and getting the desired output, some computations are required. Generally, these computations are done at the backend (except some data checks which may be done in the code for the frontend).

No.	API Name	Brief Description	Inputs it needs	Outputs / Response
1				
2				
..				

Fig. 5.9 Back-end design

To simplify the interaction between the frontend code and backend code, the backend code can be viewed as providing a set of APIs (application programming interface) which can be called by the frontend modules. If the application is a standalone application, this API call is simply function/method call in the programming language. In distributed applications, invoking a back-end API function requires making a remote API call, typically facilitated by specific libraries.

An APIs can be specified by providing an expressive name, a brief description of its functionality, the inputs it requires, and the outputs it provides. Further details can be provided, if desired. The backend design can be summarized as a table of the type shown in Fig. 5.9.

The back-end API design can be more formally specified using OpenAPI (https://www.openapis.org/), which is an open source language that encodes information about the API (information contained in this table and some additional information) in JSON format with precise information about each API. In OpenAPI an API is specified in a structured manner containing information like metadata (e.g. title, version), servers where API may be hosted, paths to the API endpoints, and for each endpoint parameters and where they are embedded in the request (e.g. as path parameters, query parameters, or in the body), response structure, security information (schemes a request must contain to be authorized) and tags (used to group api resources). This specification is machine processable, and tools are available (some from OpenAPI itself) for various tasks such as validation (e.g. check to see if API requests and responses are consistent with the API description), generating module headers, generating the API Description as HTML and creating test scripts, etc.

Programming the backend from scratch in a distributed application involves handling requests, routing them to the correct API, converting the results produced by the API to a suitable format for the front end (e.g. an html page), as well as handling other common functions relating to connection between the frontend and backend. Open-source back-end frameworks provide many of these facilities, only requiring the programmer to specify the particular code for the APIs. Some common open source frameworks for back end are Django, NodeJS, NestJS, Ruby on Rails. With these frameworks, programmers need to provide the code for implementing the APIs, as well as the routing information on what function/API to call when a request comes from FE.

With a framework, a request from the frontend comes to the framework (which typically is running as a process). The framework code looks at the routing information (urls.py in Django and routes.js in NodeJS) to determine which API function to call (which resides in views.py in Django and controller.js in NodeJS), and invokes the function. This function then calls the functions/methods in models (models.py in

Djanjgo, models.js in NodeJS) to perform the queries or operations on the database and return suitable values. These values populate a template (which is an HTML document with some fields which are populated with the data being provided), and then returned back as response to the browser. Therefore, besides specifying the API it supports, the back-end design should also include:

- The back-end framework being used.
- Libraries to be used for specific functionality like authentication, security, and access control.
- External services that may be used. Some of the APIs may require external services (e.g. google oauth), if so, that should be mentioned.

5.3.3 Model Layer Design

The back-end APIs often require data from databases which are structured to efficiently store data needed for the application, in tables and other forms for easy and quick access. Tabular databases provide commands to define the schema, create the database tables, and SQL commands to work with the tables, and return the results in specified format like JSON. (NoSQL databases like MongoDB will provide their own operations and language.) Access to these databases by programs is facilitated by specially designed connector modules that can be called by a program, which send queries/operations to the database server and returns results to the caller.

Processing data retrieved from databases generally involves organizing it into data structures that can be processed efficiently by modules in the backend code, e.g. in the internal data structures provided by the language in which the back-end modules are written, like lists, dictionaries, sets, or as objects of user defined classes, etc. For instance, the data retrieved in format like JSON needs to be converted into dictionaries, lists, objects, etc., and then passed to the modules in the back-end code for necessary processing. Similarly, the information produced by the back-end modules which need to be saved in the data store will be in data structures being used by those programs (lists, dictionaries, objects, etc). For saving them in persistent storage they need to be converted into suitable form like strings or JSON, which can then be passed to the database.

Access to the database and this back-and-forth conversion is a non-trivial task and it is desirable to combine these modules into a separate folder, often called the the model layer. The model layer modules are part of the back-end code but are grouped together (e.g. in a directory) to provide a clear view of all the data layer access/interaction modules. This layer has code to define the database schema and create the tables, which it provides as objects to the APIs. APIs can perform operations on these objects, which are suitably transformed into operations on the database.

Model layer design typically involves defining classes and methods for creating database tables, querying a database (e.g. search), or performing operations (e.g. update). For connecting to the database, making these requests and obtaining the results, interface libraries are generally provided (e.g. for MongoDB mongoose for NodeJS or pymongo for Django), which facilitate the task of converting data back and forth and perform the requested operations. Detailed database design specifics are outside the scope of this book and the readers are encouraged to standard database textbooks for further guidance.

5.4 An Example

Let us consider the student-event portal application, whose requirements and architecture design have been discussed in earlier chapters. We will briefly discuss the design of this application—the front-end design, the back-end design, and the database design. Further details about the design are given on the website.

Front-End Design

The application follows a variation of the MVC architecture, where the front-end is a set of pages with JavaScript calls to the back-end APIs (which return JSON objects) and links to other pages. From the requirements, it is clear that we need only a few front-end screens/pages: (i) Home page, which will have links to other pages for getting information about events. (ii) Events page, which display information about the events and will provide link for form to register for one. (iii) Clubs page, which will have links for showing information about clubs, and links for registering a new club, changing a club's information, proposing an event, etc. (iv) User profile page, which will display users profile, registered events, etc., and for admin will also show pending event requests etc. (v) Login page, for a user to login.

Each of these pages consists of components to provide the necessary services and information. A page may invoke back-end APIs to fulfill its request. For example, the Events page, which provides for registering for an event, will call the relevant API to authenticate the user, and to register the user for an event. These pages, along with their components and the APIs they call are given in the Table 5.2 (A common component is the navigation-bar, which provides common navigation in all screens—it is not listed in the table to avoid cluttering.)

Back-End API's Design

The front-end design focuses on the user interface and the operations they can perform. It also specifies the APIs needed to support these operations. The back-end design involves creating these APIs—specify what inputs they need and what will they return. The back-end design for the application is given in Table 5.3. Note that while it may seem that the back-end design just elaborates the APIs already specified

Table 5.2 Main user screens in the student event application

Name	Main components	APIs required
Event	**Cards:** Displaying event information. **Forms:** To register for events	AuthenticateUser, Get Event Details, RegisterForEvent
Home	**Cards:** List of Upcoming Events. **Pages:** Event details page	GetAllEvents
Clubs (includes club single info page)	**Cards:** Displaying club information. Forms: To register new clubs, edit an existing club's details, propose a new event or edit an existing event's details **Pages:** Club details page	AuthenticateUser, GetAllClubs, CreateClubRequest, UpdateClubDetails, getAboutInfo, CreateEventRequest
Profile (Includes Manage)	**Cards:** Displaying user profile information, List of registered events, List of event proposal status, List of club proposal status, List of pending event proposals with approve and reject action (for admin only), List of pending club proposals with approve and reject action (for admin only). **Forms:** Approve/Reject Event and Club Proposals, Change Admin (for admin only)	UserProfile, ApproveEventRequest, RejectEventRequest, ApproveClubRequest, RejectClubRequest
Login (includes login success)	**Forms:** GoogleAuthLogin. **Pages:** Login Success Page (for redirecting)	OAuthCallback

in the front-end design, often the front-end and back-end designers go back-and-forth between themselves to identify the APIs that are desirable to have at the back-end and ones that can serve the front-end needs.

Database Design

The database design for this application is relatively straightforward with tables for users, clubs, and events. Each table keeping various attributes for each entity.

In an MVC style, the database schema is created through suitable calls from the models layer of the back-end code. Suitable libraries are used to interact with the database to create these tables, and then pass the tables as objects to the code that needs to perform operations on the database.

Table 5.3 APIs in the back-end design

API name (description)	Inputs	Response
OAuthCallback	User credentials	Valid/Not Valid, Access Token
GetAllClubs	None	Returns a list of all clubs
CreateClubRequest	Access Token, Club data	Submit message
GetAboutInfo	Access Token, Club ID	Club data
ApproveClubRequest	Access Token, Club Proposal ID	Success message
RejectClubRequest	Access Token, Club Proposal ID	Decline message
UpdateClubDetails	Access Token, Club ID, Club data	Success message
GetAllEvents	None	Array of all upcoming events
EventInfo	Access Token, Event ID	Event data
CreateEventRequest	Access Token, Event data	Submit Message
ApproveEventRequest	Access Token, Event Proposal ID	Success message
RejectEventRequest	Access Token, Event Proposal ID	Decline message
RegisterForEvent	Access Token, Event ID	Success message/Decline message
UpdateAdmin (to be implemented later)	Access Token, User ID	Success message/Decline message

5.5 Using LLMs for Design

We have discussed approaches for developing and specifying a high level design of an application. Given the capabilities of LLMs, it seems natural to leverage them to produce the desired design specification for a project. Here, we will briefly discuss practices for using LLMs in designing an applications whose architecture have a front-end component, a back-end component, and a component to interact with the database. As previously discussed, while distributed applications easily fit into this architecture, even stand-alone applications can be conceptualized with these components—only the connectors between the different layers are different.

In this type of application design, as discussed earlier, the frontend is specified as a set of screens with navigation between them. The backend is specified as a set of APIs which the frontend can call, and the model layer provides functions to the backend for making database queries.

When using an LLM to generate design specifications for an application, it is crucial to provide the requirements for the application, (i.e. the SRS). This input informs the LLM about the various use cases, design constraints and other requirements. To ensure that the design is also consistent with the chosen architecture, it is advisable to provide the LLM with architecture specifications as well, or the key architectural decisions.

For architectures with front-end, back-end, and a model layer, we can provide the LLM with the SRS and specify the complete output structure, and then ask the LLM to generate the design specifications using a zero-shot approach. Experiments suggest that this often does not work well. Instead, we recommend to initially prompt the LLM to generate the back-end design, review it for corrections if needed, then proceed with the front-end design. *We then check the consistency of front-end design with the back-end design before moving onto the model layer and the database design.* That is, prompt chaining with chain-of-thought is a good way to prompt the LLM to produce good quality designs. For further explanation, an example elaborating the design process is provided below.

Using the feature chatGPT provides to specify the context and desired output, in the box: "What would you like ChatGPT to know about you to provide better responses?", the following can be given:

> The task is to develop the software design document (SDD) for an application whose software requirements specification (SRS) is given. In SDD the modules that should be there in different components of the application architecture need to be specified precisely. The most important characteristics of a good design is that it should be correct, which means that implementing the modules properly will lead to an implementation that will satisfy the requirements stated in the SRS. If the design cannot satisfy some requirements, that should be reported.

And in the box "How would you like ChatGPT to respond" the structure of the SDD document can be provided. The output structure should clearly specify that the design must have separate modules for front-end, back-end, and model-layer. With this context and output structure, prompting can proceed as follows:

> Prompt: Given is the following Software Requirements Specification (SRS) for a <Problem Statement or Descriptive Title>, as enclosed within triple quotes:
>
> "' ...SRS "'
>
> Please generate only the back-end design for this application, as per the structure given in the template for SDD.

After this, the reply can be corrected through subsequent prompts. (For instance, in one experiment, chatGPT only provided the APIs without numbering them or giving each API a name, and a prompt was given to assign a number and an expressive name for each of the APIs, which it did.) Once the API design is satisfactory, the design process can proceed to the front-end. (It should be noted that the front-end design is done after the back-end design, as specification of the front-end design requires specifying the APIs that are called.)

> Prompt: For the SRS given earlier and for which you have done the back-end design, please specify the front-end design in the format given earlier as part of the SDD structure.

Experiments indicate that this method of prompting works well with the LLMs generating decent quality designs. Furthermore, the sequence of back-end design first followed by front-end design and model layer design works better than doing the front-end design first. It is crucial to emphasize that the final design remains the responsibility of the designer; LLM is just a tool to help the designer.

The back-end design can also be readily converted into detailed OpenAPI specifications using LLMs, if desired. Experiments demonstrate that chatGPT can generate good quality OpenAPI design specifications facilitating the initiation of application coding.

LLMs can also be used to help in verifying some aspects of a design. For example, they can check the consistency between front-end design and back-end design. By examining the SRS, the front-end design and the back-end design can be checked for coherence. Experiments suggest that they are able to detect consistency errors (e.g. front-end calling an API that is not there in the back-end specifications).

We have discussed here designing for a general 3-tier application. LLMs are likely to be useful for designing for various architecture styles. General approach mentioned above can be tried for other styles.

5.6 Summary

- The design of a system is a plan for a solution, that when implemented, will satisfy the requirements of the system and preserve its architecture. Module-level design specifies the modules that should be there in the system to implement the architecture.
- A system is modular if each module has a well-defined abstraction and changes in one module have minimal impact on others. Two criteria used for evaluating the modularity of a design are *coupling* and *cohesion*. Coupling reflects module interdependence, while cohesion is a measure of the strength of relationship within a module. In general, good design minimizes coupling and maximises cohesion. It should also adhere to the Open-Closed principle, where modules are open for extension but closed for modification.
- A structure chart for a procedural system represents the modules in the system and the call relationship between them. Structured design methodology provides guidelines on how to create a design (represented as a structure chart) with low coupling and a high level of cohesion. It partitions the system at the very top level into various subsystems, one for managing each major input, one for managing each major output, and one for each major transformation. This cleanly partitions the system into parts, each part independently dealing with different concerns.

- The Unified Modeling Language (UML) includes various types of diagrams to model different properties of an object-oriented design. It allows both static structure as well as dynamic behavior to be modeled. Class diagrams represent static structure, by showing the classes in the system and relationships between these classes. For modeling the dynamic behavior, sequence diagrams can be used to show how a scenario is implemented by involving different objects. Using UML notation, object-oriented designs of a system can be created. The design methodology for evolving the design focuses on identifying classes and relationships between them, and validating class definitions using dynamic modeling.
- An approach for designing an application following the 3-tier architecture style is to design the front-end, the back-end, and the model layer separately. The front-end design specifies the user-facing screens, their components, and the back-end APIs they call. The back-end design details the APIs to be provided, with the inputs needed and the outputs returned specified for each API. The model layer design outlines the functions it provides to the APIs for interacting with the database.
- LLMs can be used as a tool to help in design. It may be suitable to first specify the application requirements and the structure of the desired design document. Then for a 3-tier application, prompt chaining can be employed by asking an LLM to design each of the three tiers. Experiments suggest that designing the back-end before moving onto designing the front-end is more efficient when using LLMs.

Self-assessment Exercises

1. What is the relationship between an architecture and module-level design?
2. Suppose you are given a design specification for an SRS. What criteria will you use to evaluate the quality of this design?
3. Consider a program containing many modules. If a global variable x must be used to share data between two modules A and B, how would you design the interfaces of these modules to minimize coupling?
4. A module is designed to compute the mean and median for a list of numbers. Write the interface of such a module, and explain why the module does not have the best cohesion and will not have the least coupling with caller modules. How will you improve the design to improve coupling and cohesion.
5. In a university, there are different types of people: students, faculty, and staff. When designing an application which will have some common facilities for all but some special facilities for each of them, how will you design the class(es) to represent people of the university such that open-closed principle is followed.
6. For the example design given above, for some user operations (or use cases) given in an earlier chapter, draw the sequence diagrams.
7. A python application is to be developed which gets data from the user through two screens (to be developed using UI libraries of python)—one asking for name and roll no and another for an input file in which some URLs are given. For bonafide students (assume that the application has access to a file which has list of bona fide student's records), it extracts the list of URLs from the file (if file exists and is in desired format), checks if the URL is valid, then for each URL it determines the average response time (e.g. by pinging and measuring time). After all the URLs are processed, it determines the mean, min, max of the response times over the list

of URLS, and then shows it back to the user in some format. If the application uses a two-tier architecture (a) Specify the front-end design for this application by listing the modules with a statement about what they do, and the back-end APIs they will call. (b) Specify the back-end design by listing the APIs that the backend code will support with their inputs and outputs.

8. For the student-event portal application example, whose SRS and architecture was given earlier, use an LLM to generate the design. Compare the generated design with the design given in the chapter.
9. Using the principles of modularity, evaluate the design generated, and identify opportunities to improve the modularity (even at the cost of reduced performance).
10. Define two metrics for quantifying complexity of an object-oriented design. How will you use one of them to identify highly-complex or error-prone modules?

References

1. W.P. Stevens, G.J. Myers, L. Constantine, Structured design. IBM Syst. J. **13**, 2 (1974)
2. E. Yourdon, L. Constantine, *Structured Design* (Prentice Hall, 1979)
3. J. Eder, G. Kappel, M. Schrefl, Coupling and cohesion in object-oriented systems, Technical report (University of Klagenfurt, 1994)
4. B. Meyer, *Object Oriented Software Construction* (Prentice Hall, 1988)
5. B. Liskov, Data abstraction and hierarchy. SIGPLAN Notices **23**, 5 (1988)
6. M. Fowler, *UML Distilled—A Brief Guide to the Standard Object Modeling Language* (Addison-Wesley Professional, 2003)
7. G. Booch, *Object-Oriented Analysis and Design with Applications*, Professional; Subsequent edition (Addison-Wesley, 1994)

6 Coding

The aim of coding or programming activity is to implement the design in the best possible manner. The coding activity profoundly affects both testing and maintenance. It is well known that testing and maintenance consume the maximum effort in an application—the effort spent in coding is a small percentage of the total cost. It should be clear that to achieve high productivity for the project, the goal during coding should not be to reduce the implementation cost but to help reduce the cost of testing and maintenance. During coding, it should be kept in mind that a program is read a lot more often and by many more people. Hence, they should be constructed to be easily read and understood.

Having readability and understandability as a clear objective of the coding activity can help achieve it. A famous experiment by Weinberg showed that if programmers are specified a clear objective for the program, they usually satisfy it [1]. In the experiment, five teams were given the same problem for which they had to develop programs. However, each team was specified a different objective, which it had to satisfy. The objectives given were: minimize the effort required to complete the program, minimize the number of statements, minimize the memory required, maximize the program clarity, and maximize the output clarity. It was found that, in most cases, each team did the best for the specified objective. The rank of the different teams for the different objectives is shown in Table 6.1.

The experiment shows that if objectives are clear, programmers tend to achieve that objective. Hence, if readability is an objective of the coding activity, programmers will develop easily understandable programs. It also shows that if the focus is on minimizing coding effort, program clarity takes a big hit. For our purposes, besides correctly implementing the desired functionality, ease of understanding and modification are the primary goals of the coding activity. High-quality code is that which achieves these goals.

P. Jalote, *A Concise Introduction to Software Engineering*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-74318-4_6

Table 6.1 The Weinberg experiment

	Resulting rank (1 = best)				
	O1	O2	O3	O4	O5
Minimize effort to complete (O1)	1	4	4	5	3
Minimize number of statements (O2)	2–3	1	2	3	5
Minimize memory required (O3)	5	2	1	4	4
Maximize program clarity (O4)	4	3	3	2	2
Maximize output clarity (O5)	2–3	5	5	1	1

In this chapter, we will discuss the following:

- Some concepts for writing high-quality code like structured programming, information hiding, pre and post-conditions, use of coding standards, and use of open source libraries.
- How the evolving code for an application should be structured in a repository, and how the evolution should be managed using source code control.
- Some programmer-level processes, like incremental and test-driven development, for efficiently developing high-quality code.
- How LLMs can be effectively used to help in coding.
- How to unit test modules using unit testing frameworks.
- How to review code using a one-person review or a structured code inspection process to improve the quality of the code.
- Some metrics related to code size and complexity.

6.1 Programming Principles and Guidelines

The main task before a programmer is to write readable code with few bugs. Writing high-quality code is a skill that can only be acquired by practice. However, some general rules and guidelines can be given to the programmer. In this section, we will discuss some concepts and practices that can help a programmer write high-quality code.

6.1.1 Structured Programming

The structured programming movement started in the 1970s; much has been said and written about it. Now, the concept pervades so much that it is generally accepted—even implied—that programming should be structured.

A program has a static structure as well as a dynamic structure. The static structure is the structure of the program's text, which is just a linear organization of program's statements. The dynamic structure of the program is the sequence of statements executed during the program's execution. It will be easier to understand the dynamic behavior if the structure in the dynamic behavior resembles the static structure. The closer the correspondence between execution and text structure, the easier the program is to understand, and the more different the structure during execution, the harder it will be to argue about the behavior of the program text. The goal of structured programming is to ensure that the static structure and the dynamic structures are the same. That is, the objective of structured programming is to write programs so that the sequence of statements executed during the execution of a program is the same as the sequence of statements in the text of that program. As the statements in a program text are linearly organized, the objective of structured programming becomes developing programs whose control flow during execution is linearized and follows the linear organization of the program text.

No meaningful program can be written as a sequence of simple statements without any branching or repetition (which also involves branching). Given this, the objective of linearizing the control flow is achieved using structured statements. Examples of structured statements are if-then-else, while and for loops. The critical property of a structured statement is that it has a *single-entry* and a *single-exit*. That is, during execution, the execution of the (structured) statement starts from one defined point and the execution terminates at one defined point. We can view a program as a sequence of (structured) statements with single entry and single exit statements. If all statements are structured, then during execution, the sequence of execution of these statements will be the same as their sequence in the program text.

A motivation for structured programming was the formal verification of programs. To formally prove that a program is correct, we need to show from the text of the program, that when the program executes, its behavior is what is expected. Having the the program as a sequence of structured constructs that are executed in the order in which they are included in the program text facilitates proving properties about the program behaviour from its text. Program verification is an advanced topic and is not discussed further in this text.

Hence, the correspondence between the static and dynamic structures can be obtained using single entry and exit statements. The most commonly used single-entry and single-exit statements are

Selection: if B then S1 else S2
if B then S1
Iteration: While B do S
repeat S until B
Sequencing: S1; S2; S3;...

It can be shown that these basic constructs are sufficient to program any conceivable algorithm. Modern languages have other such constructs that help linearize the control flow of a program. Programs should be written so that, as far as possible, single-entry, single-exit control constructs are used. The primary goal is to make the logic of the program simple to understand. Structured programming practice forms a reasonable basis and guideline for writing programs clearly.

6.1.2 Information Hiding

A software solution to a problem always contains data structures that contain information about the problem being worked on. That is when software is developed to solve a problem, the software uses some data structures to capture the information in the problem domain, and the application code performs operations on this information. The principle of information hiding states that the information captured in the data structures should be hidden from the rest of the application, and only the access functions on the data structures that represent the operations performed on the information should be visible to the rest of the application. In other words, the information is hidden behind a layer of modules that perform the desired operations, and the rest of the application only uses these defined operations and does not have direct access to the data.

Information hiding can reduce the coupling between modules and make the system more maintainable. Information hiding is also an effective tool for managing the complexity of developing software—by using information hiding, we have separated the concern of managing the data from the concern of using the data to produce some desired results. Information hiding is facilitated through the use of abstract data types, which many languages support.

6.1.3 Pre and Post Conditions

While testing is a commonly used approach to gain confidence in the programs written, it is known that testing can only show the presence of defects but not their absence. It indirectly builds confidence that the program is correct. To show that a program is correct, we need to show that when the program executes, its behavior is what is expected. To specify the behavior, we need to specify the conditions the program's output should satisfy. As a program will usually not operate on an arbitrary set of input data and may produce valid results only for some range of inputs, we generally need to also state the input conditions in which the program is to be invoked and for which the program is expected to produce valid results. The assertion about a program's expected final state is called the program's *post-condition*, and the assertion about the input condition is called the *pre-condition* of the program. Often, in program verification, determining the pre-condition for which the post-condition will be satisfied is the goal of proof.

Using Hoare's notation [2], the basic assertion or specification of a program segment (i.e. a sequence of statements) S is of the form

$$P\{S\}Q.$$

The interpretation of this is that if assertion P is true before executing S, then assertion Q should be true after executing S if the execution of S terminates. Assertion P is the pre-condition of the program, and Q is the desired post-condition. These assertions are about the values taken by the variables in the program before and after its execution. The assertions generally do not specify a particular value for the variables but specify the general properties of the values and the relationships among them.

Pre and post conditions provide formal specifications for a program. These specifications unambiguously and mathematically state what the program should do. Such precise specifications can help a programmer develop correct programs. They are also necessary for formal verification of programs.

To specify larger code programs, we would like to make assertions about the program from assertions about its components or statements. If the program is a sequence of statements, then determining the semantics of a composite program becomes more accessible. If a statement S is composed of two statements, S1 and S2, which are to be executed in sequence, then from the semantics of S1 and S2, we can easily determine the semantics of S (this rule is called the *rule of composition* in [2]) as follows:

$$\frac{P1\{S1\}Q1,\ Q1 \Rightarrow P2,\ P2\{S2\}R2}{P1\{S1;\ S2\}R2}$$

The explanation of this notation is that if what is stated in the numerator can be shown, the denominator can be inferred. Using this rule, if we have shown P1{S1}Q1 and P2{S2}R2, then to determine the semantics of S1;S2, all we need to show is that $Q1 \Rightarrow P2$, and then we can claim that if before execution the pre-condition P1 holds, then after execution of S1;S2 the post-condition Q2 will hold. In other words, to show P{S1;S2}R, once we have determined the behaviors of S1 and S2, we need one additional step. This allows building proofs or arguments about correctness of more extensive programs from proofs of its elements. Note that the rule only handles a strict sequence of statements. So, if we want to apply this, we need to construct the program as a sequence of statements. That is a key motivation for wanting to linearize the control flow in a program by structured programming.

A final note about the structured constructs. Any piece of code with a single-entry and exit cannot be considered a structured construct. A structured construct must also have a clear semantics which can be specified through pre and post conditions. If semantics of each statement in a structured program can be specified, then they can be composed as discussed above to determine the semantics of the program. It is then possible to formally show if a program correctly implements its specifications. Formal verification of programs is an advanced topics which we will not discuss further in the book.

6.1.4 Coding Standards and Guidelines

Recall that (industry-strength) software lasts a long time and that it is developed in teams. This means that others use code written by one developer, and for this they also need to understand it often. In addition, with time, as team members change, the code written by one person will need to be modified by some other developer without the original developer being there to guide the person. Even if the programmer who wrote the code is still with the team, he/she may come back to modify a piece of code written months (or years) earlier after having worked on many other programs and may not remember the context the person had in mind at the time of writing the programs when he/she comes back to the code. This means that code has to be written such that it is easy to understand by itself, i.e, by reading a code file, a developer can understand the program with relative ease.

To help make programs easier for by a reader to understand, programming standards and guidelines have been developed which define some "rules" for programming in some language. Having these common rules followed by all programmers can make it easier for a programmer to understand code written by another programmer. In other words, guidelines and standards encourage programmers to follow some good practices uniformly, which makes the transfer of knowledge from one programmer to another through the program easier. Here, we will discuss some conventions generally specified in coding standards for most languages.

For programming principles, we will consider some suggestions from the "zen of python". For programming standards, we will take some concepts from PEP 8 [https://pep8.org/] and from Google's Python coding guidelines. Though these are stated for Python, the concepts underlying the conventions and practices discussed here will apply to other languages as well (and hence we avoid discussing very language specific guidelines.)

Naming Conventions

During coding a lot of things have to be named—variables, functions, classes, packages, modules, etc. Naming conventions guide how these should be named, so just by the name a reader can figure out what a named object in the program represents—this makes it easier to read code. For Python, some of the conventions suggested in PEP 8 are: for variables, use lower case single letter, word, or words joined by underscore (e.g. x, i, name_last, rollno); for functions, method, and modules names use lowercase word or words joined by underscore (e.g. square_root, myfunction, my_function, palindrome); for class start the name with a capital letter (e.g. Tree); for constants use uppercase word or words (e.g. PI, GRAVITY); for packages use word or words without using underscore.

These guidelines tell the structure of the names of different objects. Names still have to be chosen by the programmer. Often, in a hurry or to save typing effort, programmers choose abbreviations or single-letter names to represent something (e.g. it may be tempting to use n for name, l for the last-name and f for the first-name). When doing this, names effectively become code for some objects (e.g. l represents the last name). Such naming requires the programmer to remember what these codes mean, which becomes hard as time elapses and programs get longer. Moreover, for a new reader, it is extremely challenging to figure out what this coding is. Hence, it is essential that all variable names should be expressive—a name should describe whatever object it is referring to so a reader can quickly figure out what it represents.

Code Layout

A large program is inevitably a collection of modules like functions, classes, methods, etc., with detailed programming statements inside these modules. To understand such programs, one needs to understand at a high level what these modules are doing and how they are connected. So, while reading the code of a module, it is essential to easily separate the code for this module from others such that the reader can focus on the code of that module without undue distraction from other module's code. This is facilitated by the proper use of blank lines. The PEP standard suggests that two blank lines separate each top-level module in a program, and that modules within a module (e.g., methods inside a class) should be surrounded by one blank line.

In a module, statements may have statements nested within them. Nesting is essential for computing. When constructs are nested, what a variable means depends on the nesting, and it is important to see the nesting, and what all statements are at the same level. Therefore proper indentation is used in programs such that nesting becomes evident from the indentation itself. Python, unlike many other languages, uses indentation to specify nesting, so some amount of indentation has to be used in Python programming. However, standards often specify the nature and amount of indentation (e.g. PEP suggests using 4 spaces for indentation). Note that following indentation guidelines is even more important in languages that use explicit means to specify nesting (e.g. use of { } to define a new scope within a scope).

There are also guidelines for using whitespaces in expressions and statements. For example, PEP suggests that a single whitespace should surround most operators. When multiple operators exist in an expression, it suggests that white spaces surround operations with the lowest priority (e.g. x**y+z should be written as x**y + z). Maximum line length (for code or comments) is also advised (e.g. PEP suggest 79 characters)—this helps all the code and comments to fit easily on a screen, which facilitates reading and comprehension.

All these guidelines help make the program easier to read visually, which helps in comprehending the program.

Documentation and Comments

While a piece of code has unique semantics as it is written in a formally defined programming language, extracting the semantics from the code can be hard. Hence, programming standards always recommend using comments to document the code to facilitate its understanding by a reader and make the code stand-alone for understanding. PEP suggests that there should be comments for blocks of code, which should be indented at the same level as the code. Such comments should explain what the code does and its purpose and not try to explain the code logic—by doing so; the reader understands the overall purpose (or desired semantics) from the comment and the logic of how it is being achieved from the code itself. So, the comment supplements the actual code for understanding.

Inline comments are typically added at the end of a statement, in the same line as the code statement text, to generally explain why the statement is needed. PEP suggests that such comments should be used sparingly, and when used they, should be separated from the statement by two spaces.

Docstrings are special types of comments that are often used by compilers/processors to produce documentation for the code. Docstrings are meant to provide documentation for a module unit (function, class, method) and in Python, they are strings enclosed in triple double/single quotes (and so can spread over multiple lines) starting from the first line after the unit, and ending with the closing triple-quote in the last line.

Some Programming Practices

Let us now discuss some programming practices which facilitate program reading and comprehension. We will discuss only a few general guidelines which are applicable to most programming languages, though many of them are from "zen of Python".

- **Explicit is better than implicit.** It is better to be clear and direct rather than relying on hidden or implicit behaviour. First, global variables should be used only if necessary. Global variables create implicit connections between modules (if a function is using a global variable, its value can be changed by other functions without this function knowing—making it much harder to understand the behaviour of the module). It also implies that all communication between modules is made explicit through expressive parameters. So, for example, instead of bundling the parameters in a list or dictionary and then passing them, it is better to pass them directly so the reader can clearly see the parameters. Another implication is that functions should return values to the caller explicitly rather than trying to return the values through the data structures passed as parameters, i.e. through side effects. For example, in Python, if a list is passed as a parameter by a function, the called function can modify the value of the original list (as the list is passed by reference). Instead the called function should return the values it wants and let the caller make the changes in its state. Similarly, in classes, it is better to explicitly pass parameters to methods rather than passing them implicitly through class attributes. There are other examples where the programmer can use some implicit behaviour to get the computation done—all such uses should be avoided, and direct methods should be used to achieve the goal.
- **Simple is better than complex.** Keeping code simple and straightforward whenever possible is preferred over introducing unnecessary complexity. Simple code is easier to understand, debug, and extend. This implies that programmers should try to keep the logic simple, even at the cost of a small amount of efficiency loss. For example, nested control structures (which will require excessive indentation) can be avoided, and the code can be kept flat, even though it may involve some

extra condition checking. Another example is to have a function that does different computations for different cases or perform multiple transformations, which makes the function more complex (and reduces its cohesion)—instead it is better to have separate functions for each of the cases or transformation.

- **Errors should not pass silently.** When errors occur, they should be handled explicitly in the program rather than ignored. Silent failures can lead to unexpected behaviour and make it difficult to diagnose and fix issues. This implies that exception handling should be used to do something programmatically if the exception does occur. It is better to handle different exceptions explicitly rather than have a catch-all exception handler.
- **Don't repeat yourself (DRY).** This is a programming principle based on the observation that many times, there are similar codes in multiple places in the application doing a similar computation. The principle suggests that such duplication should be avoided and that suitable abstraction (e.g. a function) should be created and used in all the places. The principle suggests that "every piece of knowledge must have a single, unambiguous, authoritative representation within a system" [3]. Using this principle, code for performing some functionality or clearly defined logic should be coded only once in the application and reused elsewhere. This approach improves the modularity and cohesion in the design.
- **Prefer built-in functions and libraries.** Most languages provide many built-in functions and many built-in libraries. These functions / libraries have been carefully constructed and are generally extremely reliable and bug-free as they have been used for a long time. Using the built-in function rather than writing your own is clearly better. If the function needed in an application is similar but not the same, in the function for the application, the built-in function should be leveraged to develop the new function.
- **Use standard programming conventions for Boolean conditions.** In conditional statements that evaluate to True or False, when using Boolean variables, it is better not to compare a Boolean object to True or False—instead, the value of the Boolean variable should be used (i.e., instead of using "if flag==True:", just use "if flag:"). Those values should be used for other data structures that may have Boolean values defined by the language. For example, for lists in Python, use the fact that an empty list is False when checking if a list is empty.

6.1.5 Using Open Source Libraries

As discussed in Chap. 1, the productivity of software development can be improved if we can effectively use open source software in the application being built. This directly improves productivity as much code that would need to be written,otherwise, it does not need to be written and can be used, even though integrating such code may require some effort.

A library provides a range of related functions, often in some domain. A programmer can include a library in the code being written, which makes all the library functions available. A huge number of libraries have been developed and made available as open source, particularly for languages like Python which have an ecosystem for easily publishing new libraries which the user can install. Python language provides over 200 libraries, and are over 100,000 libraries are in use. The Python Packaging Index (PyPI) repository, from where the libraries are often installed for Python, has over 500,000 projects. Today, it is fair to assume that for any application development, much code that is needed exists in various libraries. It is important to search and explore what libraries may help the programmer, before developing the code. Sometimes the libraries to be used may be a decision of the project team.

When a program imports a library, the library becomes an integral part of the source code and, hence, is part of the delivered code. So, even if some functions of the library are used, the entire library becomes a part of the application code. In this sense, there is some "unused" code in the application, and the applications size is larger than needed. However, as code itself often consumes minimal memory space, and most platforms like laptops, servers, etc., have sufficient memory, for most applications this is not of concern.

When coding, a programmer should explore the possibility of using available public libraries—this should be viewed as a good programming practice as it helps improve productivity and quality. (Note that the use of frameworks for developing an application is a decision that is usually taken earlier and a programmer usually does not have to make this choice at coding time.)

6.2 Managing Code

During the coding process, the code written by programmers in the team evolves—the overall code for the application evolves as new code files get added or deleted, and the individual code files evolve as they are changed (e.g. to add some feature or fix some defect). In such a dynamic scenario, managing evolving code is a challenge. Also, as industry-strength software lasts a long time (as stated in Chap. 1—this is a distinguishing characteristic of such software), teams working on it change over time. Hence, it is important that the code is organized to make it easier for new team members to understand how the application code is structured. Here, we discuss two aspects of code management—how to organize the code files for an application and how to control the changes to the code.

6.2.1 Code Organization

The code for any application is inevitably distributed among many files. How these code files should be organized is an important issue as later finding the desired code portion or understanding the application is facilitated by well organized code. Code organization is also necessary when multiple programmers are working on the application, as is the norm, so programmers can save the code they write in different source files which must be organized properly for the application. A well organized code is absolutely essential for maintaining and enhancing the application, which, as we have seen, is the most protracted and expensive activity in application software.

One common way to organize the code for an application is to group code files according to the architecture of the application. In this organization, for each component, there can be a folder in which code files for implementing the component reside. This organization facilitates the understanding of the architecture by a new engineer in the team (the names and organization of the folders containing code files indicate the architecture). Suitable naming of the code files can also facilitate the reader in finding specific aspects of the application (e.g. if code relating to the operation for updating the information about a user is placed in the folder for back-end code, and the code file is named update_user_information.py).

Besides the source code for the application, other files are often needed to deploy applications to end users. Good file organization can make it easier for all who work on the project and can ease deployment. Let us first consider a deployable application, i.e. which must be packaged and sent to users for deployment on their local machines. Suppose the application is called myapp. The standard practice is to have all files related to myapp in a directory, which is given the name of the application (i.e. myapp in this case). A typical structure of this directory will be:

```
myapp/
|-- myapp/
    |-- _ _init_ _.py
    |-- appfns.py
    |-- helperfns.py
|-- tests/
|-- docs/
|-- data/
|-- bin/
|-- .gitignore
|-- LICENSE
|-- README.md
|-- requirements.txt
|-- setup.py
```

Let us discuss different elements of the organization of this folder myapp (the name of the application).

- The source code may be kept in the folder myapp (sometimes may be kept in "src" folder). This directory contains the source code for the package to be installed for users. This directory can be quite deep and large, depending on the application
- The folder tests/contains all the test scripts. It is best to separate these scripts, as they need not be included in the packaging for distribution.
- Any data that the application needs, for testing or for its operation, can be kept in the folder data/.

- The folder /bin contains the binaries produced in the project. (In pure python projects, there may not be any need for a /bin folder.)
- A commonly used file is .gitignore, which specifies which files the version control system should ignore (for example, the test files may not be under version control).
- LICENCE is a file which describes the license type of this project. It is usually a text file. One can keep it simple by just stating the license type followed (e.g. MIT license.)
- README.md file is a markup file explaining this project. It is a text file explaining the application, its architecture and main modules, and how to install it. For complex applications, there can be a documentation file for each module. While README provides a summary of the project, other documents about the project that may need to be kept, e.g, requirements documents, design documents, etc., can go in the doc/folder.
- The file requirements.txt is for specifying external dependencies. For distributing an application, all external dependencies need to be clearly and separately specified, so during installation, it can be ensured that the resources that the application depends on are there in the host environment. Generally, dependencies are specified in a separate file, such as requirements.txt (or package.json). Dependency management tools use the information in these files to make the environment ready for executing the application, and installing updates.
- Setup for distribution is specified in setup.py file for Python applications. This specification is essential for the packaging and distribution of applications.

For a large application, there can be sub directories for each major module specified in the design document, and then the above organization can be followed for each module. In such a case, the standard utility functions that are used across modules can be in the parent folder of modules.

Standard top-level folder organisation is often suggested or prescribed for applications developed using frameworks like Django or NodeJS. It is best to follow those as they facilitate communicating the structure to newer members of the team who may join the project later.

The folder containing the source code is often the largest and the deepest. How these code files should be organized is an important issue as later finding the desired code portion or understanding the application will make well organized code very useful. Source code organization is also necessary when multiple programmers are working on the application, as programmers will save the code they write in different source files—well organized source files help in building and maintaining the application.

6.2.2 Source Code Control

When a software application is built, the software is inevitably contained in many files—having code in multiple files, each file containing code for some module(s), is a natural way to organize code. Typically, all files associated with this application development project will be kept in a repository for this project, files being organized logically in folders. From these multiple files, the application is built using some tool. When multiple programmers are working concurrently on the project and so may be working on the same files, there are some desirable features they need to manage these code files and changes being made to them by multiple programmers.

There are many code management tools (CMT), which are also sometimes called version control systems, which provide features for managing the code. We will explain some of the important needs of the programmers and how a CMT can provide them. For this discussion, we will assume that the code repository is maintained in GitHub, which provides free hosting for many projects, and for code version management, we will assume the tool git, which is a popular open-source and freely available code management tool and is commonly used along with GitHub.

Let us first take the simple case of one programmer working on an application. The programmer intends to keep evolving the application, release different versions, and fix bugs and other issues over many years. In this case, as there is only one programmer, the programmer does not necessarily need to put the code in a repository. Let us assume that the programmer has all files for the project in a folder and the programmer puts the folder for this application development project under the CMT (using *git init*, which puts the directory under git control and creates a hidden directory .git to provide the services). We will consider this directory as the "main" directory, i.e. containing the master version of each file. It is called the main branch. To keep evolving the application and adding new capabilities, some of the features this programmer will need from the CMT are:

- **Restore old state of files.** The source code files will evolve as the programmer will add new code or will modify existing code—to add new capabilities or to fix some errors. Often, changes made to code later turn out to be undesirable—for example, some changes made to improving the code performance may not improve the performance. In such situations, the programmer may want to undo the changes made and revert back to some older version of the code files. In other words, while code files are evolving, it is desirable to have checkpoints of the files to which the programmer can revert if needed. This is provided by the *git commit* command, in which the files added to the staging area by the *git add* command are checkpointed (committed). The *git checkout* command allows the restoration of files to a state that was committed. (Actually, checkout creates a new branch with file states of the commit being checked out. There is another command, *git revert* which reverts the files back to the old state and creates a new commit.) As multiple commits may occur over the years, it is desirable to add an expressive message when making a commit. git also provides commands to view the history of commits, as well as commands to show the difference between the current state of the file and the previous commit.
- **Parallel development.** Suppose the programmer wants to simultaneously work on a few different features and wants to keep the code for them separate. She may want to release one feature (along with earlier ones) while developing code for others. In such a scenario, there is a need to allow different branches of the

code files—one feature is developed in one branch by making suitable changes to the code files, and the other feature is developed in the other branch by making necessary changes. When the programmer is ready with code for one feature, she can merge that branch with the main branch, from where the application can be released.
For creating a new branch, the command is *git checkout* (it also allows shifting to some specific branch to make changes there). A branch *git merge* is used to merge. (Note that the checkout gives an impression that there is a copy of the files from the main branch that has been created—in reality, it works with deltas, that is, the changes.)
With parallel development, changes for the second feature may conflict with changes done for the first feature (conflict means that the same lines of code in some files may have been changed in the two branches). If this is the case, then when merging files for the second feature, *merge conflicts* will have to be identified and pointed out (git does this). The programmer has to resolve this then. *Git merge* allows the merging of branches without any merge conflicts.

As we can see, source code control is needed even if a single programmer is working on developing software. Now, let us consider the case where a team of programmers is working on developing code for an application, and each programmer will be developing and testing the code on his/her machine. This is the common scenario as all programmers have powerful laptops on which they work. Such a scenario requires a *repository* where code files for the application will be kept. Programmers can make copies in their local machines, make changes locally, and merge the changes back to the main repository when satisfied. GitHub provides such a repository in which the programmers with access to make changes can also be specified.

In this model, where there is a centralized code repository but with programmers working on their local machines (also called distributed version control systems), there is a need for a version control tool to provide some more commands (besides the ones discussed above) and for the repository to provide some features. When a project is created, a *main* branch is created in GitHub. The following features are needed from the CMT:

- **Clone the repository.** For a programmer to run the application, the entire code for the application needs to be on her machine. *git clone* provides this facility—the entire main branch from GitHub is cloned on the local machine, providing a local *master* branch. For working with the repository (including cloning), the git tool, which runs on the local machine, has to provide methods for connecting and reading/updating the files in the GitHub repository—the command *git remote* provides this connection.
- **Update the local master with the latest files from the main in the repository.** As multiple programmers are working and will be updating the main branch in the repository, when a programmer makes some changes and adds new code, the main (and so the local master) may have changed. Moreover, even if the changed code works with the old master/main code, it may not work with the new one. Hence,

before trying to update the repository with code the programmer has written, she must update her cloned repository and ensure that her code works with the updated main in the repository. This update of the local repository is done using *git pull* command. This syncs up the local master with the main branch on GitHub.

- **Update the main repository.** After the programmer is ready with the code, she would want to update the main code base in the repository, which will also make the changes available to other programmers. This can be done by *git push* command. As many programmers work in parallel, rather than pushing changes directly to the master branch (which has many potential ramifications and is to be avoided), a common approach of using git and GitHub is that a programmer creates a new branch and works on it. When ready, the programmer can use *git push* to push this branch to GitHub—which creates (or updates) this branch on GitHub. However, as changes to the main repository may have been made to ensure that programmer's code works with the current code files in the main, generally, first the local master is updated using the *git pull* command, and if the code still works fine, the branch can be pushed to GitHub. Once this is done, the changes have been moved to the repository, and the repository needs to provide a mechanism to merge the changes to the main branch. Merging a branch with main can be done in GitHub by making a *pull request* (PR) (this pull request is different from the *git pull* command). For the pull request, before making the changes to the main, a review step is added—only after the reviewer approves can the changes be merged into the main. (Of course, if there are potential merge conflicts in this PR, the merging is rejected.)

The above explanation keeps a simple project in mind with one main/master and multiple branches to motivate why the different commands are in a CMT. In addition to these, there are some commands like *git status*, *git branch*, *git log*, etc, which programmers use to get information or navigate the repository. Other advanced commands also to allow for more complex projects (e.g, with multiple parallel releases while keeping old releases available for backward compatibility.) GitHub also has many capabilities for different aspects of code and project management.

Note that once the changes are committed, they become available to all members of the team, who are supposed to use the source files from the repository. Hence, it is essential that a programmer must commit a source file only when it is in a state that others can use. The normal behavior of a project member will be as follows: check out the latest version of the files to be changed; make the planned changes to them; validate that the changes have the desired effect (for which all the files may be copied and the system tried out locally); commit the changes back to the repository.

It should be clear that if two people check out some files and then make changes, there is a possibility of a conflict—different changes are made to the same parts of the file. A CMT toolwill detect the conflict when the second person tries to commit the changes and will inform the user. The user has to manually resolve the conflict, i.e., make the file such that the changes do not conflict with existing changes, and then commit the file. Conflicts are usually rare as they occur only if different changes are made to the same lines in a file.

With a source code control system, a programmer does not need to maintain all the versions—at any time, if some changes need to be undone, older versions can be quickly recovered. The repositories are always backed up, so they also protect against accidental loss. Furthermore, a record of changes is maintained—who made the change, when, why the change was made, what the actual changes were, etc. Most importantly, the repository provides a central place for the latest and most authoritative project files. This is invaluable for products that have a long lives and evolve over many years.

Besides using the repository to maintain the different versions, it is also used to construct the software system from the sources—an activity often called *build*. The build gets the latest version (or the desired version number) of the sources from the repository and creates the executables from the sources.

Building the final executables from the source files is often done through tools like Makefile [4], which specify the dependence between files and how the final executables are constructed from the source files. These tools can recognise that files have changed and will recompile whenever files are changed to create executables. With source code control, these tools will generally get the latest copy from the repository and then use it to create executables.

What we discussed here is one of the most straightforward approaches to source code control and build. Often, when large systems are being built, more elaborate methods for source code control may be needed.

6.3 Developing Code

For a new application development project, the coding activity starts when design has been done and the specifications of the modules to be developed are available. With the design, modules are assigned to developers for coding. When enhancing an application, developers may be assigned a new feature to be added or a bug to be fixed, for which they also have to do the design. When modules or features are assigned to developers, they use some process for developing the code. Here, we discuss some effective processes that developers may use.

6.3.1 An Incremental Coding Process

The process followed by many developers is to write the code for a module and, when done, perform unit testing on it and fix the bugs found.

A better process for coding that experienced developers often follow is to develop the code incrementally. That is, write code to implement only part of the module functionality. This code is compiled and tested with some tests to check the code that has been written so far. When the code passes these tests, the developer adds

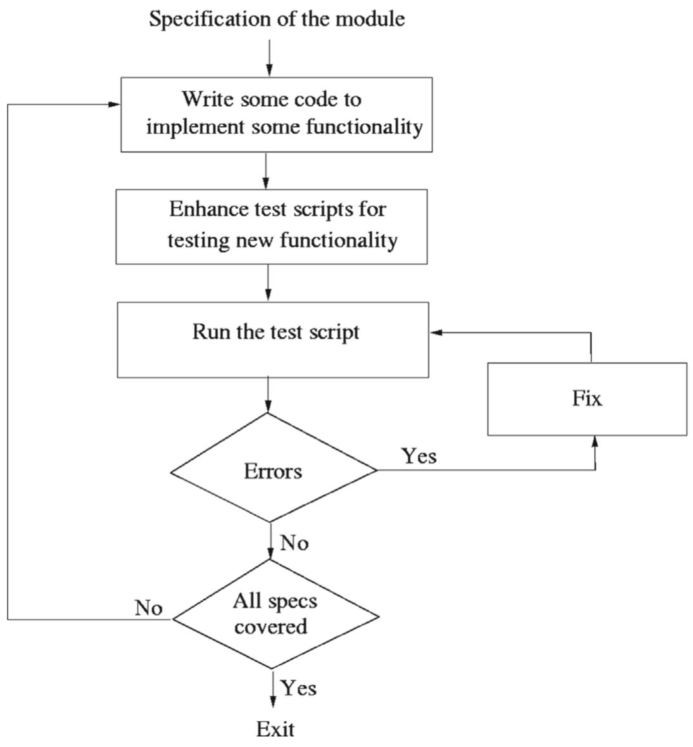

Specification of the module
Write some code to implement some functionality
Enhance test scripts for testing new functionality
Run the test script
Fix
Errors
Yes
No
All specs covered
No
Yes
Exit

Fig. 6.1 An incremental coding process

further functionality to the code, which is then tested again. In other words, the code is developed incrementally, testing it as it is built. This coding process is shown in Fig. 6.1.

The basic advantage of incrementally developing code with testing done after every round of coding is to facilitate debugging—an error found in some testing can be safely attributed to code that was added since the last successful testing. However, for following this process, it is essential that testing be done through test scripts that can be run easily. With these test scripts, testing can be done as frequently as desired, and new test cases can be added easily. These test scripts are also a tremendous aid when code is enhanced in the future—through the test scripts, it can be quickly checked that the earlier functionality is still working. These test scripts can also be used with some enhancements for the final unit testing that is often done before checking in the module.

6.3.2 Test-Driven Development

Test-Driven Development (TDD) [5] is a coding process that turns around the standard approach. Instead of writing code and then developing test cases to check the code, in TDD, it is the other way around—a programmer first writes the test scripts and then writes the code to pass the tests. The whole process is done incrementally, with tests written based on the specifications and code written to pass the tests. The TDD process is shown in Fig. 6.2.

This relatively new approach, has been adopted in the extreme programming (XP) methodology [6]. However concept of TDD is general and not tied to any particular methodology.

A few points are worth noting about TDD. First, the approach says that you write just enough code to pass the tests. By following this, the code is always in sync with the tests. This is not always the case with the code-first approach, in which it is all too common to write a long piece of code but only write a few tests covering only some parts of the code. By encouraging that code is written only to pass the tests, the responsibility of ensuring that required functionality is built is shifted to the activity of designing the test cases. That is, it is the task of test cases to check that the code that will be developed has all the functionality needed.

In TDD, some prioritization for code development naturally happens. It is most likely that the first few tests will focus on using the main functionality. Generally, the test cases for lower-priority features or functionality will be developed later. Consequently, code for high-priority features will be developed first, and lower-priority items will be developed later. This has the benefit that higher-priority items get done first, but it has the drawback that some of the lower-priority features or some exceptional cases for which test cases are not written may not get handled in the code.

As the code is written to satisfy the test cases, the completeness of the code depends on the thoroughness of the test cases. Often, it is hard and tedious to write test cases for all the scenarios or special conditions, and it is doubtful that a developer will write them for all the special cases. In TDD, as the goal is to write enough code to pass the test cases, such special cases may not be handled. Also, as each step code is written primarily to pass the tests, it may later be found that earlier algorithms were not well suited. In that case, the code may need to be improved before new functionality is added, as shown in Fig. 6.2.

6.3.3 Pair Programming

Pair programming is also a coding process proposed as a critical technique in extreme programming (XP) methodology [6]. In pair programming, code is not written by individual programmers but by a pair of programmers. The coding work is assigned not to an individual but to a pair of individuals. This pair together writes the code.

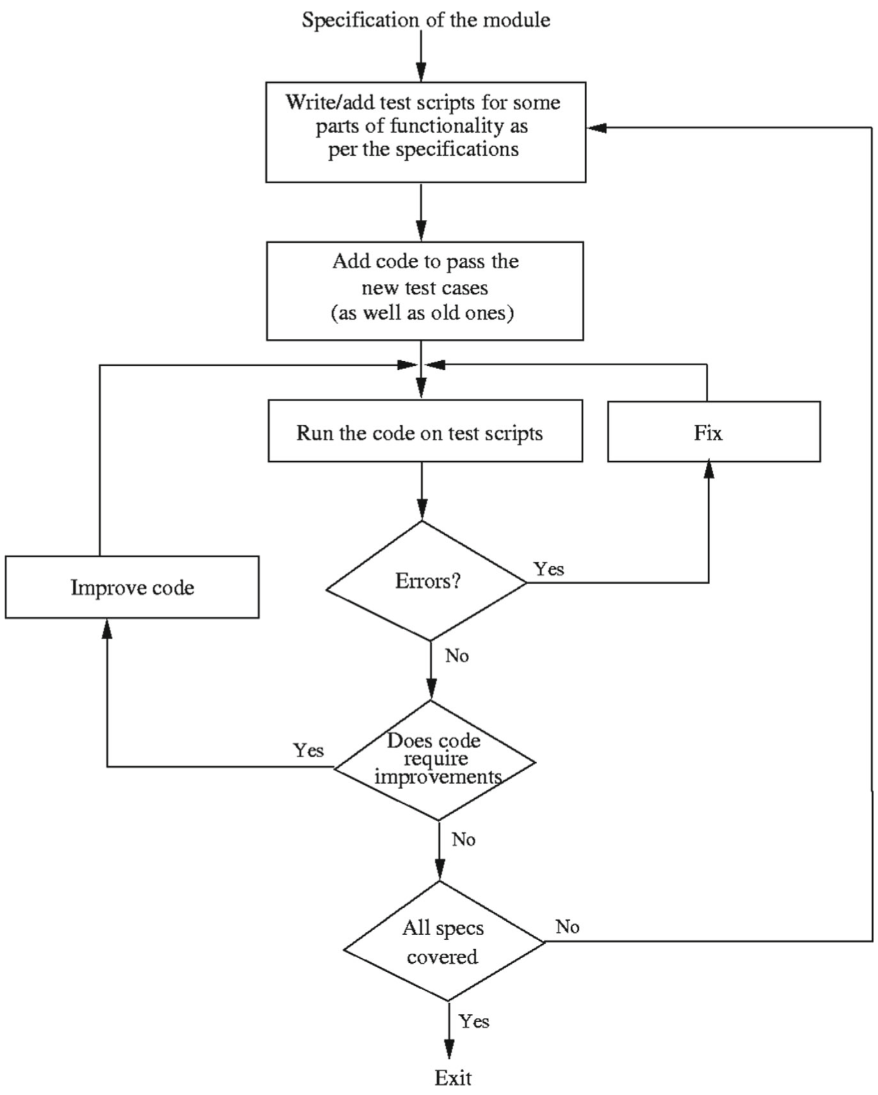

Specification of the module
Write/add test scripts for some parts of functionality as per the specifications
Add code to pass the new test cases (as well as old ones)
Run the code on test scripts
Fix
Improve code
Errors?
Yes
No
Does code require improvements
Yes
No
All specs covered
No
Yes
Exit

Fig. 6.2 Test-driven development process

The process envisaged is that one person will type the program while the other will actively participate and constantly review what is being typed. When errors are noticed, they are pointed out and corrected. When needed, the pair discusses the algorithms, data structures, or strategies to be used in the code to be written. The roles are rotated frequently, making both equal partners and having similar roles.

The primary motivation for pair programming is that as code reviews have been found to be very effective in detecting defects, by having a pair do the programming, the code gets reviewed as it is being typed. That is, instead of writing code and then getting it reviewed by another programmer, we have a programmer who constantly reviews the code being written.

Besides ongoing code review, having two programmers apply themselves to the programming task is likely to result in better decisions about the data structures, algorithms, interfaces, logic, etc. Special conditions, which frequently result in errors, are also more likely to be dealt with better.

The potential drawback of pair programming is that it may result in a loss of productivity by assigning two people for a programming task. A pair will produce better code than code developed by a single programmer. The open question is whether the improved code quality offsets the loss incurred by putting two people on a task. There are also accountability and code ownership issues, particularly when the pairs are not fixed and rotate (as has been proposed in XP).

6.4 Using LLMs for Coding

LLMs for coding has been a very active area in recent years, and genAI/LLM based tools have become extremely powerful and can generate good-quality code from specifications for a wide range of problems. LLMs are already being used widely by programmers to generate code, test cases, documentation, etc, and seek help for debugging if the program execution under testing shows errors and for reviewing the code.

Here we will discuss how a widely used genAI tool, Copilot, can be used effectively during coding. We will also discuss how an interactive tool like ChatGPT can be used. As with other tasks, we assume that the software developer remains responsible for developing and testing the code and these are tools that the developer can use. The discussion is brief as this is an extremely dynamic area, evolving rapidly. Many resources are available online, and new ones become available daily.

Using Copilot

A programmer, when developing a feature assigned to him/her, will often have to develop many modules and modify some existing modules. The developed code has to be tested properly and then integrated. If errors are found during testing, the bug has to be identified, and the code has to be fixed. Here, we briefly describe how Copilot can be used for many tasks. Some of the tasks where Copilot can help are: (1) code completion—write code from code description, (2) develop test cases for a module, (3) develop documentation/explanation for a module code, (4) debugging if errors are found during execution. We will briefly describe how these can be done and what type of prompting is being used.

When working with an IDE like VS code, the programmer is in programming mode and writing programs, and IDEs typically provide various types of support (e.g. indenting, filling the variable name, providing options for methods on an object, etc.). Using this approach, Copilot suggests code to the programmer while the programmer is working, and it is up to the programmer to accept the suggestions. It should be noted that Copilot uses Codex, an LLM trained on large amounts of source code and not all text, as LLMs like ChatGPT are [7]. This helps Copilot give more appropriate programming suggestions.

There are many resources and training programs online on how to use Copilot effectively. We will only briefly mention how it can be used here with VS code, an IDE supporting many languages. The examples here are for Python.

Code generation or completion. Programmers often write a comment for a function (or class) that they define, then define the header, and then write the code. If this sequence is followed, Copilot will suggest code at any stage—after the comment describing the function, or after the function header is defined, while the programmer enters code. It often suggests complete code for the function, particularly if the function is quite standard, and the programmer can accept the entire suggestion or accept it word-by-word. Otherwise, while entering code, Copilot uses the entire context (comments, function definition, code so far) and suggests either completing the current statement or suggesting multiple lines of code. The programmer can accept the entire suggestion, accept the suggestion word by word, or reject it (by continuing to type).

For example, suppose a programmer has to write a function to process a text file containing numbers and text and has to return the mean of the numbers in it and the total number of numbers. For this problem, a programmer may first write a function to test whether a string is a number, then use it to write a function to read the file and do the desired computation. The programmer interaction may be something like:

```
# check if a string is a number - integer or float
def is_number(s):
```

Just after the comment, or after the keyword `def`, Copilot is likely to suggest a reasonable code for this function, which the programmer can accept fully. Next the programmer may type:

```
# Compute the mean of the numbers in a file
def mean_of_numbers_in_file(f):
```

Again, Copilot is likely to suggest the code for this, which the programmer can accept. Using Copilot in this style will be enhanced by good comment on the task being done, and the programmer should consciously write the prompt in a manner that will help Copilot generate correct code, which may be different from the commenting style the programmer usually followed. This form of usage is called acceleration mode in [8]—i.e. the programmer knows how to proceed and uses Copilot as an intelligent assistant to complete the task more quickly. Though Copilot can suggest code snippets to the programmer, as it is the responsibility of the programmer to write code, the programmer must evaluate the suggestion and then accept it. Furthermore, the code snippets suggested by Copilot may sometimes be harder to debug or understand. In other words, the help is not entirely free—a programmer may spend less time writing code but has to spend more time reviewing code [7].

Another form of interaction is possible which is more exploratory where the programmer can ask the Copilot to provide multiple suggestions and then select from them [8]. This type of use requires much more effort from the programmer, who has to examine the different suggestions and then select one.

Unit test case generation. Programmers will generally test the functions they are developing independently, i.e. unit testing them. For this, they may write some unit test cases in the same file where the function is defined, or may define them as separate test files that can be run using some framework. Copilot can provide help in either of these styles. For the first, after defining the function, if the programmer types the comment:

```
# Test cases for the function foo
```

Just with this prompt, test cases will be suggested—generally as assert statements. A programmer can accept these. If some other form is desired (e.g. print a statement after the successful execution of a test case), the programmer can accept the first test case and add the desired extras, and then for other test cases; these extras will also be added. If the test cases are part of the code file and should be executed only if the program is run as a script, the following comment does the trick:

```
# test cases for foo, if the script is run directly
```

For having test cases in a separate test file to be executed by some framework (e.g. pytest or unittest for python), then the file can be created and with a suitable comment at the start, the test case generation starts working. Test cases for a framework like *pytest* can be generated in the same file as the program also.

Develop documentation for a module. Copilot has a chat provision that can help generate documentation for the code or help understand the code. For example, one can have some code and then, in the chat, ask "Explain this code" and an explanation of what the code is will generate—this can be useful for understanding some pre-existing code that a programmer has to modify. If a programmer has coded some functions, the prompt "generate docstrings for these functions" generates the docstring for the functions—which can help the programmer cross-check the code. (Docstrings can also be generated while coding by suitably leading it by inserting """ after the def and waiting for it to respond.)

Debugging code. Copilot can also help in debugging. Most IDEs can well address syntax errors. If the program has some runtime error, then the program will either give some incorrect output or will have an exception. If an exception is raised by the runtime system, then the line number where the exception occurred can be highlighted which will bring up options, one of which will be to use copilot, which then gives options to document, explain, fix or test. Choosing a fix will provide suggestions for changing the code, which the programmer can accept or decline.

Using ChatGPT

ChatGPT is a powerful LLM that can generate meaningful text given some context and suitable prompts. As programs are also text, and as a huge amount of source code of different programming languages have been used for training, it is also very effective in generating source code if prompted suitably. It can also help generate test cases for some code, documentation for existing source code (e.g. for understanding some code), analyse the code quality using some measures, etc.

Experience suggests that often, a single prompt approach can work well for a range of coding-related activities. It can be used to generate code, evaluate a given code for a problem statement, provide feedback on how to improve it, debug a given code that gives errors when executing, etc.

For generating code, simple prompts of the type "Please write Python code for the following problem: ... often work well. If it is desired that the code be written in a certain manner (e.g.it should have some functions that are then called by the main program), such instructions can also be added. It can write non-trivial programs also.

For checking the quality of a given code, the prompt can be something like (it is logically one prompt, just divided into two prompts for separating instructions from the task statement):

> Prompt: You are an experienced programmer and will be presented in the next prompt a piece of code written by a student for a simple programming task. Your role is to take the student code as input and provide qualitative feedback on the code's current structure, style, and readability. Suggest a refactored version of the code, or provide detailed advice on what to change to make it more efficient, readable, or well-structured. Your recommendations can include, but are not limited to, better variable names, the use of functions for repetitive tasks, improved control flow, or the adherence to coding standards and best practices.
> Prompt: ... code ...

It is able to suggest improvements in code structure, simplify nesting, improved names, alternative data structures for the problem. It can suggest restructured code.

For debugging a program, the LLM can be given the code and the error it is encounters and then asked what the error is and how to fix it. An example prompt is:

> Prompt: I would like your help in debugging some Python code. The code and the error are given below:
> code: ....
> Error: ...
> Please explain what the error is, and fix the bug in the code.

ChatGPT can also be prompted to generate suitable documentation—one can generate comments for functions or explanations by suitably prompting it.

6.5 Verification

Whenever code is developed, it has to be verified that the written code does what it is supposed to do, i.e. it satisfies the specifications for the code. This activity is called verification—ensuring that the code implements the specifications. In the section above, we have already mentioned that testing is one of the methods for verification that programmers universally use. Code reviews are another method that is commonly used. We will discuss these two approaches more here. Formal verification is another way to verify programs, but we will not discuss those. Here, our focus is on verifying modules or coding units developed by a programmer. Verification of the entire application, which has code developed by all the programmers in the team, is primarily done through testing, and we will discuss that in further detail in the next chapter.

6.5.1 Unit Testing

Once a programmer has written the code for a module, it has to be verified before others use it. Unit testing remains the most common method of this verification. Unit testing is like regular testing, where programs are executed with some test cases, except that the focus is on testing smaller programs or modules, typically assigned to one programmer (or a pair) for coding. In the programming processes we discussed earlier, the testing was essentially unit testing.

A unit may be a function, a small collection of functions, or a class or a small collection of classes. During unit testing, the tester, who is generally the programmer who wrote the code, will execute the unit under test with a variety of test cases and study the actual behavior of the unit being tested for these test cases. Based on the behavior, the tester decides whether the unit is working correctly or not. Executing a test case normally involves the following steps:

1. Set the system state as desired for the test case
2. Execute the unit
3. Compare result with expected results and declare success or failure.

These three steps are sometimes referred to as the Arrange, Act and Assert steps of testing. (Sometimes, they may be followed up by a Cleanup step in which whatever was setup for testing is cleaned up.)

An issue with unit testing is that the unit being tested may not be executable by itself. To execute the unit for testing, drivers may be needed, the purpose of which is to invoke the module being tested with the test data. A unit may also use other modules that have not yet been developed or may use data inputs from the larger system to function. In such situations, to keep the unit tests stand alone, test doubles are often used to provide the unit under with a testing with a simple or dummy for modules or objects to facilitate testing. These test doubles are not the actual modules or values that will be there in the full system but are simpler versions that behave like actual modules/objects. Common test doubles are stubs (which are dummy modules that the unit under test calls), dummy objects (which are objects provided for tests that do not use the value of the object but need the object to execute), mock objects (which are simplified objects, e.g. for a table, which provide values that a test may need), etc.

Note that if the checking of the results shows that it is not as expected for some test case, then the programmers will find the defect in the program (an activity called *debugging*) and fix it. After removing the defect, the programmer will generally execute the test case that caused the unit to fail again to ensure that the fixing made the unit behave correctly.

As programs have to be tested frequently, instead of entering test cases manually and then running the programs, which can be very cumbersome and time consuming, it is better to have test scripts, which are programs that will run the code modules with test cases and check if the outputs are as expected. Developing these testing scripts is an investment in time initially, but in the long run, they pay off as executing the tests repeatedly is just a click of a button, saving considerable time for manually entering test cases.

For executing the test scripts for testing the programs to be tested, testing frameworks are commonly used now, which not only execute the test cases written by the programmer (i.e. provide the driver) but also facilitate the creation of test doubles and help analyze the results. Testing frameworks like JUnit for Java and pytest for Python allow test cases to be written as simple classes or functions that call the functions/methods to be tested with test values and use assertions to check the results. To illustrate the use of such frameworks, we briefly discuss how testing can be done using pytest, the most widely used testing framework for Python.

When using a testing framework like pytest, a test case will often be declared as a function in which this sequence of steps is executed for that test case, including checking the outcome and declaring the result—this being done using the assert statement provided by the language. A test suite is a collection of these functions, and execution of a test suite means each function is executed. The framework calls the tests in the test suite, and the results are monitored by it. To facilitate the discovery of functions implementing test cases, each test function in pytest must have the prefix "test" or "test_". The files containing these functions are also prefixed similarly (if the name starts with "test_" or ends with "_test" then pytest will automatically run these files from the current directory. Otherwise, pytest must be asked to execute the file containing the test cases). Often, the test files are kept in the test folder for the project. After the execution of the test cases, the test framework will summarize the testing results in a suitable report.

For example, let us consider checking if a string or a number is a palindrome. For this, suppose the programmer writes two separate functions (in the file palindrome.py):

```
def str_palin(s):
    code for it

def num_palin(n):
    code for it
```

To test these functions, a file test_palindrome.py is created, in which the test cases are written as functions. For example, it may have these tests:

```
import pytest
from palindrome import str_palin, num_palin

def test_num_palin1 ():
    assert num_palin(101)

def test_num_palin2():
    assert num_palin(100)

def test_str_palin1():
    assert str_palin("")

def test_str_palin2():
    assert str_palin("hello")

def test_str_palin3():
    assert str_palin("Bob")
```

With this file, pytest can now be run: pytest test_palindrome.py (if pytest is run from the same directory as the test file, then the file need not be specified—it will pick all files starting with "test_"). From this file, it will find all test functions (i.e. starting with "test") and execute them and then provide a summary, which specifies how many test functions it found, which ones executed successfully and which ones failed (for each test it prints one character – "." If the assertion holds and "F" if it fails). So, the above provides the summary:

```
Collected 5 items
test_palindrom.py  .F.F.
```

Then it gives details of all the failed test cases, and finally summarizes the failure cases.

Frameworks like pytest are simple and intuitive to use. To further support testing, they provide many features. For example, if some common data is to be used for many test cases, they can be defined separately in a fixture, which test cases can use. LLMs can also be used to facilitate unit testing. We will discuss this in the next chapter.

6.5.2 Code Reviews

While unit testing of code is essential before any new (or modified) code can be integrated into the application, it is sometimes not sufficient. It is common practice to review new code or code changes before they are integrated into the main repository. Here, we discuss two approaches for review—one person review which is now widely used particularly in the open source development process, and code inspections—an older technique that is quite effective and may be used for critical modules.

One Person Reviews

One person reviews are very common for projects using open-source processes. Typically, when a programmer requests to push the code she has developed into the code repository, it is common practice to have the code reviewed and approved by another person before it can be merged. In this form of review, two main actors are involved—an author who wrote the code and the reviewer who reviews the code. The review of a code may involve multiple rounds. A review round starts with the author sending the code, marking the changes made to it to the reviewer, and the reviewer, after reviewing the code, sends the list of issues identified in the code or approves the changes. If the issues identified need further review, the reviewer may not approve the code in that round, and another round has to take place. The review ends with the reviewer approving the changes.

Though the purpose of code review is to verify that the code is ready and to identify any defects it may have, code review is a human process—a human reviews code, and the author of the code being reviewed is a human who is also responsible for accepting and rectifying the issues identified in the code by the reviewer. Hence, human aspects of code reviews are crucial for achieving the goal of high-quality code. Here, we discuss some practices that should be followed by the author and reviewer to ensure effective code review, based on [9,10].

- The author should not use the reviewer to check for issues that tools can find, ensuring that the reviewer's time (a scarce resource) is spent on identifying more substantive issues. For example, the code submitted for review should be consistent with the coding style guidelines, not break the build for the application, and pass the automated tests that may be there for the project. The author of the code should use tools to ensure that such issues are resolved before sending the code for review.
- Author should not send too large a code (changes) to be reviewed, respecting the reviews time and recognizing that the reviewer is also a programmer with other tasks in the project. It is better to split a large code into reasonable sizes and review each portion separately.
- The reviewer of the code should start the review soon and give the responses quickly (generally within a day), respecting the fact that unless the code is approved, the author cannot proceed further. This helps the author and the project as time wastage is minimized.
- Limit the number of issues raised in one round of review. If the code has many issues, then it is better to focus on high-level issues in the initial round(s), and if there are multiple similar issues (e.g. many variable names are not suitable), then instead of listing separate issues for each, it is better to list one general issue (e.g. some variables do not seem to be named appropriately, e.g. x, y, and z)
- Reviewer should be extremely cautious to ensure that it is the code that is being reviewed and not the programmer. The programmer is a peer in the team who needs to be treated with respect and trust. Hence, issues that are recorded should be about the code and never about the programmer Hence, the use of "you" should be completely avoided. So, for example, saying, "you have made this mistake in the loop condition", which can be interpreted as pointing to the failure of the programmer and is declarative, the comment should be about the code and made as a suggestion "the loop condition seems to be incorrect."
- Similarly, the use of "I" should be avoided, and instead of a comment like "I found that this statement can lead to an issue x", the comment should be more general "This statement may lead to an issue x."
- When suggesting some course of action, it should be made as a request rather than a command—in line with the respect principle of teamwork. For example, instead of saying, "move this condition checking from x to location y", the comment can be something like "consider moving the condition checking at x to the location y."
- When making suggestions, instead of giving them as reviewers views or opinions, use some principles or guidelines to justify the suggestion where possible. So, instead of saying, "the function mean_and_median() is doing two functions and should be split into two separate functions", the comment can be "the function mean_and_median() is doing two functions, and to enhance cohesion, it may be split into two separate functions."
- Give sincere praise where it is due. Even though this is not the goal of the review (which is to find issues) it supports the overall goal of high-quality software as the human programmer is more likely to respond properly to other issues that are pointed out.

- Give suggestions for fixing, where possible. Even though code review is about finding issues that the author has to fix, as the author is a peer, when you understand the fix well, you can suggest it as a "bonus" to the author

These are suggestions for the review activity following the HRT (humility, respect, trust) principles of effective teamwork discussed earlier (in Chap. 2). Keeping the HRT principles in mind, other guidelines can be added for the author and the reviewer.

Code Inspections

Code inspection is a review of code by a group of peers following a clearly defined process. The basic goal is to improve code quality by finding defects. Inspections have been found to help improve quality and productivity (see reports in [11,12]). Here we briefly discuss how code inspections may be conducted based on how they were done by a commercial organization [13].

Inspections are performed by a team of reviewers including the author, with one of them being the *moderator*. The different steps in inspections are planning, self-review, group review meeting, and rework and follow-up.

During the planning step, an inspection team of at least three people is formed, including the programmer whose code will be reviewed, and a moderator is appointed. The author of the code to be reviewed ensures that the code is ready for inspection, e.g. the code compiles correctly and the available static analysis tools have been applied to check for defects that such tools can find. The code is distributed to the inspection team, along with the specifications for which the code was developed. A checklist of what defects should be looked for in the code (which may be prepared specifically for the project) can also be shared.

In the self-review step, each reviewer goes through the entire code and notes all the potential defects he or she finds. Ideally, the self-review should be done in one continuous period of a few hours (so the code to be reviewed should be of suitable size). The reviewers also record the time they spent in the self-review.

Once all the self-reviews are done, the group review meeting is held. The primary purpose of this step is to come up with the final defect list based on the initial list of defects identified by the reviewers and the new ones found during the discussion in the meeting.

The entry criterion for this step is that the moderator is satisfied that all the reviewers are ready for the meeting. The main outputs of this phase are the defect log and the defect summary report. The meeting is conducted as follows. A team member (called the *reader*) reviews the code line by line (or any other convenient small unit). At any line, if any reviewer has any issue from before or finds any new issue in the meeting while listening to others, the reviewer raises the issue. There could be a discussion on the issue raised. The author accepts the issue as a defect or clarifies why it is not a defect. After discussion, an agreement is reached, and one member of the review team records the identified defects in the defect log. At the end of the meeting, the defects recorded in the defect log undergo a final review by the team members. Note that defects are only identified during the entire process of review. It is not the purpose of the group to identify solutions, which is done later by the author. The final defect log is the official record of the defects identified in the inspection and may also be used to track the defects to closure.

6.6 Metrics

Various metrics have been proposed to quantify different characteristics of code. Metrics can help ensure better quality code as well as better control. For the code, the most commonly used metrics are size and complexity. Here, we discuss a few measures of size and complexity.

6.6.1 Size Measures

The size of a product is a useful measure, as it is something that people often wants to know, and it is also the major factor that affects the cost and schedule. Size is also useful to estimate the productivity during a project (e.g., KLOC per person-month) or to quantify the final quality of a project (e.g. defects per KLOC). As larger programs are harder to understand, size may also be used to establish guidelines on the size of modules. For these reasons, size is among the most important and frequently used metrics.

The most common measure of size is delivered lines of source code, or the number of lines of code (LOC) or thousands of lines of code (KLOC) finally delivered. There are issues with LOC as a measure of the size of software. For example, the number of lines of code depends heavily on the language used for writing the code, and a program written in one language (e.g. assembly) may be larger than the same program written in another language. Even for the same language, the size can vary considerably depending on how lines are counted. Despite these deficiencies, LOC remains a handy and reasonable size measure that is used extensively. Currently, the most widely used counting method for determining the size is to count non-comment, non-blank lines only.

Halstead [14] has proposed metrics for the length and volume of a program based on the number of operators and operands. In a program, we define the following measurable quantities:

- n_1 is the number of distinct operators
- n_2 is the number of distinct operands
- $f_{1,j}$ is the number of occurrences of the jth most frequent operator
- $f_{2,j}$ is the number of occurrences of the jth most frequent operand.

Then the vocabulary n of a program is defined as

$$n = n_1 + n_2.$$

With the measurable parameters listed earlier, two new parameters are defined:

$$N_1 = \sum f_{1,j},\ N_2 = \sum f_{2,j}.$$

N_1 is the total occurrences of different operators in the program and N_2 is the total occurrences of different operands. The length of the program is defined as

$$N = N_1 + N_2.$$

From the length and the vocabulary, the volume V of the program is defined as

$$V = N \mathrm{l}og_2(n).$$

This definition of the volume of a program represents the minimum number of bits necessary to represent the program. $\mathrm{l}og_2(n)$ is the number of bits needed to represent every element in the program uniquely, and N is the total occurrences of the different elements. Volume is used as a size metric for a program. Experiments have shown that the volume of a program is highly correlated with the size in LOC.

6.6.2 Complexity Metrics

The productivity, if measured only in terms of lines of code per unit time, can vary a lot depending on the complexity of the software to be developed. A team will produce less code for highly complex system programs than a simple application program. Similarly, complexity has great impact on the cost of maintaining a program. Several metrics have been proposed for quantifying the complexity of a program [15], and studies have been done to correlate the complexity with maintenance effort. Here, we discuss a few of the complexity measures that have been proposed.

Cyclomatic Complexity

Based on the capability of the human mind and the experience of people, it is generally recognized that conditions and control statements add complexity to a program. Given two programs of the same size, the program with the larger number of decision statements can be considered to be more complex. The most straightforward measure of complexity, is the number of constructs that represent branches in the program's control flow, like `if then else, while do, repeat until`, and `goto` statements.

A more refined measure is the *cyclomatic complexity measure*, a graph-theoretic–based concept. For a graph G with n nodes, e edges, and p connected components, the cyclomatic number $V(G)$ is defined as

$$V(G) = e - n + p.$$

To use this to define the cyclomatic complexity of a module, the control flow graph G of the module is considered. To construct a control flow graph of a program module, break the module into blocks delimited by statements that affect the control flow, like `if, while, repeat`, and `goto`. These blocks form the nodes of the graph. If the control from a block i can branch to a block j, then draw an arc from node i to node j in the graph. The control flow of a program can be constructed mechanically. For a module, the *cyclomatic complexity* is defined as the cyclomatic number of the control flow graph for the module.

It can also be shown that the cyclomatic complexity of a module is the number of decisions in the module plus one, where a decision is effectively any conditional statement in the module [16].

Halstead also proposed some measures which can be considered complexity metrics. As given earlier, several variables are defined in software science. The ratio $n_1/2$ can be considered the relative difficulty level due to the larger number of operators in the program. The ratio N_2/n_2 represents the average number of times an operand is used. In a program in which variables are changed more frequently, this ratio will be larger. As such programs are more challenging to understand, *ease of reading or writing* is defined as

$$D = \frac{n_1 * N_2}{2 * n_2}.$$

Other measures for complexity have also been proposed and studied. The complexity of programs can be used to compare programs, to study the effects of complexity on various attributes, etc. One use is to have guidelines limiting the complexity of modules in an application to ensure that modules are not too complex, making them hard to understand (e.g. some suggest that the cyclomatic complexity of a module should be less than 10).

6.7 Summary

- As reading programs is a much more common activity than writing programs, the goal of the coding activity is to produce programs that are, besides being free of defects, easy to understand and modify.
- Use of structured programming in which the program is a sequence of suitable single-entry single-exit constructs, makes programs easy to understand and verify. Other practices like information hiding, suitable coding standards, and good programming practices also help improve code readability and quality.

- The code for an application needs to be organized properly and its evolution should be managed. Some common practices are discussed for organising code in different folders and files. Code evolution can be properly managed through source code control tools, which allow easy management of the different versions that get created, easy undoing of changes that need to be rolled back, and many other features.
- For a developer, it is most effective to develop code incrementally. This can be done by writing code in small increments and testing and debugging each increment before writing more code. Alternatively, test-driven development may be followed by writing test cases first and then writing code to pass these test cases. Though coding of a module is generally done by individual programmers, an alternative is pair programming, in which a pair of programmers do coding—both together evolving strategies, data structures, algorithms etc.
- Unit testing is widely used by programmers for verifying the code they have written. In unit testing, the programmer tests his/her code in isolation. For procedural languages, this is often a small set of procedures or functions, and for object-oriented languages, this is generally a class or a small set of classes. Unit testing can be facilitated by the use of frameworks like pytest, JUnit and unittest, which allow automated test script execution and checking for errors.
- Another method for verifying programs is code reviews. Review can be done by one-person, which is now the commonly followed practice. Guidelines for effective review have been discussed. The review can also follow a code inspection process, in which a team inspects the code.
- LLMs can facilitate coding. Tools like Copilot have been integrated with IDEs like Visual Studio Code and provide a coding assistant to programmers. In this mode, the programmer works normally and types the code and/or comments and the Copilot prompts with code for completing the block or module being written—the programmer can accept or reject the suggestion. Interactive LLMs like ChatGPT can also be used to generate code for a module or application. Exploratory approaches can also be used in which multiple options are sought from the LLM from which the programmer can choose one.
- Several metrics exist for quantifying different qualities like size and complexity of code. The most common size measure is lines of code (LOC). A number of different metrics exist for quantifying the complexity of code, the most common being the cyclomatic complexity, which is based on the program's internal logic and defines complexity as Halstead's metrics provide different measures for size and complexity based on the tokens in the program.

Self-assessment Exercises

1. What is structured programming and how does it help improve code readability?
2. How does the use of information hiding and coding standards help improve the readability?
3. List a few programming practices that will promote readability of programs.
4. Take the application whose design was given in the previous chapter. Suppose the code for that application was to be stored on GitHub. Specify the folder structure for suitably organizing the code.
5. As a programmer working in a team for developing an application, what are the source code control features that you are most likely to use.
6. Suggest some possibilities on how TDD will function if programming is being done in pairs.
7. Use your favorite unit testing framework and use it to unit test a procedure/class which requires at least one other procedure/class.
8. As a team lead, you are required to review code changes that other programmers have made before accepting them. List some DOs and DONTs for yourself for doing these reviews.
9. Select some solutions in programming language of your choice from a coding contest site. Determine the cyclomatic complexity of these programs. Give these programs to an LLM and ask it to determine the cyclomatic complexity. Compare the results.
10. Determine the size in LOC and Halstead's measure for some of the programs, and determine how strongly correlated these measures are.

References

1. G.M. Weinberg, E.L. Schulman, Goals and performance in computer programming. Hum. Factors **16**(1), 70–77 (1974)
2. C.A.R. Hoare, An axiomatic basis for computer programming. Commun. ACM **12**(3), 335–355 (1969)
3. D. Thomas, A. Hunt, *The Pragmatic Programmer: Your Journey to Mastery* (Addison-Wesley Professional, 2019)
4. S.I. Feldman, Make–a program for maintaining computer programs. Softw. Practice Exper. **9**(3), 255–265 (1979)
5. K. Beck, *Test Driven Development: by Example* (Addison-Wesley Professional, 2002)
6. K. Beck, *Extreme Programming Explained* (Addison-Wesley, 2000)
7. C. Bird, D. Ford, T. Zimmermann, N. Forsgren, E. Kalliamvakou, T. Lowdermilk, I. Gazit, Taking flight with copilot: early insights and opportunities of AI-powered pair-programming tools. Queue **20**(6), 35–57 (2022)
8. S. Barke, M.B. James, N. Polikarpova, Grounded copilot: how programmers interact with code-generating models, in *Proceedings of the ACM on Programming Languages 7, OOPSLA1* (2023), pp. 85–111
9. M. Lynch, How to do code reviews like a human (part one), 2017 (March, 2024)
10. M. Lynch, How to do code reviews like a human (part two), 2017 (March, 2024)
11. R.B. Grady, T.V. Slack, Key lessons learned in achieving widespread inspection use. IEEE Softw. 48–57 (1994)
12. E.F. Weller, Lessons learned from three years of inspection data. IEEE Softw. 38–53 (1993)
13. P. Jalote, *Software Project Management in Practice* (Addison-Wesley, 2002)
14. M. Halstead, *Elements of Software Science* (Elsevier North-Holland, 1977)
15. W. Harrison, K. Magel, R. Kluczny, A. DeKock, Applying software complexity metrics to program maintenance. IEEE Comput. 65–79 (1982)
16. S.D. Conte, H.E. Dunsmore, V.Y. Shen, *Software Engineering Metrics and Models* (The Benjamin/Cummings Publishing Company, 1986)

Testing 7

Testing is the most widely used approach for verifying if the code of an application performs as intended. There are other approaches for verification also, e.g. formal verification, model checking, and code reviews. However, due to various challenges these other approaches present, testing remains the most widely used approach. Even if other approaches are employed, testing is still necessary, as it is the primary method to observe the actual behaviour of the application in execution.

During testing, the software under test (SUT) is executed with a set of test cases, and the behavior of the system for these test cases is evaluated to determine if it performes as expected. The primary purpose of testing is to increase confidence in the functioning of SUT. Given that testing is extremely expensive and can consume significant effort, an additional practical goal is to achieve the desired confidence as efficiently as possible. The effectiveness and efficiency of testing depends critically on the selected test cases. Therefore, much of this chapter is devoted to test case selection. In this chapter we will discuss:

- Basic concepts and definitions related to testing, like error, fault, failure, test case, test suite, etc.
- The testing process—how testing is planned and testing of a SUT is done.
- Test case selection using black-box testing approaches.
- Test case selection using white-box testing approaches.
- How LLMs can be used for designing test cases.
- Metrics like coverage and reliability that can be employed during testing.

P. Jalote, *A Concise Introduction to Software Engineering*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-74318-4_7

7.1 Testing Concepts

In this section we first define common terms used in discussions related to testing. Then we discuss some basic issues related to how testing is performed, and the importance of psychology of the tester.

7.1.1 Failure, Fault, Error and Limits of Testing

While discussing testing we commonly use terms like *error*, *fault* and *failure*. Let us start by defining these concepts clearly [1].

Failure of a system or a component is its inability to perform a required function according to its specifications. A software failure occurs if the behavior of the software deviates from the specified behavior. Failures may result from functional or performance issues and occur only when the software or system is in operation.

Fault is a condition that causes a system to fail in performing its required function. A fault is the basic reason for software malfunction and is practically synonymous with the commonly used term *bug*, or the somewhat more general term *defect*. It is due to presence of faults (or defects) in the software that leads to failure during its operation.

The term *error* is used in two different contexts. It refers to the discrepancy between a computed, observed, or measured value and the true, specified, or theoretically correct value. In this sense, error refers to the difference between the actual output of a software and the correct output of the software. Error also refers to human action that results in software containing a defect or fault.

There are some implications of these definitions. The presence of an error (in an output or state) implies that a failure must have occurred, and the observance of a failure implies that a fault must be present in the system (which also implies that some programmer made some error while writing code.) However, the presence of a fault does not imply that a failure must occur. The presence of a fault in a system only implies that the fault has a *potential* to cause a failure. Whether a fault actually manifests itself in a certain time duration depends on how the software is executed.

There are direct consequences from these implications on testing. If no errors are observed during testing, we cannot conclusively determine the presence or absence of faults in the system. If, on the other hand, a failure is observed, it indicates the presence of some faults in the system. Therefore, there is a fundamental limit of testing—it can never fully guarantee that the application code is defect-free. The famous saying by Dijkstra "Testing can show the presence of bugs, not their absence" succinctly captures a fundamental limit of testing.

Due to this fundamental limit, the goal of testing is more practical—to increase confidence in the application identifying as many bugs as possible, so after testing, only a few bugs remain and even those are such that they are encountered rarely, in some very special cases and do not cause failures in common use cases of the application.

Even this practical goal of detecting as many defects as possible is hard to achieve, as it requires far more extensive testing, which consumes a lot more resources and time, than may be feasible. Therefore, while aiming to detect as many defects as possible so that the vast majority of common use cases work correctly, testers must consider the cost in terms of time and resources. This poses additional limitation on practical testing—only a limited number of test cases can be employed during testing. Hence the importance of test case design.

It should also be pointed out that during the testing process, only failures are observed, by which the presence of faults is deduced. That is, testing only reveals the presence of faults. The actual faults are identified by separate activities, commonly referred to as "debugging." In other words, after testing reveals the presence of faults, the expensive task of debugging must be performed. After removing the faults (i.e. fixing the bug), the software must be tested again to ensure that fixing has been done properly. This is one of the reasons why testing is so expensive.

7.1.2 Test Case, Test Suite, and Test Scripts

So far we have used the terms *test case* or *set of test cases* informally. Let us define them more precisely. A *test case* (often called a *test*) comprises a set of test inputs and execution conditions, designed to exercise the SUT in a particular manner [1]. Generally, a test case also specifies the expected outcome for the test case. A group of related test cases that are typically executed together to test some specific behavior or aspects of the SUT is often referred to as a *test suite*.

Note that in a test case, test inputs and execution conditions are distinct components. Test inputs are the specific values of parameters or other inputs given to the SUT either by the user or another program. The execution conditions, on the other hand, reflect the state of the system and environment which also impact the behavior of the SUT. For example, when testing a function that adds a record to a database if it does not already exist, the behavior depends on both the input record as well as the state of the database. A test case needs to specify both. For example, a test case for this function might specify a record *r* as input, and specify that the state of the database be such that *r* already exists in it.

Testing can be performed manually with the tester executing the test cases in the test suite and then checking if the behavior matches the specifications. This is a very cumbersome process, specially when the test suite contains a large number of test cases. It becomes even more cumbersome since the test suite often needs to be executed every time the SUT is changed. Therefore, the current trend is to automate testing.

Note that executing test cases requires more than just the logical design of test cases (each of which specifies the condition to be tested and the nature of inputs and outputs). For actual testing, actual inputs or values to be given to the program for executing the test case have to be defined. And the actual behaviour or output values expected need to be specified, to check if the test has failed or not. Knowing the actual behavior for given inputs implies that that there is a *test oracle* who knows the correct output for those inputs. The test oracle is generally the test case designer, who has to determine for each inputs what is the correct output—which can be extremely challenging for some inputs. So, if the test case design specifies two non-zero integers for executing a (say multiplication) test case, the actual numbers have to be specified (say 5 and 9). And the actual output (say 45) needs to be known so the tester can check if the software behaved correctly.

With automated testing, a test case is typically a function (or a method), that performs all the activities of a test case—it sets the test data and the test conditions, invokes the SUT according to the test case, and compares the results returned with expected results. Executing a test case essentially means executing this function. A test script is a program calling these test functions in a sequence.

7.1.3 Psychology of Testing

In testing, the software under test (SUT) is executed with a set of test cases with the desired goal that the test cases will detect most of the defects (as detecting all the defects is not feasible.) Designing good test cases is critical for effective testing, and since this is a creative activity, the psychology of the person developing the test cases and performing the testing becomes important.

The primary purpose of testing is to detect the errors that may be present in the program. Therefore, one should not start testing with the intent of proving that a program works; rather the intent should be to demonstrate that a program does not work and reveal any defect that may exist. Due to this, testing has been defined as the process of executing a program with the intent of finding errors [2].

This emphasis on the proper intent of testing is not a trivial matter because test cases are designed by human beings, and human beings tend to perform actions to achieve their intended goals. So, if the goal is to demonstrate that a program works, we may consciously or subconsciously select test cases that support that goal, undermining the basic purpose of testing. Conversely, if the intent is to show that the program does not work, we will challenge our intellect to find test cases toward that end, thereby detecting more defects. Testing is essentially a destructive process, where the tester has to treat the program as an adversary that must be beaten revealing errors. This is why many organizations employ *independent testing* where a separate team not involved in building the system, conducts this testing.

7.1.4 Levels of Testing

Testing is relied upon to detect the defects remaining from earlier stages in the application development, in addition to the faults introduced during coding itself. Due to this, different levels of testing are used in the testing process; each level of testing aims to test different aspects of the system.

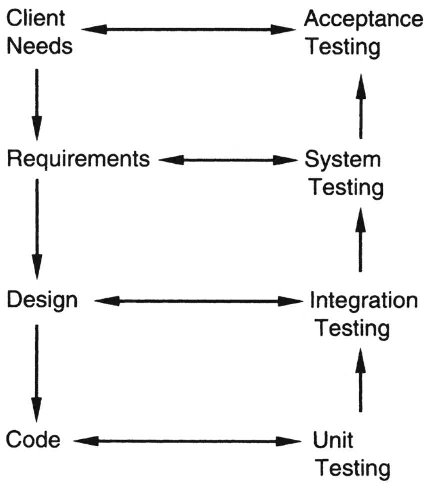


Fig. 7.1 Levels of testing

The primary levels are unit testing, integration testing, system testing, and acceptance testing. These levels attempt to detect different types of faults. The relationship between faults introduced in different phases, and the levels of testing is shown in Fig. 7.1.

The first level of testing is called *unit testing*, which we discussed in the previous chapter. Unit testing focuses on verification of the code produced by individual programmers, and is typically done by the programmer of the module. Generally, a module is offered by a programmer for integration and use by others only after it has been unit tested satisfactorily.

The next level of testing is often called *integration testing*. At this level, unit tested modules are combined into subsystems, which are then tested. The goal here is to ensures that the modules integrate properly, with an emphasis on testing the interfaces between modules. This activity can be considered testing the design.

The next levels are *system testing* and *acceptance testing*. These levels involve testing the entire application against the requirement specifications to ensure that the software meets its requirements. This is an extensive validation exercise which, for large projects, can last for weeks or months. Acceptance testing is often performed with realistic data from the client to demonstrate that the software functions satisfactorily in its intended environment. Acceptance testing essentially tests if the system satisfactorily solves the problems for which it was commissioned.

These levels of testing are performed when an application is being built from the components that have been coded. There is another level of testing, called *regression testing*, that is performed when some changes are made to an existing system. We know that changes are fundamental to software; any software must undergo changes.

However, when modifications are made to an existing system, testing also has to be done to make sure that the modification has not had any undesired side effect of making some of the earlier services faulty. That is, besides ensuring the desired behavior of the new services, testing has to ensure that the desired behavior of the old services is maintained. This is the task of regression testing.

For regression testing, some test cases that have been executed on the old system are maintained, along with the output produced by the old system. These test cases are executed again on the modified system and its output compared with the earlier output to make sure that the system is working as before on these test cases. This frequently is a major task when modifying existing systems.

7.1.5 Test Plan

When testing a SUT, particularly for higher levels of testing, a *test plan* is often essential. A test plan should specify for a SUT, approach for testing, features to be tested, test deliverables, and schedule and task allocation. *Features to be tested* include all software features and combinations of features that should be tested. These may include functionality, performance, design constraints, and other attributes. The *approach* for testing specifies the overall methodology to be followed in the current project. It may include *testing criteria* for designing and evaluating the set of test cases. *Testing deliverables* could include the test cases used, detailed results of testing (including the list of defects found), test summary report, and data about code coverage. The test plan should also outline the schedule and effort to be spent on different testing activities, specify who is responsible for testing, and the identify tools to be used.

As part of the test planning, it is desirable to specify all the test cases that will be used for testing. A test case specification explains the inputs to be used in the test cases, conditions being tested, and outputs expected. If test cases are documented, the specifications might look like a table as shown in Fig. 7.2.

With testing frameworks and automated testing, the testing scripts can serve as test case specifications, as they clearly show what inputs are being given and what output to expect. With suitable comments, the intent of the test case can also be easily specified.

There are some good reasons for specifying test cases before using them in testing. Given the limitations of testing and its heavy reliance on test cases, ensuring that the set of test cases is of high quality is crucial. Evaluation of test cases is often conducted through test case review, which requires them to be specified.

Another reason for documenting the test cases in some form is that this allows the tester to see the testing of the unit in totality and understand the effect of the total

Fig. 7.2 Test case specifications

Requirement Number	Condition to be tested	Test data and settings	Expected output

set of test cases. This type of evaluation is hard to do in on-the-fly testing where test cases are determined as testing proceeds. It also allows optimizing the number of test cases as evaluation of the test suite may show that some test cases to be redundant.

7.2 Test Case Design

The effectiveness of testing depends significantly on the quality of test cases used. The goal of testing a SUT is to detect most (hopefully all) of the defects, through as small a set of test cases as possible. Due to this basic goal, it is important to select test cases carefully—best test cases are those with a high probability of detecting defects, and whose execution gives a confidence that the absence of failures during testing implies minimal (hopefully none) defects in the software.

There are two basic approaches to designing the test cases to be used in testing: black-box and white-box. In black-box testing, test cases are decided solely on the basis of the requirements or specifications of the program or module, and the internals of the module or the program are not considered for selection of test cases. In white-box testing, test cases are designed from the structure of the code. We will first discuss some techniques for generating test cases for black-box testing. White-box testing is discussed after that. LLMs can provide another method of generating test cases–we will also briefly discuss how LLMs may be used for generating test cases.

7.2.1 Black-Box Testing

In black-box testing, the tester only knows the inputs that can be given to the SUT and what output the SUT should give. In other words, the basis for deciding test cases is the requirements or specifications of the application or module and internal working of the programs is not considered. This form of testing is also called functional or behavioral testing.

It should be clear that exhaustive testing, in which all the possible inputs are tried during testing, is impossible to perform. Randomly generating test inputs are not likely to result in effective test cases. There are a number of techniques or heuristics for selecting test cases that have been found to be successful in detecting errors. Here we mention some of these techniques.

Equivalence Class Partitioning

Because exhaustive testing is impractical, the next natural approach is to divide the input domain into a set of equivalence classes, so that if the program works correctly for one value in a class it will work correctly for all the other values in that class. If we can indeed identify such classes, testing the program with one value from each equivalence class becomes equivalent to doing an exhaustive test of the program. The equivalence class partitioning method [2] tries to approximate this ideal.

An equivalence class is a group of the inputs for which the behavior of the system is specified or expected to be similar. Each group of inputs that elicits different behavior from others is considered a separate equivalence class. The rationale behind forming equivalence classes this way is the assumption that if the specifications require the same behavior for each element in a class of values, the program is likely constructed to either succeed or fail for all values in that class. For robust software, we must also consider invalid inputs. That is, we should define equivalence classes for invalid inputs as well.

Equivalence classes are usually formed by considering each condition specified on an input as defining a valid equivalence class and one or more invalid equivalence classes. For example, if an input condition specifies a range of values (say, $0 <$ count $<$ Max), then we form a valid equivalence class within that range and two invalid equivalence classes: one with values less than the lower bound (i.e., count < 0) and the other with values higher than the higher bound (count $>$ Max). If the input specifies a set of values and the requirements specify different behavior for different elements in the set, then a valid equivalence class is formed for each of the elements in the set and an invalid class is formed for an entity not belonging to the set.

It is often useful to consider equivalence classes in the output as well. For an output equivalence class, the goal is to have inputs such that the output for that test case lies within the output equivalence class. As an example, consider a program for determining rate of return or an investment. There are three clear output equivalence classes—positive rate of return, negative rate of return, and zero rate of return. During testing, it is important to test for each of these, by providing inputs that generate each of these three outputs. Determining test cases for output classes can be more challenging, but output classes often reveal errors that are not exposed by considering the input classes alone.

Once equivalence classes are selected for each of the inputs, the next step is to select suitable test cases. There are different strategies for selecting the test cases. One strategy is to select each test case to cover as many valid equivalence classes as possible, with one separate test case for each invalid equivalence class. A somewhat better strategy which requires more test cases is to have each test case cover at most one valid equivalence class for each input, and have one separate test case for each invalid equivalence class. In this latter case, the number of test cases for valid equivalence classes is equal to the largest number of equivalence classes for any input, plus the total number of invalid equivalence classes.

As an example, consider a program that takes two inputs—a string s of length up to N and an integer n. The program is to determine the top n highest occurring characters in s. The tester believes that the programmer may deal with different types of characters separately. One set of valid and invalid equivalence classes for this is shown in Table 7.1.

Using these as the equivalence classes, we must select the test cases. A test case for this is a pair of values for s and n. With the first strategy for deciding test cases, one test case could be: s as a string of length less than N containing lowercase, uppercase,

Table 7.1 Valid and invalid equivalence classes

Input	Valid equivalence classes	Invalid equivalence classes
s	EQ1: Contains numbers EQ2: Contains lowercase letters EQ3: Contains uppercase letters EQ4: Contains special characters EQ5: String length between 0-N	IEQ1: non-ASCII characters IEQ2: String length > N
n	EQ6: Integer in valid range	IEQ3: Integer out of range

numbers, and special characters; and *n* as the number 5. This one test case covers all the valid equivalence classes (EQ1 through EQ6). Then will have one test case each for covering IEQ1, IEQ2, and IEQ3. That is, a total of four test cases is needed.

With the second approach, where one test case can cover only one equivalence class for one input, one test case could be: a string of numbers, and the number 5. This covers EQ1 and EQ6. Then we will need test cases for EQ2 through EQ5, and separate test cases for IEQ1 through IEQ3.

Boundary Value Analysis

It has been observed that programs that work correctly for a set of values in an equivalence class may fail on some special values that lie on the boundary of the equivalence class. Test cases with values on the boundaries of equivalence classes are therefore likely to be "high-yield" test cases, and selecting such test cases is the aim of boundary value analysis. In boundary value analysis [2], we choose an input for a test case from an equivalence class, such that the input lies at the edge of the equivalence classes. Boundary values for each equivalence class, including those of the output, should be covered.

In case of ranges, boundary value analysis involves selecting the boundary elements of the range and an invalid value just beyond the two ends (for the two invalid equivalence classes). For example, if the range is $0.0 \le x \le 1.0$, then the test cases are 0.0, 1.0 (valid inputs), and -0.1, and 1.1 (for invalid inputs). Similarly, if the input is a list, attention is focused on the first and last elements of the list.

Like in equivalence class partitioning, in boundary value analysis we first determine boundary values for each variable that should be exercised during testing. Where there are multiple inputs, how should the set of test cases be formed covering the boundary values? Suppose each input variable has a defined range. Then there are six boundary values—the extreme ends of the range, just beyond the ends, and just before the ends. If an integer range is min to max, then the six values are $min-1$, min, $min+1$, $max-1$, max, $max+1$. Suppose there are n such input variables. There are two strategies for combining the boundary values for the different variables in test cases.

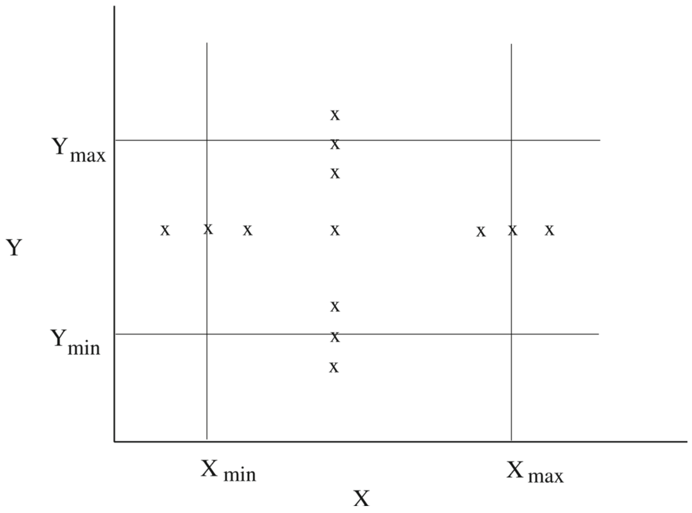


Fig. 7.3 Test cases for boundary value analysis

In the first strategy, we select the different boundary values for one variable, and keep the other variables at some nominal value. Additionally we select one test case consisting of nominal values of all the variables. In this case, we have $6n + 1$ test cases. For two variables X and Y, the 13 test cases will be as shown in Fig. 7.3.

Pairwise Testing

There are typically numerous parameters that determine the behavior of a software system. These parameters can be direct inputs to the software or implicit settings like device configurations. Each parameter can take different on values, and certain values may cause the software to malfunction. Many of the defects in software generally involve one condition, that is, some special value of one of the parameters. Such a defect is called a single-mode fault [3]. Examples include a software not able to print for a particular type of printer, a software that cannot compute fare properly when the traveler is a minor, and a telephone billing software that does not compute the bill properly for a particular country.

Single-mode faults can be detected by testing different values for each parameters. If a system has n parameters, and each one of them can take m different values (or m different classes of values, each class being considered as the same for purposes of testing as in equivalence class partitioning), then with each test case we can test one different value of each parameter. In other words, testing all these values can be done in m test cases.

However, not all faults are single-mode and there are combinations of inputs that reveal the presence of faults: for example, a telephone billing software that does not compute correctly for nighttime calling (one parameter) to a particular country (another parameter), or an airline ticketing system that has incorrect behavior when a minor (one parameter) is traveling business class (another parameter) and not staying over the weekend (third parameter). These multi-mode faults can be revealed during testing by trying different combinations of the parameter values—an approach called *combinatorial testing*.

Unfortunately, full combinatorial testing is impractical. For a system with n parameters, each having m values, the number of different combinations is n^m. For a simple system with 5 parameters, each having 5 different values, the total number of combinations is 3,125. Clearly, for complex systems that have many parameters and each parameter may have many values, a full combinatorial testing is not feasible and practical techniques are needed to reduce the number of tests.

Research indicates that most software faults are revealed on some special single values or by interactions between pairs of values [4]. That is, most faults tend to be either single-mode or double-mode. For testing for double-mode faults, we need not test the system with all the combinations of parameter values, but need to test such that all combinations of values for each pair of parameters are exercised. This is called *pairwise testing*.

In pairwise testing, all pairs of values are tested. For n parameters, each with m values, the number of pairs between any two parameters is $m * m$. The first parameter will have these many pairs with each of the remaining $n - 1$ parameters, the second one will have new pairs with $n - 2$ parameters (as its pairs with the first are already included in the first parameter pairs), the third will have pairs with $n - 3$ parameters, and so on. That is, the total number of pairs is $m * m * n * (n - 1)/2$.

The objective of pairwise testing is to create a set of test cases that cover all the pairs. As there are n parameters, a test case is a combination of values of these parameters and will cover $(n - 1) + (n - 2) + ... = n(n - 1)/2$ pairs. In the best case when each pair is tested exactly once, m^2 different test cases will be needed to cover all the pairs.

As an example, consider a software product being developed for multiple platforms that uses the browser as its interface. Suppose the software is being designed to work for three different operating systems and three different browsers. In addition, as the product is memory intensive there is a desire to test its performance under different levels of memory. So, we have the following three parameters with their different values:

```
Operating System: Windows, Android, Linux
Memory Size: 2GB, 4GB, 8GB
Browser: Chrome, Firefox, Safari
```

For discussion, we can say that the system has three parameters: A (operating system), B (memory size), and C (browser). Each of them can have three values which we will refer to as $a_1, a_2, a_3, b_1, b_2, b_3$, and c_1, c_2, c_3. The total number of pairwise combinations is 9 * 3 = 27. The number of test cases, however, to cover all the pairs is much less. A test case consisting of values of the three parameters covers

Table 7.2 Test cases for pairwise testing

A	B	C	Pairs
a1	b1	c1	(a1,b1) (a1,c1) (b1,c1)
a1	b2	c2	(a1,b2) (a1,c2) (b2,c2)
a1	b3	c3	(a1,b3) (a1,c3) (b3,c3)
a2	b1	c2	(a2,b1) (a2,c2) (b1,c2)
a2	b2	c3	(a2,b2) (a2,c3) (b2,c3)
a2	b3	c1	(a2,b3) (a2,c1) (b3,c1)
a3	b1	c3	(a3,b1) (a3,c3) (b1,c3)
a3	b2	c1	(a3,b2) (a3,c1) (b2,c1)
a3	b3	c2	(a3,b3) (a3,c2) (b3,c2)

three combinations (of A–B, B–C, and A–C). Hence, in the best case, we can cover all 27 combinations by 27/3 = 9 test cases. These test cases are shown in Table 7.2, along with the pairs they cover.

As should be clear, generating test cases to cover all the pairs is not a simple task. The minimum set of test cases is that in which each pair is covered by exactly one test case. Often, it will not be possible to generate the minimum set of test cases, particularly when the number of values for different parameters is different. Various algorithms programs are available online to help generate these test cases to cover all the pairs. In [4] an example is given in which for 13 parameters, each having three distinct values, all pairs are covered in merely 15 test cases, while the total number of combinations is over 1 million!

Pairwise testing is an effective strategy for testing large software systems with multiple parameters with distinct functioning expected for different settings of these parameters. It is also a practical approach for testing general-purpose software products that are expected to run on different platforms and configurations, or a system that is expected to work with different types of systems.

Special Cases

It has been seen that programs often exhibit incorrect behavior when inputs form some special cases. The reason is that in programs, some combinations of inputs need special treatment, and providing proper handling for these special cases is easily overlooked. For example, in an arithmetic routine, if there is a division and the divisor is zero, some special action has to be taken, which could easily be forgotten by the programmer. These special cases form particularly good test cases, which can reveal errors that are usually not be detected by other test cases.

Special cases often depend on the data structures and the function of the module. Incorrect assumptions made by the programmers about the application or the environment, perhaps stemming from incomplete specifications, can form good special condition test cases. There are no rules to determine special cases, and the tester has to use his intuition and experience to identify such test cases. Consequently, determining special cases is also called *error guessing*.

Psychology is particularly important for error guessing. The tester should play the "devil's advocate" and try to guess the incorrect assumptions the programmer could have made and the situations the programmer could have overlooked or handled incorrectly. Essentially, the tester needs to identify error-prone situations, then write test cases for these situations. For example, in the problem of finding the number of different words in a file (discussed in earlier chapters) some of the special cases can be: file is empty, only one word in the file, only one word in a line, some empty lines in the input file, presence of more than one blank between words, all words are the same, the words are already sorted, and blanks at the start and end of the file.

State-Based Testing

There are some systems that are essentially state-less where the same inputs always yield the same outputs or exhibit the same behavior. Many batch processing systems, computational systems, and servers fall in this category. At a smaller level, most functions are supposed to behave in this manner. There are, however, many systems whose behavior is state-based in that for identical inputs they behave differently at different times and may produce different outputs, depending on the state of the system. In other words, the behavior and outputs of the system depend not only on the inputs provided, but also on its state. The state of the system depends on the past inputs the system has received. In other words, the state represents the cumulative impact of all the past inputs on the system. In software, many large systems fall in this category as past state is captured in databases or files and used to control the behavior of the system. For such systems, another approach for selecting test cases is the state-based testing approach [5].

Theoretically, any software that saves state can be modeled as a state machine. However, the state space of any reasonable program is almost infinite, as it is a cross product of the domains of all the variables that form the state. For many systems the state space can be partitioned into a few states, each representing a logical combination of values of different state variables which share some property of interest [6]. If the set of states of a system is manageable, a state model of the system can be built. A state model for a system has four components:

- *States.* Represent the impact of the past inputs to the system.
- *Transitions.* Represent how the state of the system changes from one state to another in response to some events.
- *Events.* Inputs to the system.
- *Actions.* The outputs for the events.

The state model shows the state transitions that occur and the actions performed in a system in response to events. When a state model is built from the requirements of a system, we can only include the states, transitions, and actions that are stated in the requirements or can be inferred from them. If more information is available from the design specifications, then a richer state model can be built.

For example, consider the student survey example discussed in Chap. 4. According to the requirements, a system is to be created for taking a student survey. The student takes a survey and is returned the current result of the survey. The survey result can be up to five surveys old. We consider an architecture which has a cache between the server and the database, and in which the survey and results are cached and updated only after 5 surveys, on arrival of a request. The proposed architecture has a database at the back, which may go down.

To create a state machine model of this system, we notice that of a series of six requests, the first 5 may be treated differently. Hence, we divide it into two states: one representing the the receiving of 1–4 requests (state 1), and the other representing the receiving of request 5 (state 2). Next we observe that the database can be up or down, and it can go down in any of these two states. However, the behavior of requests, if the database is down may be different. Hence, we create another pair of states (states 3 and 4). Once the database has failed, then the first 5 requests are serviced using old data. When a request is received after receiving 5 requests, the system enters a failed state (state 5), in which it does not give any response. When the system recovers from the failed state, it must update its cache immediately, hence going to state 2. The state model for this system is shown in Fig. 7.4 (*i* represents an input from the user for taking the survey).

The state model often requires information about the design of the system. In the example above, the knowledge of the architecture is utilized. Sometimes making the state model may require detailed information about the design of the system. For example, for state modeling of a class we need to know it attributes and methods. Therefore, the state-based testing can be considered as somewhat between black-box and white-box testing. Such strategies are sometimes called *gray box testing*.

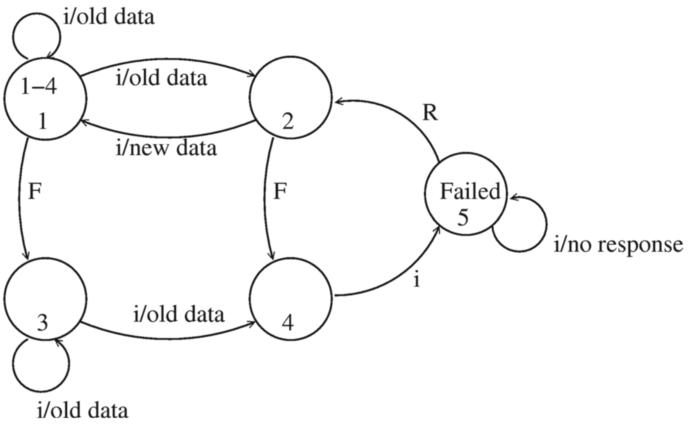


Fig. 7.4 State model for the student survey system

Table 7.3 Test cases for a state based testing criteria

S. no.	Transition	Test case
1	$1 \rightarrow 2$	req()
2	$1 \rightarrow 2$	req();req();req();req();req();req()
3	$2 \rightarrow 1$	seq for 2; req()
4	$1 \rightarrow 3$	req();fail()
5	$3 \rightarrow 3$	req();fail();req()
6	$3 \rightarrow 4$	req();fail();req();req();req();req();req()
7	$4 \rightarrow 5$	seq for 6; req()
8	$5 \rightarrow 2$	seq for 6; req();recover()

Given a state model of a system how should test cases be generated? Many coverage criteria have been proposed [7]. We discuss only a few here. Suppose the set of test cases is T. Some of the criteria are:

- **All transition coverage (AT).** T must ensure that every transition in the state graph is exercised.
- **All transitions pair coverage (ATP).** T must execute all pairs of adjacent transitions. (An adjacent transition pair comprises of two transitions: an incoming transition to a state and an outgoing transition from that state.)
- **Transition tree coverage (TT).** T must execute all simple paths, where a simple path is one which starts from the start state and reaches a state that it has already visited in this path or a final state.

The first criterion states that all transitions get fired during testing. This also ensures that all states are visited. The transition pair coverage is a stronger criterion requiring that all combinations of incoming and outgoing transitions for each state must be exercised by T. If a state has two incoming transitions t1 and t2, and two outgoing transitions t3 and t4, then a set of test cases T that executes t1;t3 and t2;t4 satisfying AT. However, to satisfy ATP, T must also ensure execution of t1;t4 and t2;t3. The transition tree coverage is named in this manner as a transition tree can be constructed from the graph and then used to identify the paths. In ATP, we go beyond transitions, and state that different paths in the state diagram should be exercised during testing. ATP generally includes AT.

For the example above, the set of test cases for AT are given below in Table 7.3. Here req() means that a request for taking the survey should be given, fail() means that the database should be failed, and recover() means that the failed database should be recovered.

As we can see, state-based testing draws attention to the states and transitions. Even in the above simple case, we can see different scenarios get tested (e.g., system behavior when the database fails, and system behavior when it fails and recovers thereafter). Many of these scenarios are easy to overlook if test cases are designed only by looking at the input domains. The set of test cases is richer if the other criteria are used. For this example, we leave it as an exercise to determine the test cases for other criteria.

7.2.2 White-Box Testing

Black-box testing primarily focuses on the functionality rather than implementation of the program. White-box testing, on the other hand, is concerned with testing the implementation of the program. The intent of this testing is not to exercise all the different input or output conditions (although that may be a by-product) but to exercise the different programming structures and data structures used in the program. White-box testing is also called *structural testing*, and we use the two terms interchangeably.

To test the structure of a program, structural testing aims to achieve test cases that force the desired coverage of different structures. Various criteria have been proposed for this. Unlike the criteria for functional testing, which are frequently imprecise, the criteria for structural testing are generally quite precise as they are based on program structures, which are formal and precise. Here we further discuss control flow-based testing, which is based on the control flow graph of the programs and is the most commonly used in practice. It should be mentioned that there are other approaches for white-box testing as well, for example, those that are based on the data-flow graph of the software. We will not discuss such approaches here.

In the *control flow graph* of a program, a node represents a block of statements that is always executed together, and an edge from node i to node j represents a possible transfer of control from the block represented by node i to the code block represented by node j. A node which includes the first statement of the program is the *start* node, and a node from where the program exits is called an *exit* node (for simplicity we assume that a program has a single exit node.)

Statement Coverage

Perhaps the simplest criterion based on the control flow graph is *statement coverage*, which ensures that each statement of the program should be executed at least once during testing, i.e. a possible goal for testing is to achieve 100% statement coverage. This is also called the *all-nodes* criterion [8].

Even the 100% coverage criterion is not very strong, and can leave errors undetected. For example, an `if` statement in the program without an `else` clause, the statement coverage criterion for this statement will be satisfied by a test case that evaluates the condition to true. No test case is needed that ensures that the condition in the `if` statement evaluates to false. This is a serious shortcoming because decisions in programs are potential sources of errors. As an example, consider the following function to compute the absolute value of a number:

```
int abs (x)
int x;
{
      if (x >= 0) x = 0 - x;
      return (x)
}
```

This program is clearly wrong. Suppose we execute the function with the set of test cases { x=0 } (i.e., the set has only one test case). The statement coverage criterion will be satisfied by testing with this set, but the error will not be revealed.

Branch Coverage

A more general coverage criterion is *branch coverage*, which suggests that each edge in the control flow graph be traversed at least once during testing. In other words, 100% branch coverage requires that each decision in the program be evaluated to true and false values at least once during testing. Testing based on branch coverage is often called *branch testing*. The 100% branch coverage criterion is also called the *all-edges* criterion [8]. 100% branch coverage implies 100% statement coverage, as each statement is a part of some branch. In the preceding example, a set of test cases satisfying this criterion will detect the error.

A limitation of branch coverage is that if a decision has many conditions in it, a decision can evaluate to true and false without actually exercising all the conditions. For example, consider the following function that checks the validity of a data item. The data item is valid if it lies between 0 and 100.

```
int check(x)
int x;
{
     if ((x >= 0) && (x <= 200))
          check = True;
     else check = False;
}
```

The module is incorrect, as it is checking for $x \leq 200$ instead of 100 (perhaps a typing error made by the programmer). Suppose the module is tested with the following set of test cases: { x = 5, x = -5 }. The branch coverage criterion will be satisfied for this module by this set. However, the error will not be revealed, and the behavior of the module remains consistent with its specifications for all test cases in this set. Thus, the coverage criterion is satisfied, but the error is not detected. This occurs because the decision is evaluating to true and false because of the condition ($x \geq 0$). The condition ($x \leq 200$) never evaluates to false during this test, hence the error in this condition is not revealed.

This problem can be resolved by requiring that all conditions evaluate to true and false. However, situations can occur where a decision may not get both true and false values even if each individual condition evaluates to true and false. An obvious solution to this problem is to require decision/condition coverage, where all the decisions and all the conditions in the decisions take both true and false values during the course of testing.

It should be pointed out that none of these criteria is sufficient to detect all kind of errors in programs. For example, if a program is missing some control flow paths that are needed to check for a special value (like pointer equals nil and divisor equals zero), or not implementing some requirements, then even executing all the paths in the program will not detect the error. This is a fundamental limit of white-box testing—it can only test what has been implemented cannot test for what simply has not been implemented but should have been.

Test Case Generation and Tool Support

Once a coverage criterion is chosen for testing, two primary challenges arise. The first is to determine how well a set of test cases satisfy the criterion, and the second is to generate a set of test cases for a given coverage. Generally tools are used to determine the level of coverage achieved in some testing. These tools often also provide information about the parts remaining uncovered in testing, that can guide the selection of further test cases. However, a fully automated tool for selecting test cases to satisfy a coverage criterion is generally not possible.

One method for using white-box test cases is to select test cases using black-box approaches and then augment them based on information provided by these tools about what portions of the code has been missed in testing.

Selecting test cases to execute parts of as yet unexecuted code is often very complex, even with the aid of tools. Due to these challenges and other practical considerations, instead of requiring 100% coverage of statements or branches, the goal might be to achieve some acceptably high percentage (but less than 100%).

Several open source tools are available for statement and branch coverage, the criteria that are used most often. These tools also provide higher-level coverage metrics such as function coverage, method coverage, and class coverage. To get the coverage data, the execution of the program during testing has to be closely monitored. This requires that the program be instrumented so that required data can be collected.

7.3 Using LLMs for Generating Test Cases

We have seen that designing test cases is a critical task in testing that consumes a significant amount of effort by programmers/testers. As with other tasks in software development, LLMs can be used to streamline test case design as well, so as to reduce the effort needed by programmers, thereby enhancing productivity. Here we discuss how LLMs may be used for generating test scripts at unit testing level and test plans at system testing level.

Generating Test Scripts for Unit Testing

For unit testing, often test scripts are used. It will be helpful if a programmer can generate the test scripts using LLMs, which implies that the LLM has to generate the actual inputs as well as develop the assertions for checking the output. To get unit test scripts, we discuss one strategy for giving the prompts to an LLM—this approach was used extensively in experiments reported in [9]]. For a survey of works in using LLMs for testing, reader can refer to [10]. The approach has these steps:

1. Define Role/Persona: Instruct the model to assume the role of an expert in Quality Assurance or Software Testing Engineering.
2. Specify Task and output format: Clearly define the task and the output format. For unit test it can be: "Generate unit tests for the given code using Pytest", as the specification of the framework specifies the format of the test cases. We can also add further explanations like "covering all possible edge cases"
3. Use Delimiters for Code Input: Enclose code snippets using appropriate delimiters to mark distinct parts of the input. Contextual comments within the code can provide additional information.
4. Task Decomposition: If there are multiple units (e.g. functions) in the code, then prompt the LLM to generate the test cases for each function separately.
5. Review and Ask for Missing Cases: After the initial response, review the test cases and give follow up prompts to ask for additional test cases or if anything was missed.

An example of such a prompt is:

> Prompt: You are a software testing engineer. Generate unit test cases in pytest for the provided code, ensuring coverage of all potential edge cases.
> Code - '''..... '''

Once the test scripts are generated, they should be reviewed by the programmer for correctness. The scripts can then be executed by the programmer. Most frameworks provide information on execution of test cases like statement and function coverage coverage. Using these results, further prompts can be given to refine the test cases (step 5).

Let us consider giving follow up prompt to generate more test cases for covering missing statements. The prompt for this tells the LLM that some statements were not executed by the test cases it has generated, and ask it to generate more test cases to cover the missing statements. An example of such a prompt is:

> Prompt: The above test cases only give 61% coverage. The following lines from the code file are not being covered given by the line numbers 4, 9–11, 23, 29–33. Please write more test cases to improve the coverage of the unit tests.

This iterative approach may need to be repeated several times. Experience indicates that such follow up prompting helps LLM to generate test cases to improve coverage. Repeating this a couple of times can help, but after a few times improvement may stop [9]. It is important to note that achieving 100% statement coverage may not always be feasible (for example, if there is unreachable code). In such cases, testers should consider alternative methods to augment and extend test cases beyond automated generation.

While LLM can generate test cases, the test cases are not always correct, in that the assertion may be incorrect. Studies indicate that about 30% of the test cases have incorrect assertions (so the test case fails not because the unit being tested has a bug, but because the assertion is wrong) [9]. This requires the test cases to be carefully reviewed by the programmer before accepting them.

Unit test cases can also be generated with copilot, as discussed in the earlier chapter. Generally, in the IDE, after the module has been coded, just giving a comment indicating that the programmer wants to develop test cases, or defining a function whose name starts with "test" kicks-in copilot which suggests an executable test case, or multiple test cases.

Generating Test Cases for System Testing

When generating test cases for an entire application, the situation is somewhat different. Often system level testing is done manually—as emulating user behaviour/inputs is often hard, particularly for an application which has GUI front end. In such a situation, a test plan is made, which can be reviewed to check that it will perform thorough testing, and then executed manually.

As it is for end-to-end testing, the system test plan should be generated from the requirements for the application, so all the requirements are adequately covered. So, for using LLM, the context to be provided is the software requirements specification, and the output that is expected is a test plan. For this situation, the above sequence of steps has to be modified a little. The revised sequence of steps for the process is:

1. Define Role/Persona: Instruct the model to assume the role of an expert in Quality Assurance or Software Testing Engineering.

2. Specify Task and Context: Clearly define the task and the context. The context in this case is the SRS. Use Delimiters for providing the SRS.
3. Specify Output Format: As a test plan is being sought, the output format should be outlined. For example, the LLM can be informed that each test case should specify the condition being tested, the inputs to be provided, and the expected output.
4. Prompt Chaining: As an SRS is typically extensive with many use cases, rather than requesting it to generate test cases for the entire SRS at once, a tester can get more test cases if after giving the SRS, test cases are asked for each use case separately. This also helps the tester to ensure that each requirement is covered.
5. Review and Ask for Additional Cases: After the initial response, review the test cases and give follow up prompts to ask for additional test cases or any missing requirements. For example, if the use cases did not specify exception conditions, the LLM can be prompted to generate the exceptional situations, and then asked to generate test cases for them.

As before, the last step will be used after the generation of test cases from the initial prompt and reviewing them for deficiencies. Then in subsequent prompts, the deficiencies (e.g. it is missing edge cases, or invalid inputs, ...), can be specified one by one and for each new test cases can be asked for. Some techniques can also be specified (e.g. boundary value analysis, special cases) and the LLM can be asked to generate additional test cases using those techniques. An example of the initial prompt for test case generation for an application is:

> Prompt: You are a software engineer. You are provided with the SRS of a Student Event Management portal. The text given below in triple quotes is the System Requirements Specification of this Portal. Go through it, and you will refer to it for answering the questions in upcoming prompts.
> SRS: '''... Student event management system SRS...'''

This prompt defines the role, the task, and sets the context. To generate test cases for use cases, the tester can provide further prompts, each prompt giving one use case and asking for test cases. An example:

> Prompt: Now that you have gone through the SRS of the Student Event Management portal, you are to generate test case designs for each possible use case in a tabular format having the following 4 columns: functionality/condition to be tested, input action/input values, expected output/behaviour, and additional comments.
> For the use case given below in triple quotes, generate all possible test case designs in a tabular form. ''' .... one use case from the SRS ...'''

One can ask the LLM to generate test cases for all the use cases. Experience suggests that it may generate fewer test cases, as compared to prompting it to generate test cases for each use case separately.

It is also beneficial to ask the LLM to consider the SRS and identify exceptional situations which may not have been mentioned in the use cases. Once these situations are identified, it may be asked to generate test cases for those scenarios.

Generating all the desired test cases to achieve comprehensive coverage may not always be feasible and not all the test cases generated relevant. Therefore, it is essential that the test cases are reviewed by the tester and suitably modified and augmented.

7.4 Test Case Execution

After the specification of test cases, the next step in the testing process is their execution. This step is also not straightforward. The test case specifications only specify the set of test cases for the unit to be tested, but, executing them may require construction of driver modules or stubs. It may also require modules to set up the environment as stated in the test plan and test case specifications (e.g. ensure that database has the desired data). Only after these preparations are complete can the test cases be executed.

7.4.1 Using Open Source Testing Tools and Frameworks

Testing is expensive because it needs to be performed whenever code changes, and code changes are frequent. Clearly, executing all the test cases through test scripts, rather than executing manually, can help in speeding up the testing. To achieve this automated testing using test scripts, testing frameworks and tools are essential. When using these test frameworks, tasks such as the setting of the environment, providing inputs for a test case, and checking the result of computation are all handled in the test scripts.

Numerous open source testing frameworks and libraries are available for developers to use for automating testing. For unit testing, frameworks like pytest, unittest, JUnit, are commonly used by programmers. There are a host of more advanced open source tools with more features and capabilities. For example, Selenium allows for end-to-end testing of a web application, and supports multiple programming languages, and all the major browsers and operating systems. Its web-driver allows it to have scripts that take inputs and invoke APIs as if the interaction is through a browser. It also allows record and playback of tests. There are many other such test automation frameworks like Cypress, Playwright, TestCafe.

Regardless of how the test cases are executed, the programmers must decide when to run these testing scripts. Such running testing scripts can be time consuming and resource intensive, it is best to execute them strategically during the development. One common practice is that the application level test cases are executed every time some change are made to the code, i.e. whenever the main branch of the code repository needs to merge some changes into it (typically done by a pull request in gitHub). While the programmers may run these test scripts on their machines before making such requests, there are tools to automatically execute the test sripts on such merge requests. GitHub Actions is one such tool—it executes the specified actions or programs/scripts on defined actions in gitHub. So, this testing can be automated by specifying that the test scripts be executed every time a pull request is made. (This step is more challenging than it appears as execution of test scripts requires the application to be running in the environment is it designed for, and this requires additional steps like hosting the application, that need to be taken before the test scripts can be run.)

7.4.2 Defect Logging and Tracking

During test case execution, defects are inevitably discovered. These defects are then fixed followed by re-testing to verify the fix. To facilitate reporting and tracking of defects found during testing (and other quality control activities), defects found are often logged. Defect logging is particularly important in a large software project which may have hundreds or thousands of defects that are found by different people at different stages of the project. Often the person who fixes a defect is not the person who finds or reports the defect. For example, a tester may find the defect while the developer of the code may actually fix it. In such a scenario, defect reporting and closing cannot be done informally. The use of informal mechanisms may easily lead to defects being found but later forgotten, resulting in defects not getting removed or in extra effort in finding the defect again. Hence, defects found must be properly logged in a system and their closure tracked. Defect logging and tracking is considered one of the best practices for managing a project [11], and is followed by most software organizations.

Let us understand the life cycle of a defect. A defect can be found by anyone at anytime. When a defect is discovered, it is logged in a defect control system, along with sufficient information about the defect. The defect initially enters the "submitted state" essentially implying that it has been logged along with information about it. The job of fixing the defect is then assigned to some person, who is generally the author of the document or code in which the defect is found. The assigned person performs the debugging and fixes the reported defect, after which the defect enters the "fixed" state. However, a defect that is fixed is still not considered as fully done. The successful fixing of the defect is verified. This verification may be done by another person (often the submitter), or by a test team, and typically involves running some tests and reviewing the fix. Once the defect fixing is verified, then the defect can be marked as "closed." In other words, the general life cycle of a defect has three states—submitted, fixed, and closed, as shown in Fig. 7.5. A defect that is not closed is also called open.

This is a typical life cycle of a defect which is used in many organizations (e.g., [12]). Ideally, at the end of the project, no open defects should remain. However, this ideal situation is often not practical for most large systems. In addition to using

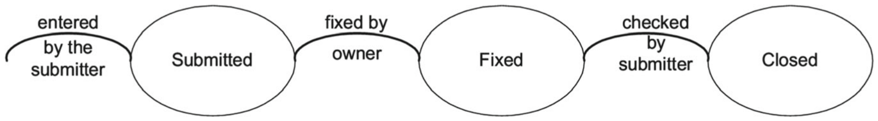


Fig. 7.5 Life cycle of a defect

the log for tracking defects, the data in the log can also be leveraged for analytical purposes. We will discuss some possible analysis later in the chapter.

7.5 Metrics

We have seen that during software testing, the software under examination is executed with a set of predefined test cases. Since the quality of delivered software relies substantially on the quality of testing, a natural question arises regarding the effectiveness of the testing process. The primary purpose of metrics is to try to answer this and other related questions. We will discuss some metrics that may be used.

7.5.1 Coverage Analysis

One of the most widely used approaches to asses the thoroughness of testing involves coverage measures. As discussed previously, some of the common coverage measures that are used in practice include—statement coverage and branch coverage. To use these coverage measures for evaluating the quality of testing, proper coverage analysis tools are employed which not only report the coverage achieved during testing but also highlight the areas that remain untested.

Organizations often build guidelines establishing the level of coverage that must be achieved during testing. Generally, the coverage requirement is higher for unit testing, but lower for system testing as it is much more difficult to ensure execution of identified blocks when the entire system is being executed. Often the coverage requirement at unit level can be 90% to 100% (keep in mind that 100% may not be always feasible due to unreachable code).

In addition to the coverage of program constructs, coverage of requirements is also often examined. It is for facilitating this evaluation that in test case specification the requirement or condition being tested is mentioned. This coverage is generally established by evaluating the set of test cases to ensure that sufficient number of test cases with suitable data are included for all the requirements. The coverage measure here is the percentage of requirements or their clauses/conditions for which at least one test case exists. Its common practice to aim for complete coverage at the requirement level before deeming testing acceptable.

7.5.2 Defects Data

Defect tracking tools provide insights into the status of identified defects. As discussed previously, a defect may be open (i.e. it has yet not been fixed), or it may be closed (i.e. it has been fixed). From the defect data, various metrics can be observed aiding project teams in assessing the readiness of the application for release.

One measure that can be useful is the total number of open defects. During system testing, which typically focuses on defect discovery without any feature development, the number of open defects ideally decreases as more defects are closed. (Initially, it may even increase as fixing some defects may introduce or unearth more defects.) When the total number of open defects reaches a "reasonably small" level, and this is a judgement, it indicates lower risks associated with releasing the application. At this stage, the open defects are also examined for their severity. A release criteria for an application can be that the total number of open defects should be lower than a threshold, and there are no open defects of "critical" severity.

For a large applications, this last stage of testing when the application is being tested for release and no new features are added can be quite long—weeks or even months. From defects data some metrics can be extracted that are useful in this stage. Defect arrival rate is one such measure which is the the number of new defects that are being logged for an application per unit time (often a week). Generally, the defect arrival rate should go down as testing proceeds, though it may increase at initial stages of testing. If this rate is not reducing sufficiently, it may indicate that some serious issues persist and may require rethinking of the release plan. Release criteria for an application can be stated in terms of defect arrival rate—here also open defects will need to be examined to ensure that critical defects have been closed.

Another measure that can be used is the defect closure rate. When plotted along with the defect arrival rate, it provides a good picture of what is happening during testing. Initially, the defect arrival rate may be more than the closure rate, but as testing proceeds, the defect closure rate should become more than the arrival rate.

7.5.3 Reliability

Upon completion of testing and software delivery, development concludes. It is clearly desirable to know, in quantifiable terms, the reliability of the software being delivered. As reliability heavily correlates with the quality of testing, besides characterizing an important quality property of the product being delivered, reliability estimation has a direct role in project management—it can be used to decide whether enough testing has been done and when to stop testing.

Reliability of a product specifies the probability of failure-free operation of that product for a given time duration. Measuring reliability for software can be an extremely time consuming exercise, hence reliability estimation models are employed rather than actual measurement. Most reliability estimation models require that the occurrence of failure be a random phenomenon. In software even though failures occur due to preexisting bugs, this assumption holds for larger systems.

Let X be the random variable that represents the life of a system. Reliability of a system is the probability that the system has not failed by time t. In other words,

$$R(t) = P(X > t).$$

The reliability of a system can also be specified as the *mean time to failure* (MTTF). MTTF represents the expected lifetime of the system. From the reliability function, it can be obtained as [13]

$$MTTF = \int_0^\infty R(x)dx.$$

Reliability can also be defined in terms of failure intensity which is the failure rate (i.e., number of failures per unit time) of the software at time t.

From the measurement perspective, during testing, measuring failure rate is possible if besides defects, application failures are also logged. A simple way to do this is to compute the number of failures every week or every day during the last stages of testing. Though failures and defects are different, if in the last stages of testing defects that cause failures are fixed soon such that they do not cause multiple failures, defect data can be used to estimate failures. In this case, the defect arrival rate approximates the failure rate. A release criterion for an application can stipulate that the failure rate at release time be zero failures in some time duration, or zero failures while executing a test suite.

Note that the failure rate during testing do not represent the failure rate of the application during operation as testing is usually intense and often one time unit of testing may be similar to multiple time units of normal operation for generating failures. Hence failure rate tracking, by suitable modeling, can give a rough sense of reliability in terms of failures per day or per week, for more accurate reliability estimation, more robust models must be used. Software reliability modeling is a complex task, requiring rigorous models and sophisticated statistical analysis. Many models have been proposed for software reliability assessment, and a survey of many of the models is given in [14, 15].

Its worth mentioning that as failure of software also depends critically on the environment in which it is executing, failure rates experienced in testing will reflect the ultimate reliability experienced by the user after software release only if testing closely mimics the user behavior. This may not be the case, particularly with lower levels of testing. However, higher levels of testing strive to replicate real world conditions. And in this case, the reliability estimation can be applied with a higher confidence.

7.5.4 Defect Removal Efficiency

Another notable analysis is *defect removal efficiency*, which can only be assessed after software release. Its aim is to evaluate the effectiveness of the testing process, not the quality of testing for a project. This analysis strives to enhance the testing processes in the future.

Usually, after the application has been released to the client, the client discovers defects, which have to be fixed. This defect data is also generally logged for tracking purposes. Within a few months, most client-found defects surface (often the "warranty" period is 3 to 6 months).

Once the total number of defects (or a close approximation to the total) is known, the *defect removal efficiency* (DRE) of testing can be determined. The defect removal efficiency of a quality control activity is defined as the percentage reduction in the number of defects by executing that activity [16]. As an example, if 20 defects were found after delivery, and 200 during system testing, The DRE of system testing is 200/220 (just about 90%), as the total number of defects present in the system when testing started can be approximated as 220.

It should be clear that DRE is a broad concept which can be applied to any defect removal activity, such as design reviews, or unit testing. This assessment requires analyzing and logging the phase in which each defect was introduced, alongside its discovery. With this information, when all the defects are logged, the DRE of the main quality control tasks can be determined. Such insights are invaluable for enhancing the overall quality process.

7.6 Summary

- Testing is a dynamic method for verification and validation, where the software is executed with carefully designed test cases and the behavior of the software system is observed. A test case is a set of inputs and test conditions along with the expected outcome of testing. A test suite is a set of test cases that are generally executed together to test some specific behavior. During testing, only the failures of the system are observed, from which the presence of faults is deduced; separate activities have to be performed to identify the faults and remove them.
- The intent of testing is to increase confidence in the correctness of the software. For this, the set of test cases used for testing should be such that for any defect in the system, there is likely to be a test case that reveals it. To ensure this, it is important that the test cases are carefully designed with the intent of revealing defects.
- Due to the limitations of the verification methods for early phases, design and requirement faults also appear in the code. Testing is used to detect these errors, in addition to the errors introduced during the coding phase. Hence, different levels of testing are often used for detecting defects injected during different stages. The commonly employed testing levels are unit testing, integration testing, system testing, and acceptance testing.
- For testing a software application, overall testing should be planned, and for testing each unit identified in the plan, test cases should be carefully designed to reveal errors and specified in a document or a test script.
- There are two approaches for designing test cases: black-box and white-box. In black-box testing, the internal logic of the system under testing is not considered

and the test cases are decided from the specifications or the requirements. Equivalence class partitioning, boundary value analysis, and cause-effect graphing are examples of methods for selecting test cases for black-box testing. State-based testing is another approach which can also be viewed as gray-box testing in that it often requires more information than just the requirements.

- In white-box testing, the test cases are selected based on the internal logic of the program. Often a criterion is specified, but the procedure for selecting test cases to satisfy the criteria is left to the tester. The most common criteria are statement coverage and branch coverage.
- LLMs can help in generating scripts for unit testing, and test plans for system testing. The test cases generated by LLMs are not always correct and may have redundant cases.
- Executing test cases requires suitable tools. There are a host of open source tools available. Logging defects and tracking them to closure is a good practice that is widely followed.
- Key metrics during testing include coverage achieved during testing, open-defect density, and reliability of the software under testing. Assessing reliability requires models to be used, which often require considerable amount of data. Defect removal efficiency is another crucial metric, particularly useful for enhancing the testing process.

Self-assessment Exercises

1. Define fault, error, and failure.
2. What are the different levels of testing and the goals of the different levels?
3. Suppose a software has three inputs, each having a defined valid range. How many test cases will you need to test all the boundary values?
4. For boundary value analysis, if the strategy for generating test cases is to consider all possible combinations for the different values, what will be the set of test cases for a software that has three inputs X, Y, and Z?
5. Suppose a software has five different configuration variables that are set independently. If three of them are binary (have two possible values), and the rest have three values, how many test cases will be needed if pairwise testing method is used?
6. Consider a vending machine that takes quarters and when it has received two quarters, gives a can of soda. Develop a state model of this system, and then generate sets of test cases for the various criteria.
7. Consider a simple text formatter problem. Given a text consisting of words separated by blanks (BL) or newline (NL) characters, the text formatter has to covert it into lines, so that no line has more than MAXPOS characters, breaks between lines occur at BL or NL, and the maximum possible number of words are in each line. (A program for this is given in [17].) (i) For testing this program select a set of test cases using the black-box testing approach. (ii) Use as many techniques as possible and select test cases for special cases using the "error guessing" method.
8. Write the code to implement the textformatter (or have LLM generate it). Select a set of test cases that will provide 100% branch coverage.
9. An application is being built which will be of size 10 KLOC (suppose no framework/libraries etc are being used). Suppose the following is the defect injection rates during different phases:

Requirements—3 defects/KLOC; Design—2 defects/KLOC; Coding—20 defects/KLOC. Suppose the project only uses unit testing and then system testing. Suppose during unit testing a total of 150 defects are found, and during system testing 75 defects are found. Assume all defects are fixed. (i) What is the defect removal efficiency (DRE) of unit testing. (ii) What is the final quality of this application.

References

1. IEEE. IEEE standard glossary of software engineering terminology. Technical report, 1990
2. G. Myers, *The Art of Software Testing* (Wiley-Interscience, New York, 1979)
3. M. Phadke, Planning efficient software tests. Crosstalk (1997)
4. D. Cohen, S. Dalal, M. Fredman, G. Patton, The AETG system: an approach to testing based on combinatorial design. IEEE Trans. Softw. Engin. **23**(7), 437–443 (1997)
5. T. Chow, Testing software design modeled by finite state machines. IEEE Trans. Softw. Engin. SE-4 **3**, 178–187 (1978)
6. R. Binder, *Testing Object-Oriented Systems—Model, Patterns, and Tools* (Addison-Wesley, 1999)
7. J. Offutt, S. Liu, A. Abdurazik, P. Ammann, Generating test data from state-based specifications. J. Softw. Testing, Verificat. Reliab. **13**(1), 25–53 (2003)
8. S. Rapps, E.J. Weyuker, Selecting software test data using data flow information. IEEE Trans. Softw. Engin. **11**(4), 367–375 (1985)
9. S. Bhatia, T. Gandhi, D. Kumar, P. Jalote, Unit test generation using generative AI: a comparative performance analysis of autogeneration tools, in *Proceedings of the LLM4Code at ICSE 2024* (IEEE, 2024)
10. J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, Q. Wang, Software testing with large language models: survey, landscape, and vision. IEEE Trans. Softw. Engin. (2024)
11. N. Brown, Industrial-strength management strategies. IEEE Softw. (1996)
12. P. Jalote, *Software Project Management in Practice* (Addison-Wesley, 2002)
13. K.S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, 2nd edn. (Wiley-Interscience, 2002)
14. W. Farr, Software reliability modeling survey, in *Software Reliability Engineering*, ed. by M.R. Lyu (IEEE Computer Society, McGraw Hill, 1996), pp. 71–117
15. J.D. Musa, A. Iannino, K. Okumoto, *Software Reliability-Measurement, Prediction, Application* (McGraw Hill, 1987)
16. S. Kan, *Metrics and Models in Software Quality Engineering* (Addison-Wesley, 1995)
17. J. Goodenough, S.L. Gerhart, Towards a theory of test data selection. IEEE Trans. Softw. Engin. SE-1, 156–173 (1975)

8 Application Deployment

In the previous chapters, we have discussed various aspects of developing and testing a software application. During development, a software application is typically a set of source code files organized in directories and developed by different programmers in the development team. Generally, the main branch of the repository keeps the files that represent the accepted version of all the files and from which the final application is built.

During the project, programmers in the team make clones of the repository for the project on their local machines, and work on the files. They may modify existing code files, add new files, and periodically update the repository. To run the application for testing, they typically build the application on their machine and execute it on their machine. (Repositories are essentially just a centralized file system with some extra controls for changing files—in themselves, they cannot execute an application.) Therefore, during development, while the application code (and other assets for the application) may reside in a repository, the application is built and executed on the machines of team members. We will call this—the repository and the machines used by programmers for application execution—the development (or producer) environment.

Since the application has been developed for specific sponsors and users, it has to finally run on their hardware resources and environment, which we call the sponsor (or user/target/consumer) environment. So, when the development team is satisfied and ready to "release" the application, it has to be moved from the producer environment and deployed in the consumer environment such that the users can use it.

Given the complexity of applications today, application deployment can be fairly complicated. In this chapter, we will briefly discuss some simple approaches for deploying modest-sized applications. We will discuss the deployment of two types of applications: stand-alone installable applications that run on the user's machines and web applications, where the front end of the application runs within the browser, which is running on the user's machine, and the back end runs on some server in the sponsors' environment, and the database runs on a database server to which the back-end server can connect. We will not discuss advanced topics (e.g., continuous deployment, updating applications with patches, etc.).

In this chapter, we will discuss:

- The overall deployment process and the key activities in it—release, install, and activate.
- The packaging of an application for releasing it to the end users, and how it is done for installable applications developed in a compiled language or an interpreted language, and how it is done for web applications.
- How installation and activation are done for installable applications and web applications.
- How LLMs can be used to help in application deployment.

8.1 The Deployment Process

As mentioned, a software application will be a set of source code files organized in directories in a repository, in which the main branch will contain the code for the current version of the application. The deployment process is the set of activities to be performed so that the application becomes available for the users [1].

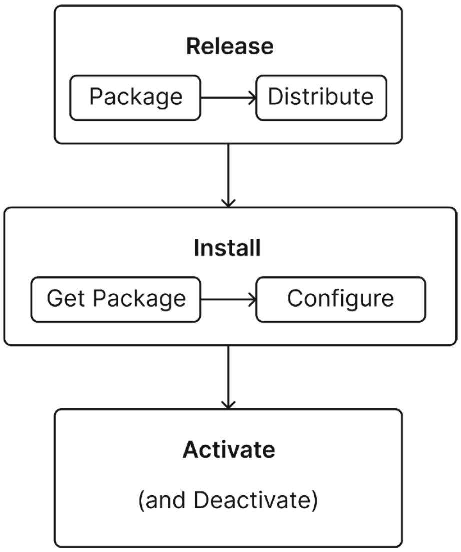
The deployment process comprises many activities, the main ones being release, install, and activate [1,2].

Release activity is what the application development team does to make the application ready for deployment, typically at the end of a development cycle when the team is satisfied that the application is ready for use by end users. An important task is *packaging* the application so it can be transferred to the sponsor's environment. The package must contain all the components of the application, specification of all the dependencies of these components (i.e., what other systems or components must be in the environment for the application to run), the deployment procedure, and other relevant information.

While release and packaging are done in the development environment, installing an application is done in the sponsor's environment. It requires receiving (i.e., transferring) the application from the developer environment into the sponsor environment, extracting the components from the package, and configuring the components/application for proper execution. (A special case of installing an application is updating the application in which the older version in the consumer environment is to be updated with a newer version of the application that has been released.)

Finally, after the application has been made ready by installing it in the consumer environment, it has to be executed (or activated). That is, the application starts executing in the consumer environment and becomes available for use. The overall deployment process is shown in Fig. 8.1.

Fig. 8.1 The deployment process



We will discuss these three major activities further in this chapter. Besides these major activities, there may be other associated activities like deactivating existing versions of the application, de-installing old components or applications etc.—we will not discuss these.

A key challenge in deploying an application is managing the dependencies between components. The components of an application often depend on some other components within the application or in the environment. If these dependencies are not available when the component is executed in the consumer environment, it may not function properly. Internal dependencies between components of the application are often simpler to handle—they may require the installation of components in some order. However, external dependencies on the environment require more work.

Let us understand these external dependencies. When a developer executes the application in the developer environment, the application has access to all the resources in the environment. For example, the developer may already have all the libraries, frameworks or tools that the application uses installed and are therefore available to the application and may have the resource files (e.g., images) needed for its execution. In such a situation, the application code will execute properly. However, even after all the application code files have been installed on the user side, if the same libraries or tools (with the same version) or the resource files that are needed for the application to run are not available in the user environment, this application which was working perfectly in the development environment will not work in the user environment. Hence, during deployment, dependencies have to be handled appropriately.

8.2 Release and Packaging

A software application release involves making the final version or the newest update/version of the software available to the users for installation and use. Release is the act of moving the application from development to its use by users. An application is deemed ready for release after its features/requirements have been implemented, all test scripts have successfully executed, and the end-to-end system testing plan has been satisfactorily completed. A release may be internal or to a limited group, such as for testing purposes, to the users or beta testers. We will assume that the release is to the users.

The release phase primarily prepares the application to be transferred from the producer environment to the consumer environment. Packaging is the key step in this process. During packaging, the resources necessary for the software application to operate correctly are gathered and typically packaged in one file that can be easily transferred to the target environment and installed there. Resources include the application code files as well as any other resource files (containing images, data, access codes, etc.) needed by the code during execution. The package typically also consists of a list of external dependencies on which the application code depends. Additionally, the package may also contain a description of the system requirements, setting of environment variables, etc. that are important for installing the application properly in the customer environment.

The application may be a stand-alone installable application, in which case the application package will be sent to the user, who will then install the application on a device and use it. For a web application, the package has to be sent to the server where the application is to be hosted and configured properly for use by users (through a browser). We will discuss packaging of these two types of applications separately.

8.2.1 Packaging Installable Applications

Packaging consists of combining all the code files for an application, along with the resource files that the code files use, into a single file. Packaging combines all the files containing code, data, images, scripts, etc., for the application into a "package", which is a single file. This package can then be transferred to end users through various distribution mechanisms like email, download, App Store, etc.

The package contains all the files and information necessary to install and run the application on the user's machine. The package can also serve as an installer for the application through suitable tools which will have a script to install different parts of the application and its dependencies in the desired order and configure them suitably.

Packaging approaches can be different for different programming languages and environments. Different tools are also used for this purpose. We will briefly discuss the packaging of application developed in a compiled language like C/C++ or an interpreted language like Python or Java.

Packaging an Application Developed in a Compiled Language

Compiled languages like C/C++ are commonly used to build applications, particularly installable applications. Applications developed in these languages can be directly compiled into an executable file, which the operating system can run (or execute) on the user's machine. Typically, separate source files in an application may be compiled to create separate object files. These object files are then linked to create a single executable file for the application. This executable file can be executed in the environments for which it has been compiled. The goal of packaging such applications is to combine the executable file of the application with all the external files and resources that the executable needs to run properly.

For packaging a C/C++ application, the source code files of the application have to be compiled. Often different source files may be compiled separately, producing multiple object files. To build the application, these object files must be suitably linked to produce a single executable file. I.e, the build process compiles the source files and links them to produce the executable for the application.

In a large project, often only a few source files need to be changed to fix a bug or add a feature—the rest may remain unchanged. In such cases, compiling all source files and linking them to build the executable is wasteful and time-consuming. To avoid this, special programs like makefile are used, which automate the process and orchestrate the compilation and linking of a C++ program. For an application, its makefile specifies what files the final executable depends on and how those files are generated (e.g., by compiling some source file). When a makefile is executed (using the make command), it will avoid compiling the source files that have not changed since the last compilation.

The source code for an application may require some external libraries to execute. Typically, the final executable produced for the application will also include these libraries.

Even with the external libraries included, the executable file for the application may require some external resource files (e.g., images, data files) from which the programs may get the data they need for execution. During packaging, these files will also have to be included in the package. Finally, there may be some setup script that places the files in suitable folders and sets the environment variables in the system (e.g., the path) for executing the application. Such a file must also be included in the package. We can assume that the packaging compresses all these files into one file, e.g., a zip file.

Note that the application will have some dependencies on the operating system and the hardware, and one executable cannot run on different hardware and operating systems. Due to this, different executable files must be produced for different operating systems and hardware and packaged suitably.

Packaging Applications in Interpreted Languages

Interpreted programming languages like Java and Python are hugely popular now, and most of the development of new applications is done in these languages. In such languages, the source code is converted by a compiler to an intermediate byte code, which is then executed by the language's interpreter. In other words, to execute such an application, the interpreter (or the run time system) for the language (which comes as an executable) has to be executed, and this interpreter then "executes" the application code.

An application developed in an interpreted language like Python can be distributed to end users in multiple ways. The main ones being:

- Distribute the application as source code. In this method, the developers do not mind sharing the source code, and the end user has sufficient knowledge to install and run the source code using the compiler and the runtime system to get the application working.
- Distribute the application as bytecode. In this mode of distribution, the application is packaged into the intermediate bytecode, and then this package is sent to the user. The user still needs the runtime system to run this application but may not need the compiler.
- Distribute the application as an executable. In this mode of distribution, the application is distributed as an executable, which can be directly executed by the operating system on the user's machine. To achieve this, the runtime systems needed for executing the application must also be included in the package, along with the application code.

To distribute the application as source code, the source code of the application and all necessary resource files need to be packaged (excluding files that are not needed for executing the application, like test case files). One simple way is to zip the application and send the zip file—the file can be unzipped, if desired, or directly executed (e.g., Python can execute a zip file if it has a entry file "main.py" in it.) In this style of distribution, the target environment will need to have all the external dependencies installed (besides having the language compiler and runtime). Python also provides the "zipapp" utility, which packages the application for distribution, including all the dependencies specified in the 'requirements.txt' file. This file can be executed by Python interpreter on the target machine.

For packaging applications in bytecode format, the application must be compiled to create the bytecode for it. The bytecode can then be distributed, along with all the resources that it may need. To execute the application, the target machine will need the runtime system (or interpreter) for the language, though the compiler may not be needed. For example, Java provides a command to create a JAR file for the application from its source files, bundling all the bytecode files into one JAR file. This JAR file can then be executed on the target machine using the Java runtime system. Regular methods of JAR file creation often just package the application files but not the dependencies (i.e., the packages that the application code may be using). Such a

JAR file can execute only if the dependencies are also available in the target system. Tools like Maven can create a JAR file that will also include the dependencies.

For packaging applications to be distributed as executable files, which can be executed in the consumer environment without having Python/Java installed, the executable file must include the application code, all the necessary resource files as well as the interpreter (or run time system) to execute the application code. For Python, PyInstaller is a popular tool used for packaging Python applications into standalone executables for distribution across different platforms.

Pyinstaller requires a file in the main folder of the application that provides the entry point of the application (i.e., main()). It analyzes the application, resolves dependencies, and creates an archive file containing the application code, any libraries it imports, and all the resources they require (such as data files, images, or configuration files). It also includes the complete Python system (including the installation utility, pip) as executable in the archive—this makes the package an executable (we will see below how this executes in installation). It should be pointed out that Pyinstaller can create executable only for the platform on which it is running. This means that when working on an Apple machine, one cannot create an executable for a Windows machine–to create an executable for a Windows machine, the Pyistaller packaging has to be done on a Windows machine.

8.2.2 Packaging Web Applications

A web application, as we have seen, typically has three independent components, each of which has its own code and runs on a separate hardware. The *front-end code*, typically HTML with JavaScript embedded in it, runs within the user's browser. The *back-end code*, which services the requests from the users via the front-end code, typically runs on a server connected to the web. Also, there is a *database server* that has the database and executes the requested queries from the back-end code (often, the interface modules for the database may be organized as a separate layer in the back-end code).

During development, all the code is kept in a repository. When working with a clone of the repository on a developer's machine, the full application is generally executed on the developer's machine. Most application development frameworks provide support for running the back-end server on the local machine, typically using local hosting support so the programmer's machine does not need to be connected to internet when running an application locally. Often, support for the database is also provided locally (or the database server may be hosted on the local machine). So, the programmer can run the front end of the application on the browser on the machine while also running the back end and database on the same machine. This allows the application to be tested in a developer environment.

Once the application is ready for release, it has to be packaged for hosting on the sponsor's sever (which is on the web and has an IP address as well as a domain name) so the users can access it on the web through the pages provided by the application.

The back-end code can run on the server assigned for it and can connect to the database server for the application.

Let us first briefly discuss how packaging is done for a traditional web application where the front end is in HTML with embedded Javascript, and the back-end code is in some language like PHP. Typically, the repository will have a folder for the application that will contain sub-folders to organize the code files (as discussed in an earlier chapter) and the resource files (e.g., images under the images folder). Generally, there is a fixed entry point for an application in its folder (e.g., index.php file for PHP)—when the application is accessed by a user from the browser, the web server (configured with the domain name for the application) will pass the request to the php server for this application, which will then execute the index.php file. The external dependencies for the application are often specified in a file as well. To make the application available to the users, this folder (and any dependencies the programs in this folder that have been specified in a file like requirements.txt) has to be available on the sponsor's server.

For packaging and transferring, the simplest way is often to archive the folder for the application into one file (e.g., by using the zip utility) and send it to the server. Alternatively, the application folder can be cloned on the server. In both cases, the folder structure for the application is replicated on the sponsor's server. We will explain how these are installed in the next section.

Web applications based on frameworks like NextJS or Django specify a certain folder structure for the application and also how to specify external dependencies in a specified file in the application folder. For packaging and transferring the package, the approaches mentioned above—create an archive and send it or clone the folder—can be used.

8.3 Installation and Activation

The released package is now ready for insertion into the consumer environment. The package first enters the target system in the installation phase. The application is then re-assembled back from the package. Hence, this phase consists of two sub-phases. The first sub-phase involves the insertion or transfer of the package from the source to the destination. The second sub-phase consists of configuring the application to make it ready to operate in the consumer environment. Finally, after installation is complete, the application can be activated for use by the users—this step is often quite simple.

For ease of installation, the complete released package is pushed onto a remote place like a server, an app store, or a repository to make it available to the users. The "package" first enters the target system in the installation phase by users downloading and saving the package on the local machine or server. The application files are then re-assembled from the compressed package in the form they were in at the source end before packaging—this may be simply unzipping the files in a folder. (If the files are cloned from the repository, then this step is unnecessary).

To get the application ready for running, besides having the code and resource files organized suitably in proper folders, there may be a need to set up the environment. Often, a setup script file is provided in the package for this. In this situation, the setup script has to be run first. Typically, these setup scripts may unzip the application package to place the executable and resource files in proper folders, install all external dependencies, and set up all the environment variables. Once this is done, the application is ready to be run or activated in the user environment. If a setup script is not provided, the instructions for installation are typically provided. In that case, these steps will have to be executed manually to install the application.

Activation of an application refers to starting the application after it has been installed. While it is a distinct phase in application deployment, we will consider it along with installation. Often, activation is essentially starting the execution of the application. It may also involve other things like the deactivation of older versions of the application, etc.—we will not consider these activities. Let us consider the installation and activation of installable applications and web applications separately.

8.3.1 Installable Applications

As discussed in packaging, installable applications can come as executables, packaged source files, packaged bytecode files, along with the resource files the code files need. These are packages that can be installed on the target machine and then used. For installation, it has to be first received—this is typically facilitated by the distribution setup of the application. It can be obtained by downloading it from a site, through an app store, or even receiving it by email.

Once the package is received on the target machine, the application needs to be installed in a folder (and then executed or activated). Installation methods may differ depending on the type of application.

Installing an application received in source code format may require unzipping the compressed file in a folder, which will, besides setting the source files, also set up the resource files needed by the source files. It may also provide instructions on executing the application (using the compiler and the runtime for that language). In some cases, the compressed file itself can be executed directly (e.g., in Python if it contains an entry point in the main folder).

Installing an application in JAR format just needs the JAR file of the application to be extracted and saved, and all the necessary resource files to be saved in suitable folders. Activation will require executing it using the JAR execution command.

If the application is an executable built from C/C++ code, then also installation requires unzipping the package, saving the executable in some location, saving the resource files in suitable folders, and setting the environment variables suitably (maybe through the setup script that may have been provided). Once this is done, the executable file can be directly executed (e.g., by double clicking it or giving a command line instruction). Typically, this activation will create an operating system process to execute the application.

Executing the executable created by tools like Pyinstaller for interpreted languages is different. Here, too, the operating system starts a process that will first unzip the compressed file into temporary directories. It will then start executing the runtime system for Python (which comes in executable format), which then executes the application code. When the application code execution completes, the main process that started the runtime removes the temporary folders created (where code files for execution were kept), leaving behind only the original executable package for the application.

Most applications have installers to help simplify the installation process. The installation process typically automates the extraction of code and resource files, setting up the environment, configuring the application, etc. Typically, the first step of an installation is performing compatibility testing, which involves verifying whether the user's device can run the application. The next step is extracting or unzipping the package for the application and extracting the code and resource files. The installer may also install dependencies that this application needs. Finally, the installer sets up the environment variables and other settings. The installation process ends with a notification to the user that the application has been installed successfully (or reports any errors that may occur during installation) and how the application can be activated/used.

8.3.2 Installing Web Applications

Let us first discuss installing applications where the front end is in HTML+JavaScript and back end is written in a language like PHP. The application has to be installed on a server on the web with some web server running—we can assume that it is Apache (which is one of the most widely used web server).

While the back-end code for such applications can be executed within a web server like Apache, we will assume that there is a separate application server for serving the application requests (e.g, PHP-FPM for executing PHP applications)—this makes the application more efficient and frees the web server for receiving new requests while an older request is being serviced by the application server process. Typically, the web server will pass the request for executing code for an API in an application to the application server process, which may create a separate thread or process to execute the code for the requested API. When the execution is done, the application server will send its response to the web server, which will return it to the user. The application server response can be an HTML page or a JSON object.

Also, most applications would like to have a distinct URL for their landing page—it can be either as a subdomain, i.e., myapp.dname, or it can be of the form dname/myapp. Installation in these two options is almost similar, with minor differences in how the web server and network are configured to handle incoming requests for the application.

For hosting such an application, a clean approach is to have a virtual host with sufficient resources allocated to it. So, first, a virtual host has to be set up with an operating system and some remote access mechanism (e.g., ssh) so the application team can access the host. Suitable entries are made for network configuration (e.g. DNS), so that requests for this application are directly routed to the virtual host. If it is a subdomain, a static IP address may be allocated to the virtual host. Setting the virtual host is typically done by the IT system/network administrator, who hands over the virtual host to the application team, which can install software and files on it.

It is on this virtual host that the application needs to be installed. As mentioned above, this can be done by getting all the files as an archive file and extracting them in a folder on the host for the application—for this option, the software to unzip the package is needed to be installed on the host. Alternatively, the application folder can be cloned from the repository on this virtual host—for this option, suitable software (e.g., Git) is needed to be installed on the host. After getting the application files, all the external dependencies have to be installed, which may require some installation software (e.g., pip) to be available on the host. At the end of this step, all the files for the application will be in the folder for the application including the external dependencies.

The application team has to now install the application. The main steps involved in installing this application are:

- Install all the software needed to run this web application. This means installing the web server (e.g., Apache), the application server (e.g,. PHP-FPM), and the database (e.g., MongoDB).
- Suitably configure the web server to work with the application server. This may require enabling suitable modules in the web server (e.g., proxy-related modules in Apache)
- Install all the extensions needed for accessing the database through PHP.
- Update the PHP code to connect to the database on this host, as the code from the development environment was likely configured to access the database through localhost.
- Configure the web server for this host by specifying the application directory for the application, and granting permissions to the directory to the web server.
- The web server may need to be reloaded.

If the request coming from the browser running the front-end code is for a static file (i.e., a regular HTML/PHP file), the web server just goes to the application folder, fetches the file and sends it. If the request is for executing a PHP script, then it will pass it on to the application server, which will get the PHP script from the application folder, execute the script (which may interact with the database), and then return the result to the web server.

In this approach, the back end provides the APIs that the front-end code can call. Traditionally, when this is done, the back-end code returns a full HTML page for rendering. For creating the HTML page, templates will often be used, which will be suitably populated with the data computed by the API code (which may have been obtained from the database) and then sent.

Modern approaches for executing this have become more refined. The front-end code calls a back-end API, but the back-end API only returns the results of the computation as JSON objects. Using these objects and the current page being rendered for the user, the front-end code just updates part of the currently rendered page with new data from JSON objects. This makes user experience smoother as the entire web page does not have to be reloaded—only those components of the page that need to change due to the results from the API need to be changed.

The application code in PHP will need to be configured suitably to interact with the database on the host, so the statements for connecting the database in the application code will have to be changed (e.g., during development, if "localhost" was being used, it will have to be suitably replaced).

To make the application live (activating it), the web server is informed about the application so it can route all requests for the application to the PHP server. And the PHP server is started for the application.

8.4 Using LLMs for Deployment

Packaging and installation often become quite complex, requiring a series of steps to be taken. If an application needs to be hosted, additional steps need to be taken to get a virtual host either from the local IT administrator or from a cloud provider. This requires a sequence of steps to be performed to prepare the host for hosting the application.

LLMs can be quite useful in guiding this process. For example, for a web application whose source code files are all on GitHub and which is to be hosted on a virtual host provided by an organization, prompts can be given to ask for the steps to be followed for each step.

For example, for a web application that is to be hosted on a virtual host, multiple steps need to be taken to make the host ready, migrate the files, and then install the application. The following type of prompt works well:

> Prompt: We have a web application developed using HTML + CSS + JavaScript for the front end, Node.js for the back end, and MongoDB for the database. All the source files are on GitHub, with files logically organized in different folders for front end, back end, models, etc. To host this application, we have been provided with a virtual host. Please provide step-by-step instructions on what I should do for hosting this application.

A prompt like this gets a reply that lists the various steps the team needs to perform on the virtual host, on GitHub, for activating the application, etc. The steps are quite detailed, and if needed, further clarification or details can be sought for any of the steps. For example, the prompt above gave many steps (with details), including some prerequisites:

Prerequisites

- Access to the Virtual Host: Ensure you have SSH access to the virtual host.
- GitHub Repository: Ensure your code is organized and pushed to a GitHub repository.
- Domain Name (Optional): If you want your application to be accessible via a custom domain, you should have a domain name registered.

Steps to Host the Application

- Connect to Your Virtual Host
- Install Required Software (Node.js, npm, MongoDB)
- Clone Your GitHub Repository
- Install Project Dependencies
- Configure Environment Variables
- Set Up and Start MongoDB
- Set Up Reverse Proxy with Nginx—install, configure, restart
- (Optional) Set Up a Domain Name
- Start Your Node.js Application
- Test Your Application.

Instructions and often actual commands were given for each of the steps. For any step that is unclear, the LLM can be asked to provide an explanation and more detailed instructions for that step (and it does provide them).

The LLM can also provide detailed steps for hosting on a cloud like AWS—its instructions may include detailed instructions for opening an account in AWS and setting it up for hosting.

Overall, LLMs can be very useful for guiding the application hosting process, particularly for non-experts who will otherwise have to go through various guidelines to understand the steps that need to be taken.

8.5 Summary

- The deployment process is comprised of multiple activities, mainly release (including packaging), installation, and activation. A key issue in this process is how to properly handle dependencies in the application code, without which an application that may have been functioning satisfactorily in the development environment may not execute in the consumer environment.
- Packaging an application requires combining all source files, files for other resources the source needs (e.g., images, other files), and the scripts that set up the environment variables for the application to run. What needs to be packaged and how it differs for stand-alone installable applications and web applications.

  - Packaging an installable application in a compiled language requires linking the object files for different source files into one executable file and then packaging it along with other resource files needed into a package.
  - Installable applications written in interpreted languages can be sent as source code files, requiring extraction from the package in the user environment, and then compiled and executed using the compiler and runtime for the language. Such applications can be packaged in the byte code form, in which case they have to be executed using the runtime of the language. They can also be packaged as an executable, wherein, the runtime system for the language is also packaged, along with the bytecode. The runtime is executed in the user machine, and it then executes the application code.
  - Packaging web applications requires combining the source files for the front end and the source files for the back end, along with the specification of the external dependencies and other resource files. These files may be zipped into one for sending to the server where the application is to be hosted. Alternatively, they can be cloned on the server.

- Installation of an application requires extracting files from the package and organizing them properly in the user environment for execution, as well as setting the configuration suitably for activation or execution of the application. Installation methods also differ for installable applications and web applications.

  - For installable applications, the files are extracted and organized in folders as needed. The setup script is executed (or the steps are done manually) to set up the environment variables suitably. For executing the application, if it is an executable, it can be just run. If the application was sent as byte code, then the runtime system for the language needs to be run and made to execute the code for the application.
  - For installing web applications, a hosting server has to be chosen. On this server, besides the web server, suitable application servers will be started, depending on what framework is being used. The application will have to be made visible to the internet by suitably configuring the server—this has to be done by the network administrators of the organization where the server is hosted.

- LLMs are an effective tool for identifying the sequence of steps to be performed for deploying an application. They can provide detailed instructions also.

Self-assessment Exercises

1. Explain the basic problem of application deployment.
2. Discuss the role of dependencies in the deployment process. Why is it crucial to handle dependencies correctly, and what can go wrong if they are not managed properly?

3. You have developed an installable application written in a compiled language like C/C++, and the source files are all on a repository like GitHub along with test files and all other files. List the steps that you need to perform to send the application to a user through email.
4. If an installable application written in an interpreted language has to be sent as an executable to the users, what all must it include in the package? How may such a package be executed on the user's machine?
5. For a web application besides the code for the frontend and for the backend, what other files need to be included in the package to be sent for installation?
6. You have developed a web application that allows anyone to register and perform some operations. Suppose the repository for this is on GitHub, properly organized. List the steps you have to take to host this application so it can be used by any user.
7. Describe the series of events that occur when a user requests an operation from a web application through a front-end screen provided by the application.
8. Prompt the LLM for different types of deployment methods and ask for the series of steps. Ask for further details of the steps till you are completely understand the process.

References

1. A. Carzaniga, A. Fuggetta, R.S. Hall, D. Heimbigner, A. Van Der Hoek, A.L. Wolf, A characterization framework for software deployment technologies. Technical report, Technical Report CU-CS-857-98, Department of Computer Science, University of Colorado, 1998
2. A. Dearle, Software deployment, past, present and future, in *Future of Software Engineering (FOSE'07)* (IEEE, 2007), pp. 269–284

Index