TURUN
YLIOPISTO
UNIVERSITY
OF TURKU

Basic course on software engineering

TURUN
YLIOPISTO
UNIVERSITY
OF TURKU

Basic course on Software Engineering

.

University of Turku
Department of Computing
Software engineering
2024
Panu Puhtila, Hanna-Kaisa Terho, Oshani Weerakoon, Olli Heimo, Tapani Joelsson, Jari Lehto, Tuomas Mäkilä

Sisällys

Kuvat

Taulukot

1 Introduction

According to the Institute of Electrical and Electronics Engineers (IEEE), software engineering is defined as:

> *The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.*[1]

There are numerous reasons to write computer programs. Some people do them as hobby projects or to improve their quality of life, others to solve problems in mathematics or astronomy, for example. Creating programming projects has classically been a good way to learn and study how computers work.¨

Most major software projects are done for business reasons and therefore the principles of software production follow the laws of business. As a result, a professional software developer must be able to consider a wide range of aspects of their work, from project management to budgeting and from understanding stakeholders to life-cycle management. Many software products are also subject to laws and regulations, the basic principles of which, or at least their existence, should be known to the software developer.

Software today is pervasive - it's everywhere - and digitalisation has moved from digitising and automating analogue systems to upgrading, improving and automating digital systems. In addition to automation, software is widely used to streamline processes, create new services and ways of working, meet business needs, improve competitiveness and, of course, generate savings. What they all have in common is the enormous power of software development to change the world around us.

This textbook provides one cross-section of software development. We will explore software

[1] IEEE 610.12-1990

development through agile methods, looking at the different stages and methods as an individual software developer sees them during the course of a software project. Due to the iterative nature of software production, many parts of the material overlap as the same techniques are used at different stages of software development. Consequently, different chapters of the book discuss the same techniques from different perspectives to achieve a broader view. The material has been sourced from a wide range of sources, including academic articles, videos on the subject, and blogs from software companies and individual software developers, so that the reader can learn more about the aspects of software development that interest them.

1.1 Problem definition

From a problem definition point of view, software production is a process involving many interlinked factors that are difficult to manage and solve. Software production can therefore be described as a "wicked problem". A problem is such a problem when the requirements for its solution are not known until the work is completed or the requirements change during the work progression [1]. Similarly, 'tamed problems' are problems for which a clear definition can be found. However, this does not mean that the problem is small or easy to solve, it just means that it is well defined [2]. Those interested in problem definition should consult John Dooley's blog post for a concise summary of the topic [3].

Developing a piece of software is a project for which there are an infinite number of different approaches. Different types of software require different approaches in terms of methods, techniques and working methods: for example, developing a global information system is very different from developing a medical device. The simplest method of software development is code-and-hack [2], where the software is modified until the customer (usually the programmer her/himself) is satisfied. However, the development of a software product typically requires a large and diverse group of participants. This team may consist of a single freelancer, a small start-up team or a multinational organisation employing thousands of developers.

While the distributed digital environment brings its own challenges in terms of communication, coordination and project management, technological advances such as cloud computing,

[2] The Code and Fix Model

collaboration tools and distributed version control tools have made it easier to overcome these challenges. Decentralised working can also open up new opportunities, such as the recruitment of talent across country borders, the flexibility of work and the possibility of exploiting the benefits of different time zones in terms of continuity of work. Software production is creative endeavour (cf. Knuth's *The Art of Computer Programming* -the name of his book series[3], art, not science)

There is no one right solution to most problems in software production. Project management based on linear, predictable progress is ill-suited to this problem domain. Modern software projects, however, make extensive use of flexible and iterative approaches that allow for adaptation and learning as the problem becomes more precise. Figure 1.1 on the next page illustrates the difference between a pre-designed software production process and an actual implementation. In practice, software production is therefore more like a private jet that is actively steered towards the destination that best suits the customer, rather than a train that moves from point A to point B on its tracks.

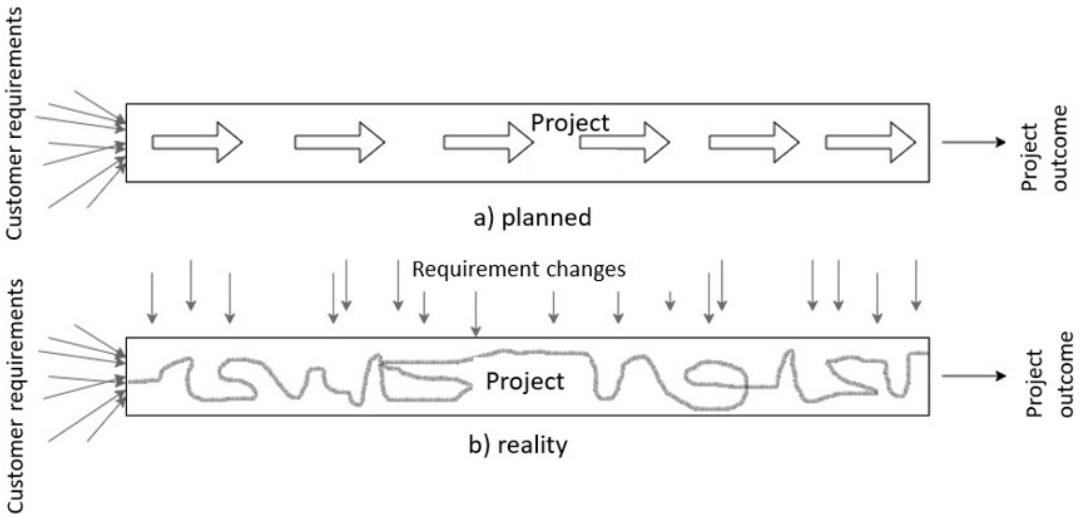

Figure 1.1: Software project as a process [4]

[3]The Art of Computer Programming: `https://en.wikipedia.org/wiki/The_Art_of_Computer_Programming`

1.2 Software and software product

A software product can refer to many types of software, ranging from simple utility scripts to complex systems consisting of multiple software components, devices and services. What distinguishes it from pure computer software is its business dimensions. A software product is **a product** whose objective, from the developer's point of view, is to generate revenue to cover at least the direct development costs of making the product, and from the business customer's point of view, is to enhance the efficiency of a business function or a process or reaching a specific business target. Software products aimed at consumer customers add value to their daily lives or hobbies, but can also be purely recreational.

Software products vary in complexity and scope. The amount of code in a software product can range from a single line to tens of millions of lines, and the code can be spread over tens of thousands of separate files [5]. There can be anywhere from one to thousands of authors, scattered in numerous teams, working in different offices, in different countries, on different continents and in different time zones. An example of such a decentralised multi-year and multi-faceted project is the GTA V game project, which involved about a thousand developers over several years [4]. This reflects the complexity and scale of software product development and the amount of work required to develop and maintain it.

A software product is not just code. The code is at the heart of it, but it is not the only part that determines the quality and functionality of the product. In addition to the code, a software product often involves a wide range of other materials and components that complement the functionality and usability of the product. For example, user manuals and training materials are an essential part of user support and enable the software to be easy to use.

A software product is not just code. The code is at the heart of it, but it is not the only part that determines the quality and functionality of the product. In addition to the code, a software product often involves a wide range of other materials and components that complement the functionality and usability of the product. For example, user manuals and training materials are an essential part of user support and enable the software to be easy to use.

[4]Wikipedia-article about developing GTA V: `https://en.wikipedia.org/wiki/Development_of_Grand_Theft_Auto_V`

Many software products also make use of graphics, sounds and videos, which, if implemented in a comprehensive way, can enhance the user experience and make the software more attractive. Audio-visual material can be used to illustrate even complex concepts, which is particularly important in systems where the user has to interact with many different elements. In some cases, the software product may also include hardware components. For example, in embedded systems and IoT applications, the combination of software and hardware forms a product package.

However, it should be remembered that a software product is a product, i.e. a good or service, which aims to create economic value for both the customer and the developer. While it is clear, that the starting point of a software project is to create added value for the customer, problems arise if the customer cannot clearly describe his needs and the software developer fails to understand what the project is trying to achieve. The software developer must also take into account his own business, i.e. that the project is also profitable for the developer.

Although there are systematic and efficient approaches to software production, according to a Tivi article [5], only one third of IT projects can be considered successful when the criteria for success are on time, on budget and on target. Moreover, it is estimated that up to 10-15 percent of IT projects fail completely. A Tivia report [6] on information systems procurement suggests that the most typical reasons for failure in information systems projects are lack of communication and interaction, insufficient resources and ineffective project management. The report also states that the client side may have a lack of understanding of the methods and practices involved in the software industry, which can lead to communication problems and misunderstandings. Common pitfalls can be avoided following software engineering practices, and in particular, by investing in communication. As mentioned above, a software product is more than just software code - it is a complete solution that meets the needs and expectations of the customer and provides economic value to both the producer and the customer. The development of a software product is first and foremost a joint effort of active mutual communication. Instead of being on opposite sides, the software developer and the customer should be seen as partners in the software project working together for common success.

[5] Miksi suurin osa IT-hankkeista epäonnistuu

1.3 Software Life Cycle

Software development can be viewed through a life-cycle approach. The purpose of a lifecycle model is to describe the development phases that are followed in a software development project. What is essential about the life cycle model is its reproducibility; it provides a defined framework that can be used as the basis for a wide range of software development projects. There are several different life cycle models, a classic example being the waterfall model described in the following subsection. Of the lifecycle phases, the implementation or programming stage is usually the most familiar and concrete for recreational and early career developers. However, the actual software project involves several different phases, not all of which involve writing single line of software code.

1.3.1 Waterfall model

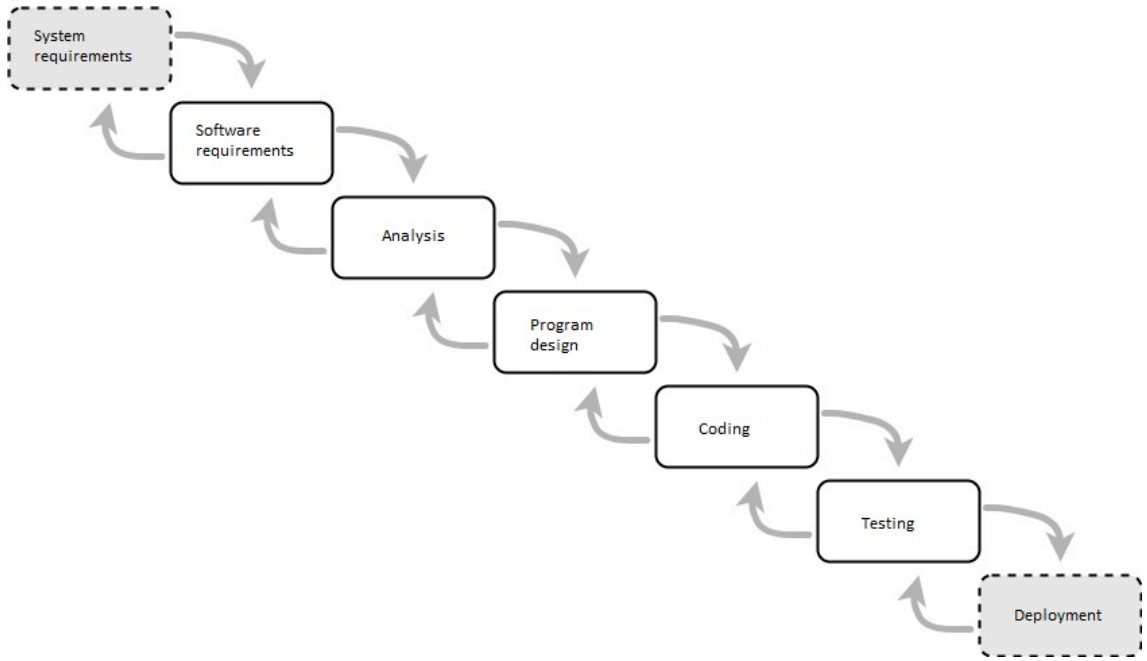

Figure 1.2: Waterfall model

The waterfall model6 has been known as a concept since at least the 1950s. The classic article on the subject is W. Royce's Managing the Development of Large Software Systems [7] from 1970. The model (Figure 1.2) presents software design as a step-by-step process from defining the system and software requirements to analysis and software design. This is followed by programming and testing, and the final stage is deployment. Royce's model has

been interpreted in such a way that the steps take place sequentially without returning to the previous step. This is incorrect, as Royce suggests that the same cycle can be run more than once, that iterations between stages can be exploited and that the customer can be involved at different stages of the process.[4] The model does not address the post-implementation and pre-definition stages of the software. For these, the complementary steps are as follows:

Vesiputousmalli[6] on tunnettu käsitteenä ainakin 1950-luvulta saakka. Klassinen artikkeli aiheesta on W. Roycen *Managing the Development of Large Software Systems* [7] vuodelta 1970. Malli (kuva 1.2) esittää ohjelmiston suunnittelun vaiheistettuna prosessina, joka etenee järjestelmän ja ohjelmiston vaatimuksien määrittelystä analyysiin ja ohjelmiston suunnitteluun. Näiden jälkeen suoritetaan ohjelmointi ja testaus, sekä viimeisenä vaiheena käyttöönotto. Roycen mallia on tulkittu niin, että vaiheet tapahtuvat peräkkäin järjestyksessä ilman että edeltävään vaiheeseen enää palataan. Tämä ajattelutapa on virheellinen, sillä Roycen näkemyksen mukaan samaa sykliä voidaan suorittaa useamman kerran, vaiheiden välisiä iteraatioita on mahdollista hyödyntää ja asiakasta voidaan myös osallistaa prosessin eri vaiheissa. [4] Malli ei ota kantaa ohjelmiston käyttöönoton jälkeisiin vaiheisiin eikä vaatimusmäärittelyä edeltäviin vaiheisiin. Niiden osalta täydentäen vaiheet näyttävät seuraavalta:

- *Pre-definition*
- *Procurement*
- Specification
- Analysis
- Design
- Development
- Validation
- Deployment
- *Maintenance*

[6] Waterfall model

- *Decommissioning/Replacement with a new system*

Brief history of software engineering: History of software engineering

1.3.2 RADIT

RADIT is a software development process model with the steps of Requirements, Analysis, Design, Implementation and Testing [8]. RADIT activities are found in Royce's waterfall model and are highlighted in white in Figure 1.2) of the waterfall model. In the figure, software requirements correspond to the Requirements phase of the RADIT model and programming to the Implementation phase. The RADIT model functions are found in almost every software development project model in some form.

The aim of the requirements definition phase is to understand the customer's needs and expectations for the software. This includes both functional requirements, i.e. what the software does (e.g. a user must be able to send messages to other users through the system) and non-functional requirements, i.e. how the software should work (e.g. the system must allow 10,000 users to work simultaneously). The analysis phase does not only focus on requirements analysis; it also involves high-level technical analysis to formulate the architecture of the system. The lower level software design is done in the design phase. In this phase the software components, the data model of the components, the method boundary interfaces of the components and the relationships between the components, following the higher level architecture are usually defined. The design phase results in a software plan that guides the implementation phase. In the implementation phase, the software code is written and executed using the selected programming languages and tools. This phase includes the actual programming and, in many cases, the testing of method-level functionality, or unit testing. This phase is also often referred to as the integration phase, because its main challenge is the interoperability of the code and software modules produced by different developers. The final phase of the RADIT cycle is testing. The testing phase ensures that the software as a whole works as intended and meets the customer's requirements. [8]

Among the test-specific process models, it is worth mentioning the V-model (figure 1.3), which describes the relationship between software lifecycle design and testing. The V-model is discussed in more detail in the information box in subsection **??** on testing.

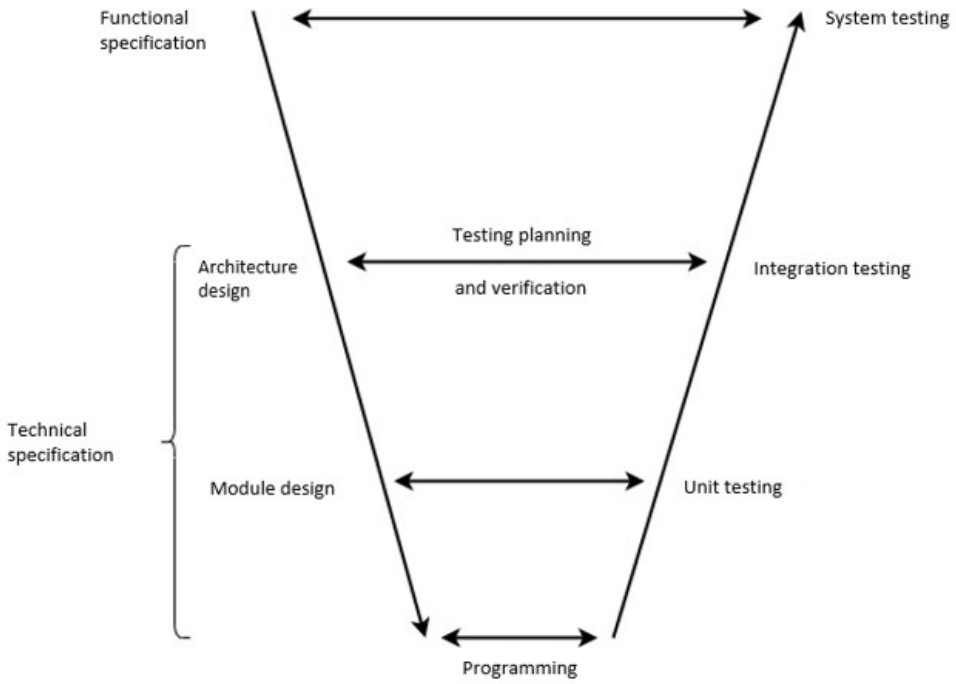

Figure 1.3: V-model

1.3.3 Agile methods

Agile software development has its roots in the 1990s, when the software industry started looking for alternatives to traditional project models. In 2001, the idea of agile was concretised when 17 software development experts met in Utah and created the Manifesto for Agile Software Development[7].

The manifesto defined four core values and twelve principles that should guide the chain of software development. They emphasise, among other things, the importance of individuals and interaction over processes and tools, the priority of working software over comprehensive documentation, cooperation with customers instead of contract negotiations, and a willingness to respond to changes instead of following plans. The Agile Manifesto has served as the basis for many modern agile methods such as Scrum, Kanban and Extreme Programming (XP).

Agile practices are working methods that support and enable developers to take independent responsibility for their own work and planning. The principles of agile working are that teams

[7]Manifesto for Agile Software Development

work in a self-organised way, agreeing on work tasks among themselves. This is how the team develops common working methods and practices. In addition to the technical work, the team suggests the timeframe in which tasks can be completed. The practical work is based on active communication between individuals and teams, supported by the tools and methods used.

1.4 Software project

A software project is the process of producing new or updating existing software. As discussed in previous chapters, the ultimate purpose of a software project is to understand the customer's business needs and translate them into functional requirements (FR) and non-functional requirements (NFR), taking into account the constraints and boundary conditions of the project (Figure 1.4). Only after these are established, can the project move on to functional specification. Modern software projects rarely start from scratch, as new software is often built on the basis of existing platforms, application frameworks or existing software versions. They may also be based on similar systems designed for different environments or uses [4, s. 26]. Sometimes the old code base may determine the tools available to the project, and the customer may also have their own guidelines.

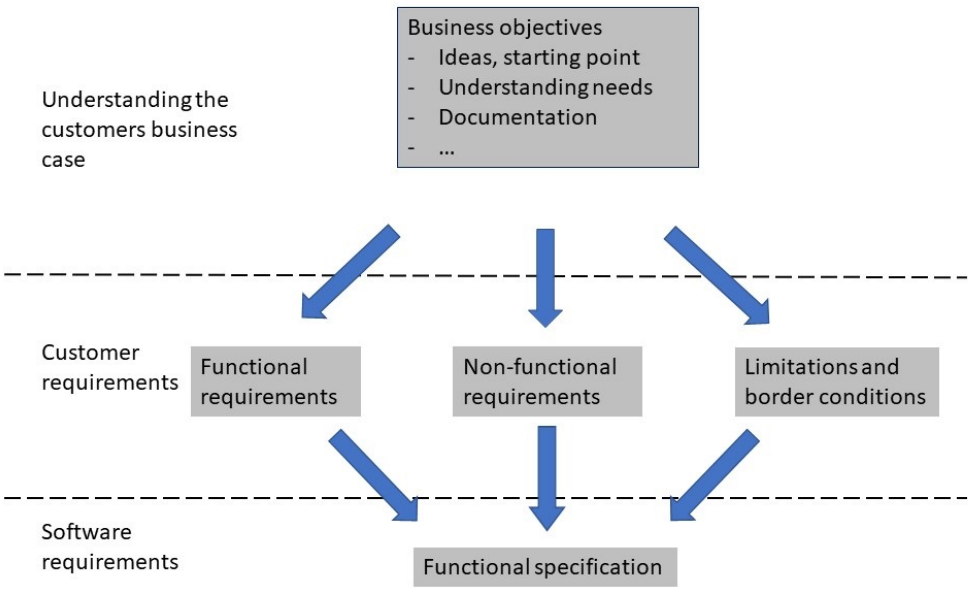

Figure 1.4: Customer and software requirements [4]

The Iron Triangle (also known as the triple constraint) is a way of describing the constraints that affect a software project (or any project), i.e. time, cost and scope (figure 1.5 on page 11). Quality is depicted in the centre of the triangle and is the only invariant in the triangle model, i.e. quality must be maintained even if changes occur in other aspects of the project. The basic idea of the project triangle is that the any element of the triangle is dependent on changes in the other elements. For example, if the scope of the software is to be enlarged, the effect is reflected in an increase in the budget and/or a reduction in the available time. By the same logic, speeding up a project requires either increasing the budget or reducing the scope of the project. If there is no flexibility in the budget and schedule, it may be necessary to reduce the functionalities of the software.

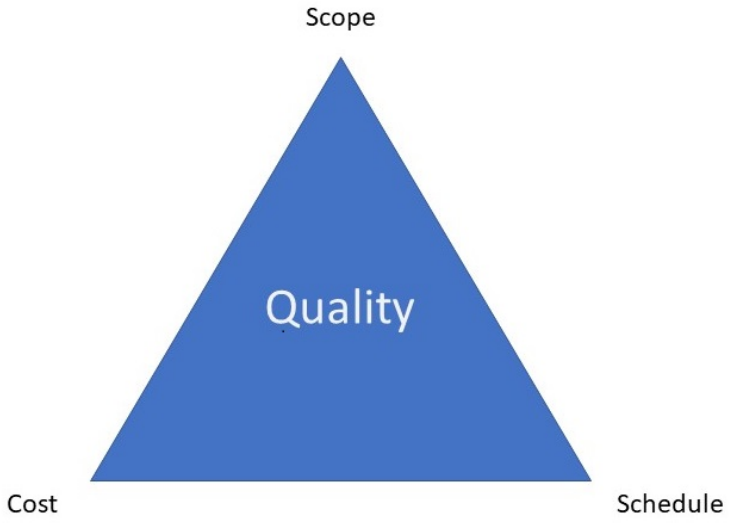

Figure 1.5: Iron Triangle

1.5 Types of software

Depending on the software and the needs, there is a wide variety of different types of target customers for a software product. The types of software can be roughly divided into off-the-shelf software, (commercial off-the-shelf software, COTS software) and custom software/tailored software. The challenge from the developer perspective is to find a balance between product-orientation and customer centeredness [4, s. 23]. That is, the more precisely the customer's

requirements can be taken into account, the more satisfied the customer usually is with the end result. Furthermore, the more customers' needs can be satisfied by off-the-shelf software, the fewer resources the developer has to dedicate in maintaining a wide variety of products, which often increases the developer's turnover but also significantly reduces the cost to the customer.

1.5.1 Off-the-shelf product

An off-the-shelf product is a software product that has not only one target customer, but a wider group of customers. These customers have partly similar and partly different needs, but their requirements are similar enough that the same product can be offered using customer-specific configurations. The main advantage of an off-the-shelf product is that it is less customised, which means less defining, developing and testing from a developers point of view. Off-the-shelf product differs from a customised product in that the customer approaches software procurement in a product-oriented way [4]. From that perspective, the focus of the software project is on mapping the market of existing products rather than on selecting a suitable collaboration partner for complete development process. Excellent knowledge of the market and customer needs is key to the development of an off-the-shelf product.

1.5.2 Custom/Tailored software

Custom software is designed and developed to meet the individual needs of a specific organisation or group of users. As custom software is directly designed for a specific customer, it enables high level of customisation to take into account the customer's specific needs and wishes. For the customer, the focus is on selecting the most suitable software vendor/partner for the project [4]. For the developer, this particular approach requires more resources, as maintaining and taking into account differences and special features of each separate system takes more effort. However, it should be noted that customised software is now being built on top of both openly available platforms and the supplier's own internal platforms. This blurs the distinction between off-the-shelf and customised software.

1.6 Roles in Software develoment

A software project requires a wide range of skills, both within the development team and among the stakeholders involved in the project work. Modern agile software development, especially for smaller development teams, emphasises the involvement of each team member in all phases of development. Nevertheless, projects also require specialised skills and roles responsible for project management.

1.6.1 Customer

The customer is a person or an organisation that funds the software project and is usually responsible for managing the project and defining its objectives. To start, the customer makes a procurement contract with the chosen company defining budget, schedule and scope of the project. From the customer's perspective, the software project is usually linked to specific business targets; the primary objective of the software project is to ensure that the investment delivers the desired business benefit or solves a specific problem. As a result, the customer often perceives the project from a boarder perspective than simply as acquiring software. The customer may be part of the same organisation as the end users, but may not directly use the final product. The project success can be evaluated in a light of the Iron triangle aspects: Completed on time and within budget, including the features and functionality required to achieve the defined business objective.

1.6.2 End user

End user is a person or a group of people who actually use the software product. End user satisfaction is one of the most important criteria for measuring the success of a software project. It is also possible, that though project may be financially and administratively successful, the end user is not satisfied with the product. This is why it is important to take into account the end-users' needs, preferences and experiences at an early stage of the development project. However, it is also good to bear in mind that the end user does not know everything and is not always right.

1.6.3 Development team

The software team is made up of different groups of experts. Especially large software projects require many participants and a wide range of skills. In bigger software companies, teams can be geographically dispersed. The composition of the teams depends on the project features and the size of the project. For example, a project developing an embedded system will require software developers, electronics designers and mechanical designers. In most cases, the software team is responsible for organising its own work and defining the technical solutions to meet the customer's requirements, taking into account the constraints and boundary conditions of the project. Today, software development teams typically follow practices and working models based on agile values and principles, such as Scrum.

Product Owner

In a Scrum team, the product owner [4, s. 48] is responsible for the financial performance of the project and keeping the product vision clear. His/her role is to ensure that the team understands the needs of both the customer and the end user. The role of the product owner is to act as an interface between the many stakeholders in the software project. The product owner can be either employed by the customer or by the supplier. The product owner maintains and prioritises the product backlog according to the requirements of the system. The backlog consists of items, each with an initial time estimate and an estimate of business value. The items in the worklist can be for example:

- Product features
- Use cases
- User stories
- Requirements
- Bug reports
- Developing documentation
- Improving the architecture

Scrum Master

The Scrum Master guides the team's work and takes care of agile practices and implementation. The Scrum Master can be seen as a kind of team coach, facilitating internal cohesion, self-organisation and optimal performance of the team. His/her role is to act as a Scrum expert and to ensure that the team operates in accordance with the Scrum process. The responsibilities of the Scrum Master include monitoring the achievement of the sprint objectives. The Scrum master ensures that a task is not marked as complete until all its definition-of-done (DoD) conditions have been met, e.g. code has been written, test cases exist and run successfully, and documentation has been updated. In addition, scrum master is responsible for the team's functionality and for removing impediments to the team's work. [4, s. 49] Although the scrum master's job duties are similar to those of a traditional project manager, it is important to note that he or she is not in a supervisory position with respect to the other team members.

Software designer

The designer's key task is to work out what the software should do and how it should be implemented, both technically and non-technically. The design phase lays the foundations for the structure of the software, its functions and how it will integrate with other systems. Design roles vary and focus on different aspects of the software, e.g. architecture, user testing and interfaces, databases and so on. In the largest systems, those involved in high-level design are called software architects. Designers often use diagrams (e.g. UML) and other tools to help them design the structure.

Software developer

Software developers transform designs and requirements into functioning software by writing, testing and maintaining code. In reality, most coders read code much more than they write new. It should also be noted that writing the first version of the code takes only a fraction of the time it takes to maintain it. The work of a software developer can vary greatly depending on the project, the stage of the work and the roles of the team. In some projects, the coder may focus strictly on a particular phase of development, while in others he or she may be involved in different aspects of the project, such as design, testing and deployment.

Tester

The software tester's job is to ensure that the software developed works as intended, meets the specified requirements and is free of bugs before release. The role of the tester includes a wide range of tasks, which may vary depending on the nature of the project and the development methods used. Software testing uses several levels of testing to ensure the quality of the software at different stages of development. Each level aims to test the software from different aspects, from individual components to the functioning of the whole system. The novice software tester typically performs testing at unit, integration and system levels. In general, each software developer performs unit tests on the code they write.

Support person

A wide range of support, training and guidance is needed to use a complex software. A range of support activities and materials are available to ensure that users of the software have the necessary skills to use the system. Training can be provided not only for end-users, but also for administrators and main users. In some cases, it might be worthy to pilot a new system with a smaller group of test users. The test users can act as support staff in the customer's organisation when the system is opened up to other employees. Support methods may include e.g. user manuals, wiki pages or training videos and service-desk support. Accessibility is also an important consideration when designing technical support materials and services.

1.7 Software project tools

Software projects use a variety of tools for design, development and communication. The tools and approaches are designed to support and streamline the work of software teams. This subsection introduces some of them.

1.7.1 Development tools

Integrated Development Environment (IDE) refers to a software application that is used by programmers in developing and maintaining software code. It provides a number of code management tools such as a text editor, compiler/interpreter, debugging tools and often a

version control system. The use of an IDE makes the software development process more efficient by integrating the necessary development tools in the same environment. IDEs make the software developer's job easier in many ways; syntax highlighting makes the code clearer and easier to read, while automatic completion speeds up the writing process and reduces errors. In addition, many IDEs support software development project management, such as task tracking and scheduling. Modern IDEs are often extensible and customisable through add-ons or plug-ins, allowing developers to tailor the environment to meet their own needs or the requirements of a particular programming language or technology stack. Examples of popular IDEs include Visual Studio Code, IntelliJ IDEA, Eclipse and PyCharm, which support different programming languages and development frameworks.

AI applications using large language models have become more common in recent years and can be used to support programming in many ways. One tool that uses AI is GitHub Copilot, which uses the OpenAI GPT-3 model to suggest entire lines of code or even functions based on user-written code and comments. AI can be used for example in refactoring and optimising code, debugging and commenting. Programming assistance tools can enhance the efficiency of programming steps, but do not eliminate the importance of overall understanding and problem solving as a key programming skill.

1.7.2 Version control and CI/CD

At the heart of agile methods is an iterative and incremental way of working, which means that software is developed in short cycles that produce working software in a continuous way. Version control is the system that enables this approach and is used to track and manage code changes in a software project. Git, developed by Linus Torvalds in 2005, is one of the most popular version control systems today. Git provides a way to manage software projects by offering features such as branches, merging and change history viewing. Version control also allows multiple developers to work on the same source code at the same time. Without version control, managing the source code produced by a development team larger than a few people would be virtually impossible.

CI/CD stands for Continuous Integration and Continuous Delivery/Deployment, which are key practices in modern software development. CI/CD aims to automate the software devel-

opment and release processes, thereby speeding up and improving the efficiency of software delivery to customers.

CI refers to the practice where developers integrate their code changes into a shared repository several times a day. With each code change, automated tests (e.g. unit tests and integration tests) are performed to ensure that the new code does not break the existing software. CI aims to detect and fix bugs as early as possible reducing the complexity of problems and the cost of repair. The version control tool is the core of the CI system.

In modern software projects, the aim is to release new changes as they come in. New features and changes can be integrated into a product that is in a test environment or already in production. Continuous deployment takes the concept of continuous delivery a step further by automating the software release process so that approved changes are moved directly into the production environment without manual steps. This requires extensive test coverage and automation to ensure that releases to production do not cause problems to users. GitLab and Github are the most popular web-based software project tools today, providing a platform for software lifecycle management. Both are based on the Git version control system, combining code management, version control and CI/CD. Jenkins is another popular web-based CI/CD application.

1.7.3 Communication tools

The rise of remote and hybrid working since the Covid era has changed the way software companies work in a distinct way. Distance working is particularly prevalent in the IT sector, where the nature of many tasks allows working from any location. As distance working has become more common, communication patterns have also changed. Traditional e-mail has been replaced and supplemented by instant messaging services, which allow real-time as well as asynchronous communication. In addition, the sector is making use of distance working practices such as video conferencing applications, which allow location independent participation. Examples of commonly used remote communication tools include Teams, Zoom, Slack and Discord.

1.7.4 Other tools

Other tools are also likely to be used during the project, depending on the project and its scope, the technical environment and the client. Other examples of tools and their applications are given in the following sub-chapters and will be discussed further in future chapters.

Design

Various modelling standards and tools are used for software design and process modelling. Modelling can be used to represent the structure, behaviour and relationships between different parts of a system. This course introduces UML modelling, which can be used to depict class structures, sequences and interactions between software components. One of the most widely used modelling tools in software development is Visual Paradigm, which supports several modelling standards. For smaller projects, any program can be used that outputs boxes and lines. An easy-to-use design tool available without licence fees is Draw.io, which allows you to create diagrams directly in your browser.

Testing

Software testing tools exist for a wide range of purposes and testing needs. The choice and use of testing tools depends largely on project's requirements, the technologies used and the team's preferences. An example of a recommended testing tool is Selenium (WebDriver and IDE), an open source automation tool for testing web applications. Libraries and tools can be added to Selenium according to the testing needs. Cucumber is an example of a tool that integrates seamlessly with Selenium. It allows test cases to be written using Gherkin syntax, which resembles natural language. This makes test scripts understandable also for non-technical stakeholders.

Documentation

There are many styles and ways to document, from simple text files to visual presentations. Typically, documentation is supported by wikis and other knowledge management platforms. Take for example Confluence, a wiki-type platform from Atlassian that allows you to manage and share projects, ideas and documentation. GitHub and GitLab projects also have inte-

grated wikis, which enable the management and sharing of documentation for software projects. Projects can also use traditional text processing tools (Word, Latex), technical writing tools, API documentation, and tools that automatically generate documentation from source code. It is worth remembering that documentation is not an end in itself, but a tool to help the team, the project and the product move forward. It helps to structure information and externalises the ideas of many different people involved in the project.

1.8 Summary

This textbook is not about programming and is not a programming book. The aim of the book is to understand the overall process that results in the desired software product and to outline the role of the individual software developer in this process. The following chapters describe in more detail the different stages of software production, the tasks involved and factors that influence them. Note that many of the requirements are soft, they are not directly expressed in lines of code. These non-functional requirements underlie the work throughout, guiding the design and implementation, forming collaborative working methodology and common practices for software production.

2 Sustainable Software Engineering

Sustainable software engineering integrates environmental, economic, and social aspects into software development. This chapter explores the ethical foundations of sustainable software engineering, practical steps for green coding, and ensuring fair access through accessibility. By considering sustainability from the beginning, software developers can minimize negative impacts and create software that supports long-term well-being for both society and the planet.

2.1 Ethics in Software Engineering

Ethics is fundamental to the practice of sustainable software engineering. Ethical considerations provide the framework for making responsible decisions that consider the broader impact of technology on society and the environment.

2.1.1 The Relevance of Ethics in Software Engineering

Software have enormous potential to improve the quality of life today. Software developer plays a central role in defining and guiding the lives of people. It is therefore necessary to take ethical considerations into account in the design and development of software. It should be noted that ethics also lay foundation for the sustainable software engineering presented later in this chapter.

It is important to discuss what ethics means in practice when doing a software project. Identifying ethically charged situations requires attentiveness and sensitivity to identify ethical issues and challenges in different contexts. Understanding the stakeholders and their role in the software product environment is of paramount importance. Detection is also facilitated by picking up certain buzzwords such as privacy, security, copyright, automation, security, sustainability, energy consumption, artificial intelligence, etc. from the discussion.

2.1.2 Ethical Dimensions of Software Development

Ethics is an integral part of professional responsibility in software engineering, and the consideration of ethical issues is part of the personality of a good software professional. An ethical software developer or professional takes into account transparency, accountability and ethical principles in their work. This builds trust among the work community, customers, users and partners and creates a solid foundation for long-term success. Ethically developed software helps to protect the privacy of users, prevent harmful consequences and reduce risks.

Should failures arise, either directly or indirectly, from software engineers choosing to neglect their professional responsibilities, such actions constitute unethical professional conduct. This neglect not only breaches ethical standards but also bears the potential for severe repercussions, including harm to the public because critical software system failures can cause casualties or public injuries [9]. It then places a significant moral burden on those who are responsible [10].

Software technology also has a wider societal impact. A morally responsible developer should generally avoid technical and societal biases, when developing the software [11]. Software can affect jobs, the economy, education, child development and democracy. Ethically responsible behaviour helps avoid discrimination, misinformation, surveillance or abuse. It promotes social well-being and justice.

Also, it is best to clearly communicate to its stakeholders, the actual intents of the software without hidden agendas. For example dark patterns are design tricks used in websites and apps to distract or deceive users into making choices they would not normally make, often leading to negative outcomes for the user. one abundant usage of dark patterns is cookie consent banners in website that are designed in a way to trick the user to give consent to data collection.

Particular attention should be paid to situations involving children, health, power relations, core societal functions, sexuality, environmental issues, fundamental and human rights.

It is also important to note that ethical behaviour in software engineering is not only good practice, but often a legal requirement. Many countries and organisations have adopted privacy and security standards and other regulatory frameworks that require ethical behaviour in the design and use of software. Poor ethical choices can also expose you to reputational liability.

Ethically charged situations and ethical problems should be analysed through ethical theories as well as governing laws that are applicable.

Ethical software engineering requires making informed decisions that consider the broader impact of technology. Understanding and applying sustainable choices during development of software allows software engineers to make informed decisions that balance performance, functionality, and energy efficiency.

2.1.3 Ethical Dilemmas in Software Engineering

In the context of software engineering, ethical issues can arise in different ways and at different stages of the software production process.

For example, in the early stages of a project, we are often not yet in the right problems, in which case we talk about ethically charged situations - a situation in which there are ethical issues or challenges that require consideration and evaluation based on ethical principles. Such situations may arise in a variety of contexts and contain moral or ethical dimensions that need to be taken into account in decision-making.

Software developers are responsible for upholding ethical standards and respecting consumer rights, regardless of changing technology trends. Identifying appropriate behavior can be challenging. Below are some prevalent ethical dilemmas in software development [12].

Unethical data collection

The rise of digital marketing has increased the value of user data, which is crucial for certain software product development. It is essential for customers to be fully informed about what data they are sharing and its usage. Developers often face a dilemma between exploiting user data for system development or addressing concerns about data misuse. For example, at the development stage, software designers may face an ethical problem related to the handling of user data, data protection and privacy, especially when the data is sensitive. There may also be ethical problems in the selection of certain technology that is not energy efficient or software has unintended consequences for its users or for society at large.

Algorithmic bias

Computers do not understand morality! Bias can slip into systems unintentionally, if not carefully considered during development. For example, For instance, Google faced backlash on their latest AI image generation model, Gemini related to image generation of people from diverse backgrounds and they stopped the service [13]. Developers should consider potential biases by understanding social norms and critically analyzing data usage.

Poor security

The software sector is often vulnerable to security breaches. Developers should constantly equip themselves with latest knowledge on the security threats and mitigation techniques, and introduce necessary security measures to the software.

Poorly defined priorities

Software teams often focus excessively on developing new features at the expense of improving existing ones, which can sideline ethical considerations. This oversight can lead to ethical lapses in software development.

However, many of the so-called 'dilemmas' in ethics concern a very narrow area of software development, and are rather questions about what the software developer is willing to do in his work. For example, one could take the objection of Google's employees to the company's cooperation with the US military, or the problematic nature of various facial recognition systems, the development of self-guiding weapon systems, etc.

Resolving ethical problems often requires reflection and evaluation, balancing a number of different values and principles. This is particularly challenging because there is often no single right solution.

Example: An ethically charged situation

you are a new employee on probation in a software company. A client requests in an email (cc to your manager) that you make a customer record for them, which you discover may be in breach of data protection legislation and put their customers' privacy at risk. You need this job.

Question 1: What do you do?

Question 2: You are replying to a previous email when your manager replies to you and the client, promising that you will implement this register according to the specifications. What do you do?

2.2 Sustainable Software Development

Software development utilizes a considerable amount ecological, economical and social resources today and being sustainable about utilization is a part of the ethical software engineering. Sustainable software engineering is aimed at reducing the bad impacts of software development to the society, at large [14].

2.2.1 Environmental Responsibility

Software engineers have a responsibility to minimize the environmental impact of their work by optimizing the energy used during the development software products and their subsequent usage. For example, software systems both in normal desktop computers, mobile devices and game consoles supporting our everyday tasks, and in power plants, nuclear stations or super computers handling advanced calculations might be performing their expected tasks well but at the cost of consuming extensive amount of energy and thus emitting green house gases to the environment, if not running with a computational efficiency.

It is an ethical responsibility that a software engineer codes with environmental sustainability in mind while continuing to raise own awareness about latest sustainable coding practises.

2.2.2 Green Coding

Creating more efficient algorithms that use fewer resources is central to green coding since it directly reduces memory usage and, consequently, the energy consumption of the application.

When practising green coding, a software developer should be able to choose **energy-efficient programming languages** that suit the requirements of the client.

In terms of web services, **optimizing the data handling** between services, for example, data compression before transmission or minimizing unnecessary data transfers will reduce computational overhead which will result in less energy consumption.

Also, improving the **efficiency of the code** itself is a crucial aspect because it requires less computational power and, consequently, consumes less energy. To achieve this the following approaches could be utilized.

Non-pessimization is the practice of avoiding inefficient coding from the outset. It's about writing code that is not unnecessarily complex or resource-intensive.

Algorithm optimization is improving the logic and efficiency of algorithms to lower their computational complexity, which in turn reduces the energy used during their operation.

Loop optimization includes minimizing the overhead from repetitive loop execution, through various methods, for example, by avoiding loop dependencies.

Parallel processing can notably enhance execution speeds and cut down on energy consumption by effectively distributing tasks across several cores.

Green UI/UX Design

Technical optimization is one side of the green software development. Other side is optimization of the user experience to support more sustainable use of the unerlying system and also save users' time for more meaningful and hopefully sustainable tasks. Creating user interfaces that are not only user-friendly but also energy-efficient plays a major role in sustainable software development. Excessive animations and complex user interfaces increases power consumption. Sometimes, the unorganized and excessive use of UI/UX elements can even diminish the whole user experience.

Avoiding dark patterns in UI/UX design is crucial for green coding. These deceptive practices are not only unethical and sometimes illegal, but they also increase energy consumption.

First, dark patterns lead to higher use of website analytics by tricking users into consenting to data collection, which activates these tools. Second, they add unnecessary elements and complicate the interface, making users spend more time and clicks on the site, which further raises energy usage.

2.2.3 Measurements of Power Consumption

Without means to objectively analyze the power consumption of a software system it can be hard to really to improve the environmental sustainablity of the system let alone determine if there is need for further optimization. In measuring energy utilized by a given software both hardware (e.g.AC-meters, Meters connected to the DC -power supply, Integrated power measuring circuits) and software tools (e.g.: Intel PCM[1], Syspower[2], Windows E3[3], Website Carbon Calculator[4]) can be used. Furthermore, Green Algorithms[5] is an example of a freely available online tool for estimating the carbon footprint of software operations, developed by Lannelongue et al. in 2021. [15]

Carbon Footprint and Measuring Co2 Emission

Carbon footprint is generally defined as the total emissions of carbon dioxide (CO2) and methane (CH4) including all relevant emissions, absorption's, and storages by a specific group, system, or activity [16].

Life-Cycle Assessment (LCA) [17] and the *Greenhouse Gas Protocol (GHG)* [18] are two methods to calculate the carbon footprint in the products and processes. LCA is often used for assessing individual products and processes, whereas GHG is designed for countries and large corporations. Although calculations sometimes focus solely on CO2 emissions, it's more usual to consider the equivalent emissions of other greenhouse gases as part of the total carbon footprint. It should be noted that in many cases the majority of the CO2 emissions caused by a software system are connected to operational energy consumption of the system.

[1]Intel PCM

[2]Syspower

[3]Windows E3

[4]Website Carbon Calculator

[5]Green Algorithms

The tools such as Carbon Footprint[6] by Google, Customer Carbon Footprint Tool[7] by AWS can be used to measure the power carbon emission by a given cloud application.

2.3 Accessibility to Software Products

One part of social sustainability of software is to incorporate accessibility into the software development process involves considering accessibility at every stage of development, from design through to implementation and maintenance. It is only ethically right for a software developer to establish the fair digital access to differently-abled community based on the product audience.

Inclusive design aims to make products accessible and usable for people with various backgrounds and abilities [19]. It involves considering a wide range of user needs and preferences throughout the design and development process. Major companies like Microsoft, Apple, and Google have adopted inclusive design principles, demonstrating the importance of accessibility in creating successful, user-friendly products. Inclusive design goes beyond addressing psychophysical limitations to encompass all aspects of diversity, ensuring that products are accessible to as broad an audience as possible.

Aiming to make web content more accessible to people with disabilities, Web Accessibility Initiative (WAI) of the World Wide Web Consortium (W3C) has complied the **Web Content Accessibility Guidelines (WCAG)** [20]. WCAG is based on four foundational principles, often referred to by the acronym POUR: Perceivable, Operable, Understandable, and Robust. These principles ensure that content is accessible by providing guidelines.For example, the*turku.fi* [8] website provides *color contrast*(ensuring sufficient contrast between text and background colors) and *variable font sizes* to navigate within the site.

To effectively incorporate accessibility into the SDLC, software engineers and designers will need to focus on the following aspects.

1. Start with an understanding of the diverse needs of their user base, including those with permanent, temporary, or situational disabilities.

[6]Google: Carbon Footprint

[7]AWS: Customer Carbon Footprint Tool

[8]turku.fi

2. Adopt inclusive design principles, actively considering accessibility at every stage of the development process.

3. Ensure that accessibility considerations are integrated into the curriculum for software development and web development courses, preparing future developers with the necessary skills to build inclusive applications.

3 Starting a software project

The aim of this course is to introduce the reader to the main topics of software development in chronological order and therefore in this section we will cover topics that are primarily related to the early stages of a software project. As a result, a large number of the topics are primarily concerned with organisational and communication issues, while technical implementation and its design are largely the subject of Chapter 4.

3.1 Understanding the client

Doing a proper groundwork is essential for the software project. At the beginning of the project, it is a good idea to find out the background information of the potential customer. This preliminary research provides a basis from which to build a deeper understanding of the customer during the start of the software project. Gathering more specific information at this stage will help the project team to better understand the customer's needs, save time, and limit the potential for misunderstandings.

It is essential to know basic information about the company, including its size, industry, location, year of establishment, and number of employees. In addition, understanding the company's business model — who the products or services are aimed at, the value proposition, and the revenue logic — is important to envision how the software should support the company's objectives. In other words, you need to understand who you are making the product for.

It would also be useful to gather an understanding of the industry the company in question operates in. Current trends in the industry, legal and regulatory requirements, and specific challenges and opportunities are all factors that can influence the design and implementation of a software project. This knowledge will help to identify potential barriers and opportunities in working with the client in a timely manner. Knowledge of the products or services

offered by the client's competitors, as well as the technologies they use, provide valuable indications of what innovations or improvements the customer may be looking for. You can also ask your own networks for background information and consult open sources of information. Researching a company's website, social media, and news articles can provide insights into the company's current state, software needs, and valuable insights into its culture, values, and how it communicates with its customers and stakeholders.

3.2 Pitch your know-how / idea

Once you have an idea of how something should be done, you have to pitch it to potential customers. The customer in this sense is not just the paying customers but more generally the people whose acceptance is the key to taking the idea and going ahead with it. A customer can also be a member of your own organisation, e.g., a manager or product owner, or really anyone in a position to make decisions about how the project will be implemented.

The fact is that the world is full of great ideas. Only a small percentage of those ideas ever make it to the implementation stage, and this is very much dependent on how well you present it to the rest of the organisation, i.e., to potential customers, so that they also feel that this is the idea that should be implemented. It is also an unfortunate fact that the goodness of the idea itself is often not enough to give it the green light in the minds of the decision-makers, but it must be presented in a way that is convincing and "sellable" so that others also see the value in the idea. We will now go through some key steps on how to do this.

- Tying the idea to the wider context and goals of the organisation: this is very much due to the fact that the idea must be something that contributes to the overall purpose and goals of the organisation. It is not necessarily futile to try to extend these goals, but it is much more challenging than getting an idea through that feeds them.

- Demonstrate a clear understanding of what problem the idea addresses: a solution to a problem that the organisation does not clearly identify and that you yourself cannot explain in a meaningful way is extremely difficult to get accepted.

- Condense the idea into an easily understandable form: you need to be able to summarise

the idea clearly enough to get the main points across in "five minutes" before your audience loses interest.

- Provide a reasonable estimate of the time and resources needed: having a reasonable work plan shows others that you have thought about the problem and taken the time to consider it.

- Explain the metrics used to assess the success of a project: presenting raw facts about how to assess the success of an idea will make a huge contribution to decision-makers wanting to take it on.

- Be honest about ambiguities and weaknesses: don't lie if some aspect of the idea is not fully thought-out (because it may not be at the ideation stage), but admit any unknowns directly if they exist.

- Be able to explain your idea without the use of PowerPoint slides and other aids: you need to know your idea well enough to be able to explain it without any extra materials to the interested parties.

- The wow factor: it doesn't hurt to add an element to the idea that appeals to the listener's emotions and makes them think that this is the best thing since pre-sliced bread.

3.3 Identify stakeholders

In the context of software engineering, stakeholders are people, groups of people, or organisations that are in some way related to a software project. Stakeholders can be both internal and external. Internal stakeholders are directly related to the organisation or project, while external stakeholders are entities outside the organisation or project. Stakeholder identification is the process of identifying and understanding the individuals, groups, and organisations that are associated with the project or that may be affected by the project's activities. Stakeholder identification is a key part of project management and planning, as it helps to ensure that all stakeholders are taken into account and that needs and expectations are met.

Stakeholders in a software project may include, for example:

- Customers: software users or organisations that subscribe to or use the software. Customers can be end-users or other organisations.
- Client representative: the client's representative may have different objectives from those of the body they represent.
- Users: the individuals or organisations that will be the end-users of the software. Understanding the needs and expectations of users is important in the design and development of software.
- Uses: the software can be used to manage the information and daily life of a group of people, for example, in a retirement home, an HR department, or a senior citizen's home.
- Citizens: if it is a public administration project, the payer is often different from the client.
- Shareholders and management: the purpose of a limited company is to generate profits for its shareholders, unless the articles of association provide otherwise.
- Project team: team members involved in software development, such as software designers, programmers, testers, and the project manager. The project team is also an important stakeholder group that influences the software development process.
- Project owner: people in a managerial position in a project sometimes have different objectives from the project staff.
- Stakeholder organisations: other organisations or stakeholders that are in some way connected to the software project. For example, integration partners, subcontractors, certification authorities, or regulatory bodies may be important stakeholders.
- Administrators and other experts: the persons or organisations responsible for the maintenance, security updates, and ongoing support of the software after it has been used.

To identify stakeholders, you need to understand both the project process and the environment in which the software is produced. Categorising the parties directly involved in the

project into groups is therefore a good way to start the stakeholder analysis. It is then desirable to begin a broader analysis of the impact of the software on the environment in which it is implemented. This can be done using methods familiar from survey research to identify the structure of the organisation and the expected impact of the software product on reality. It is also important to find out what each stakeholder expects from the project or software so that their needs can be taken into account. The professional skill is to be able to distinguish between what is wanted and what is needed.

For major projects or those involving major changes, it is a good idea to consider stakeholder engagement and involvement at an early stage to avoid both design errors and user resistance. Stakeholder identification is an ongoing process, and the stakeholder list should be updated as necessary as the project progresses. This will ensure that stakeholders are taken into account throughout the project and that their needs and expectations are met.

3.4 Mapping the implementation technology

When planning a software project, the implementation technologies are largely but not entirely determined by what you are doing. The general nature of the project—e.g., whether an inventory management system for the logistics industry or a publishing system for the media, whether the system will operate over the network or locally, how many simultaneous users the system must be able to handle, how large the data volumes the system will handle, etc.—will determine and limit the possible technologies to be used in a particular way. Some frameworks have characteristics that make them better suited to building large systems, while others are better suited to building small and medium-sized systems. If the amount of data circulating in the system is huge, the backend implementation must be based on a language and database type that is well optimised for such a load, etc.

- Scalability: a very key issue to consider when choosing an implementation technology, especially when building a system with an ever-increasing number of users and/or data to be handled, is scalability, i.e., how the technology used will fit the requirements of these changing conditions. Some technologies are designed to scale effortlessly to roughly all different levels of use, but others do not scale in a meaningful way when moving to

higher volumes, which can cause significant problems and even a situation where the entire system has to be migrated to a new structure.

- Technology diffusion: certain software technologies achieve technological standardisation or at least extreme widespread use by software developers. Others do not. The more widely used a programming language, engine, framework etc. is, the better documented it tends to be, the more tested it has been in practice at different scales, the more libraries are available for it, and the easier it is to find guidance on how to solve common (and less common) problems that may arise in development. This creates an effect where "popularity begets popularity" and feeds the pre-existing tendency for other players in the field to decide to use the technology.

- Specialised uses: basically, programming is programming, and with enough work with any language or framework you can, at least in theory, do anything. However, most of these technologies are designed and intended to solve specific problems where they are in their right element, but the flip side of this is that they are not very practical for some other situation. For example, although it is theoretically possible to create web pages using C alone, in practice no one wants to try it because there are much better tools for web programming. Similarly, if you want to do signal processing systems or 3D engines, you don't do them in Java. When choosing an implementation technology, it is therefore important to familiarise oneself with the general uses, strengths, and limitations of the different technologies so that the features of the chosen technology can most effectively enable the desired implementation to be built.

In practice, however, all software companies specialise in certain types of software and as a result, maintain expertise in certain technologies. A software developer who does custom integration between systems does not code games, nor does a web developer do firmware for embedded systems because the applications and technologies used for implementation differ enormously. For this reason, the right choice in software development, the above-mentioned questions about the choice of implementation technology, are largely if not completely resolved in advance. However, websites can be built on many different platforms, reference frameworks, and languages; games can be built with different engines and languages, and so on. Very often,

however, if the customer wants an implementation using technology X and the company does not already offer solutions using that technology, the customer will look for another provider because the nature of software development means that the introduction of new technologies is a relatively large cost for the company, both in terms of money and human resources.

3.5 Software life cycle planning

Software life cycle planning, i.e., how long a system is to be used and how its use is to be phased out and transferred to a new system in an orderly manner without interruption, is an essential part of modern software development. Emphasis on the word modern because in the past this aspect, as well as many other issues discussed in this course, was not much considered when designing software, which is one of the reasons why there are still many critical systems implemented with very old technologies, which are maintained at great cost and effort because their decommissioning cannot be done in a safe way to ensure the availability of services during the migration phase to the new system. In particular, in sectors where IT was introduced early on, such as banking and finance, there are still information systems in daily use today that are implemented in languages such as COBOL, a 65-year-old programming language.

Although COBOL was originally designed for just such systems, it is clear that time has overtaken it technically, but since it has been used to build systems that are responsible for up to $3 trillion in global money transfers every day, it is not a simple operation to begin to overhaul such an architecture. This is why COBOL developers are in a very special position today where, due to the scarcity of these specialists, their skills are sought after by employers, even though in practice the job is simply to maintain old systems — and to design migrations to newer technology systems.

In general, the software life cycle can be seen as either linear or cyclical. In a linear model, software is usually intended to be used for a specific purpose for a certain period of time, after which it is run down in an orderly manner. In practice, the more common modern model is the cyclical model, where the software development cycle is repeated until it is deemed obsolete or redundant by the developer or the customer for whom it was built, at which point users are notified of the end of the life cycle and a possible migration to a new system.

The end of the software life cycle involves many issues, the solutions to which are best planned in advance. What happens to the data processed in the system? Will it be backed up? How will the system be shut down in concrete terms? Will the old data be transferred to a possible new system? As software is nowadays hardly ever distributed through physical channels but all downloads and installations are done over the network, questions about the disposal of materials, which were in the past associated with system shutdowns, have become irrelevant. To some extent, however, these same issues still apply today to what happens to the physical equipment in the system, although very often today it is either leased or located in a rented server space in data centre.

Technical debt refers to the phenomenon where software obsolescence, upgrades, poor design, and inadequate requirements specifications create situations that force other upgrades. In part then, this is what has already been discussed in this chapter — ageing technology in systems forces changes. But in part, it is a slightly more complex phenomenon which is partly independent of the natural ageing of technology. Imagine a situation where you start making changes to software — when you change one part of the software, it often "lightly requires" you to change something else, usually several other parts. This is not a "hard requirement" in the sense that the system will work in some way without these other changes being made but not optimally. So there is an accumulated technical debt that has to be paid off at some point. The more this kind of technical debt accumulates, the more work it takes to "pay it off," i.e., to implement the changes that need to be made to get rid of the debt. Another example of technical debt accumulation is a situation where the requirements specifications are not quite finished, but you still start working on the first version of the software for some reason, and the refinement of the requirements specifications then accumulates technical debt when the implementation has already been done in some direction.

3.6 Scheduling, resourcing, and budgeting

When you start developing the system, you agree on some sort of tentative timetable for when the system will be complete and ready for production. The timetable should also take into account the estimated time required for the different tasks and how the different tasks link

together to form a critical path. The critical path refers to the interlinking of tasks that must take place in a certain order because the start of the next task is dependent on the completion of the previous task.

In some cases, this schedule may be more flexible and in others more rigid, but it is equally important to understand that time is always limited and that the time spent on development work has a cost, which is made up of employee salaries and other costs to the employer — office space rentals, equipment costs, software licenses, and other costs of running the business. On top of this, of course, you also have to take into account how much profit the business is expected to make. Given the salary levels of developers and other technical staff, the costs and therefore the number in the final price tag of the project will rise very rapidly the more hours of work it involves. This issue will be covered in the exercises in this course, but it is also useful for the sake of developing your own perceptual skills to give these figures a little spin to understand the general scale of where you are when it comes to the cost of software projects.

Take for example a situation where an IT entrepreneur runs a small-scale limited company. For the sake of the example, we can assume that the company has four developers working full-time and specialises in building and maintaining websites. If we assume that the hourly wage of a developer is 20 €, just the developer's salary in such a company per month is: $146 \times 20 \times 4 = 11,680$ €. Add to this the compulsory employer's expenses, and the total rises by about 30% to just over 15,000 €. For example, if we assume that the rent for the office of such a company would be around 1000 € per month, and even if we exclude other costs such as electricity, water, internet connections, and equipment leasing costs from this estimate, we can already see from these figures some kind of indicative estimate of how much money has to come into the house each month to keep the business viable, let alone profitable. It should also be noted that the 20 €/hour assumed here is the average salary of a junior developer; senior developers and specialists must be paid significantly more. And only after all these costs have been covered can the entrepreneur start to think about the amount they would pay themselves as a salary.

If such a company contracts a project with a time estimate of 4 months and where all employees work full-time (i.e., their working time is not used for other projects), then it can be directly calculated that the minimum price to be asked for this is $4 \times 16,000$ AC = 64,000 €

and that at this price the entrepreneur does not yet make any profit on the whole job. If the entrepreneur himself is to make any reasonable profit, the price tag for a project of this kind will be at least 100,000 € or more.

In practice, of course, this is never the case, but in such small companies, each of the four developers is constantly working on several different client projects, sometimes in collaboration with each other and sometimes on their own because otherwise there is no way to make such an activity profitable without the amount of money the client has to pay becoming unbearable. In larger companies, on the other hand, several employees or even teams may be assigned to each project because the scale of the projects is also much larger.

The Gantt chart shown in Figure 3.1 is a graphical representation for representing and measuring the time required for a task, where the scale is the unit of time used, and the tasks are represented by bars of a certain length of time. The Gantt chart is a fairly old invention, but it is still widely used in scheduling. However, its weaknesses are that its graphs do not take into account the variation in time over the course of tasks or other job-related characteristics such as time use. The simplicity of the concept is a double-edged sword in this respect because although it is easy for both developers and stakeholders to understand the progress of the project, it cannot describe all the factors that influence the duration of the project.

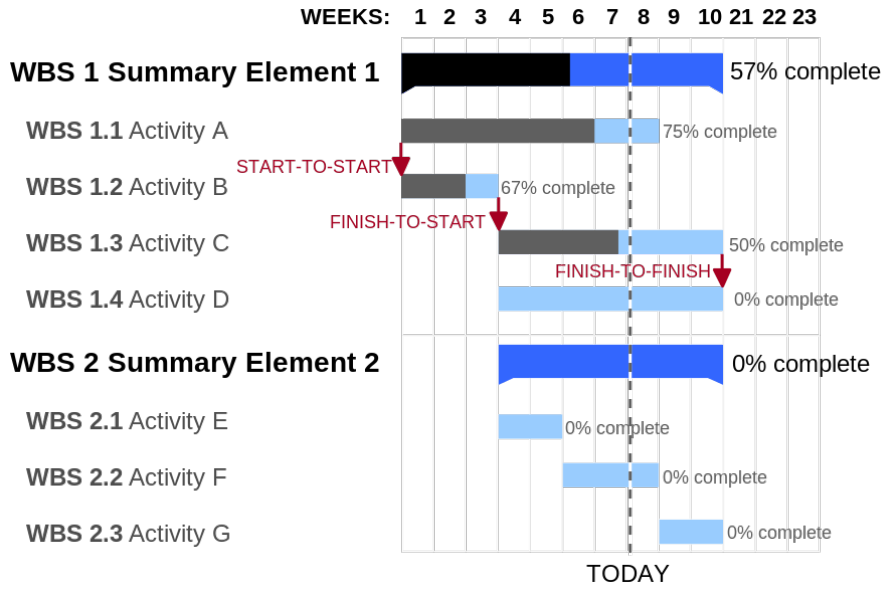

Figure 3.1: Example of a Gantt chart (Source: Wikipedia)

3.7 Selecting members of the implementation team

When considering the selection of team members, the existing resources of your organisation should be taken into account, i.e., what skills are already available in the organisation that is making the software, and how to proceed if the skills are not already available in-house. To make this process easier to understand, let's first review the most common roles in a software development organisation.

3.7.1 Skill levels of software developers

- **Trainees/Junior developers:** Developers working under this title are early in their careers and usually have very little, if any, actual work experience. In almost all organisations, their workloads are sized to reflect this lack of expertise, and junior developers are generally not required to have anywhere near the same pace of development as more experienced developers. In most cases, junior developers are also not given the same level of independent responsibility, at least at the very beginning of their career, as more experienced developers, but are usually closely supervised and monitored by a more experienced developer, such as a senior developer, who can provide guidance on technical problems and can also act as an orienter to the working practices of the organisation.

- **Mid-level developers:** This term is not used very often, but there are many "basic coders" who fit this description. These are developers who usually have a few years of work experience and are therefore well above junior developers, but not yet experienced enough to have made the transition to the ranks of senior developers or other more demanding roles.

- **Senior developers:** Developers who have reached this level are almost always professionals who have been doing software development for years or decades and have earned their place in the software development world through the quality, speed, and initiative of their work. For this reason, senior developers also, in practice, always have considerably more personal responsibility for the outcome of their work than younger developers and to some extent often play more of a managerial role.

- **Supercoders:** Supercoder is not really a job title, but it refers to highly talented and productive application developers whose workload is equivalent to the productivity of many "normal" developers. Supercoders are virtually always at least senior developers in their job titles and have decades of very broad experience in various aspects of software development.

3.7.2 Specialised roles in software development

- **Architects:** The software architect is the person who defines the broad lines of the software in the early stages of software development and envisions and guides its extensions in later stages of development. The architect does not necessarily have to be a hardcore technical expert, but very often architects are former senior developers with extensive experience in software development. In general, you might not find the title of 'system architect' in smaller software companies, but the same tasks are performed there by senior developers.

- **Front-end developer:** A developer with a strong focus on developing the user interface side of the software and related technologies.

- **Back-end developer:** A developer who is primarily focused on developing the back-end part of the software.

- **Full-stack developer:** A developer who specialises in managing a "technology stack" (i.e., interrelated front/back/database technologies) and can therefore work seamlessly with both front-end and back-end developers. Usually, such a role is developed by focusing first on one and then on the other of the roles mentioned above.

- **Specialists:** This category includes all of the roles working in software development and IT more broadly who do not actually do software development, but whose input is essential to the operation of projects and organisations. This category includes server administrators, unit testers, user managers, etc. In smaller organisations, these tasks are likely to be performed by development staff in addition to their other tasks, but in larger organisations, they are virtually always their own job titles.

When starting to think about which members to allocate to a software project, the experience level of the selected members and their possible previous specialisation in a particular aspect of software development should be taken into account. For example, if a junior developer or more is assigned to the project, they should ideally have an experienced developer present to check on the progress of the work. If the project requires the use of a technology for which the skills are not already available in the in-house talent pool, it must be considered whether it makes more sense to try to hire new developers or to train existing talent in the new tools.

Experienced developers can be expected to pick up new languages and frameworks with very little effort, while a junior developer may spend an inordinate amount of time learning something completely new. However, it should also be borne in mind that the different technologies are conceptually and sometimes also syntactically closer and closer to each other so that it may be easier to learn them if the developer already knows how to use a similar system. In short, if you need to start learning new technologies, and if you already have React and Python experts in your team, it makes more sense to have a React expert learn Angular and a Python expert learn Django if there is a need to learn one or the other.

The alternative is of course to start hiring completely new staff who already have the desired skills. This can be approached from two angles; either you hire a new employee directly or you rent temporary staff from somewhere as a consultancy. The problem with the first process can be that finding the right person always takes a certain amount of time which may not be available, and the nature of the skills required may limit the pool of available candidates considerably. The problem with the second option is that such developers are usually quite expensive, and although the developers providing consultancy services are usually quite skilled, this does not necessarily guarantee that an author brought in quickly from outside will be well suited to the working environment. In addition to the actual development work, some other aspects of software development, such as graphic design, may also be best served by outsourcing the work to another company.

3.8 Build a development environment

Building a development environment means slightly different things for different types of software projects, but certain things, such as the use of version control, have become practical standards that are followed in all areas of the software industry. In this subsection, we will discuss the most important aspects of the development environment, namely version control, CI/CD pipelines and the test environment itself where the software is tested before it can be moved to production.

3.8.1 Version control

Version control systems are one of the cornerstones of modern software development, enabling both the coherent tracking of code changes, their cancellation when necessary, ensuring code availability through repositories, and the coordination of the work of multiple developers in a rational way and if necessary isolation into separate development lines that can then be merged. In practice, version control almost always refers specifically to the Git software (and the associated GitHub and GitLab environments), which is by far the most popular and used version control tool, but in reality, there are many other systems like this.

Using version control tools really effectively to get the most out of them requires some familiarity, but it is essential for the design and implementation of a software project that all contributors understand at least the basics of the system used. For more complex version control operations, however, there should be at least one person on the team who knows how to use the more advanced features of the version control system in question and can, if necessary, solve any problems that may arise, such as Git merge conflicts. In general, implementing version control is trivial. However, in some more specific use cases, for example when working with large game engines, configuring repositories is a bit more complex and requires some extra work.

3.8.2 CI/CD pipelines

Continuous Integration/Continuous Deployment pipelines refer to the design of software development in such a way that the process of testing, integrating, and moving the finished code into

production is as straightforward as possible, hence the term pipeline. An essential part of such a pipeline is the automation of all possible aspects so that as little human effort as possible is required in any part of the release process. Large version control providers such as GitLab and GitHub offer their own CI/CD tools that allow these processes to be easily integrated with the actual version control into a single process where automated system tests new features in code and then automatically takes them to production without the need for humans to actually observe the process.

The first part of the CI/CD concept, Continuous Integration, contains the part of the process where changes to the source code are tested and integrated into the repository automatically. The second part, Continuous Delivery, is sometimes referred to as Continuous Deployment, which is a detail to pay attention to. The issue is that these two almost similar concepts do not mean exactly the same thing but in fact when we talk about Continuous Delivery, we mean a pipeline where before going into production someone has to sign off on the changes, whereas in Continuous Deployment the pipeline is fully automated in the way described in the previous paragraph.

3.8.3 Test environment

To ensure that everything works in practice as it should, it is usually a good practice to build a test environment with the characteristics of a production environment. The idea is to create an environment that is configured so that when the application under development is run in it, it gives a response that is exactly the same as what is wanted in production.

In practice, this means that databases, servers, cloud environments, and other variable elements are configured to be perfect replicas of what will be used in production. It is also common for the actual development work to take place in a test environment via some form of remote access so that the effect of changes made to the code can be seen as quickly as possible. Only when a new feature works in the test environment exactly as it should in production is it taken forward to production.

3.9 Preparing the ground for teamwork

When a new project is started and the team members have been finalised, it is necessary to hold a meeting to go over the general outline of the project so that everyone is on the same page about where we are going and what responsibilities belong to whom. Of course, exact work tasks are not yet allocated because they will come with Scrum or some other work management method in their own process as the work progresses, but the team's roles are agreed upon.

This is also the time to decide on common working methods if they have not already been defined at the organisational level and to discuss which technologies are needed for the project. In a good situation, a team operates like a well-tuned machine where everyone knows their domain and things flow smoothly and under their own weight.

3.10 High-level components and interfaces

In today's world, there is very little software that would work entirely alone. Almost all systems have some form of interconnection with another system, mainly in the sense that information is transferred either continuously or periodically between multiple different systems. Usually, this is either to retrieve some information from one system or to send information to another system that provides services that cannot be provided by its own system, the parameters of which are then sent. A very common example of this is text messaging systems; sending text messages over the network requires certain technical features that are possible but relatively laborious to develop in-house. For this reason, it is common to use service providers that offer a paid API that allows you to send the content of a message from your system to another system that will handle the delivery of the message to the recipient. Basically, when we talk about an information system of a large company such as an insurance company or a bank, we are really talking about a set of systems with several separate information systems that interact with each other and together form one large system.

To make this possible and organised, interfaces must be defined between these systems describing how system A can request information from system B without needing to know anything about B's properties except for the interface. System B must therefore provide an API, which defines how A can formulate requests in such a way that B can provide the desired

data and vice versa. This allows completely different organisations or teams to be responsible for the operation of two different data systems without knowing anything about how the other system is implemented as long as the interfaces are defined in such a way that data transfer is possible.

As a result, virtually all systems today offer such an API, which is usually very straightforward and simple to use. However, when defining interfaces, it is important to understand what data the users of the interface are allowed to access and how. For example, very recently a large Finnish software house which maintains a very widely used system to support teaching gave access to the system's interface to a third party who was developing their own application alongside the original system. As a result, personal data processed on the system inadvertently became available to the third party's users.

The design of the software API should start early in the development process, taking into account not only usability but also the availability and accessibility of information and of course other factors that may be related to security and privacy. In addition to designing your own API, it is vital to identify what data or services the software needs from other systems, how their APIs work, and to plan how to use these to develop the components that retrieve or send data in the desired way.

3.11 Leveraging existing code

Modern software development relies very heavily, perhaps even too heavily, on using libraries and modules coded by someone else doing the software development and reusing the building blocks that your own team has built in the past. This is of course a positive development in terms of workload as it allows things that used to take days or weeks to code to be available as ready-made packages often for free. However, when talking about third-party solutions, it should be noted that the development and updating of these components is often beyond the reach of the in-house team. Furthermore, in the case of systems developed or maintained by a third party free of charge, there is virtually no guarantee that these components cannot be changed into something completely different or simply discontinued without notice. In the case

of commercial components, this is not nearly so common. For the most part, this is not a problem, but it can become one in some cases.

When starting a software project, it is therefore important to map the extent to which you can find code that can be reused in a new project, and teams that have been developing software for a long time tend to have quite large libraries of ready-made implementations of often repeated functions or at least templates that can be quickly adapted to new uses. This is partly because software houses tend to concentrate on producing a particular type of software, e.g., websites or games or "integration blocks" or production control systems so that the basic structure of successive projects tends to remain somewhat similar. This is partly because some programming languages inherently encourage the construction of structures that are as reusable and generic as possible and can be applied to a wide range of problem-solving.

If you can't find a ready-made implementation from your own previous projects, the chances that someone else has already created one are very high. This does not mean that there is a pre-coded solution to every problem that you can implement in your own software at the click of a button, but especially for commonly used languages, there is a very wide range of libraries available.

4 Software development process

In this section, we will focus on the development process itself, as the title suggests, again trying to maintain at least an indicative chronological order in the way things are presented.

4.1 Functional and non-functional requirements

Software requirements can be divided into two distinctly different sections, namely functional and non-functional requirements. In very simple terms, functional requirements define what the software must do, and non-functional requirements define how the software does it.

So you could say that the functional requirements are much more important, because the things they specify are necessary for the software to work at all. If these requirements are not met, the software will not do what it is designed to do. If, on the other hand, the non-functional requirements are not met, the software will still basically do what it is intended to do, but it will not do it in a way that is aesthetically, usability-wise or otherwise experientially desirable.

However, since most software development in modern times is not done in academic ivory towers where usability or appearance is irrelevant, but very much to meet the needs of paying customers, the importance of non-functional requirements has also grown considerably. The modern consumer wants software that not only works well, but is also aesthetically pleasing and intuitive to use. It should also be noted that non-functional requirements are generally considered to include various aspects of security and user privacy in the operation of the software.

It is possible and often most reasonable to describe the functional requirements as different scenarios, known as use case diagrams. These diagrams describe the functions of the software as clearly and at the highest level as possible. They are not intended to be definitive blueprints

of how the software will actually be built, but rather guidelines for what will happen when the various functions are performed.

There are no equally clear guidelines and tools for defining non-functional requirements. There are tools for this work, but they are, like the non-functional requirements, much more open to interpretation and, on the other hand, much more general and generalisable to different types of software. For example, the use of the Nielsen heuristics, defined as early as 1994 by Jakob Nielsen, a pioneer in usability research, is to some extent established in software usability design. These heuristics consist of "ten commandments" formulated by Nielsen based on his research, which are intended to serve as rules of thumb for usability design. It must be understood, however, that non-functional requirements are also very subjective, and when designing software tailored to a specific user or customer, the customer's individual needs may be more important than, for example, adherence to the design principles mentioned above.

Although it may seem a little counter-intuitive, generally speaking, non-functional requirements include things like security and user privacy. In today's world, these are of course of paramount importance, and failure to comply with them can, in the worst case, be costly for both the operators and the suppliers of the system.

4.2 Understand the customer's need at a general level

Before any actual design or development work can be done, the customer's need for the software's functionality must be thoroughly understood, in other words, a requirements definition must be made in which the functional and non-functional requirements for the software's functionality are clearly defined. In practice, this requires a lengthy communication process with the customer, in which both the general level of what the customer would like the software to do and the functional and non-functional requirements of the software are discussed.

A very high-profile example of this is the Apotti project of a few years ago, where the Uusimaa Hospital District purchased a very expensive information system that ultimately did not meet the needs of the customer. The primary reason for this (apart from the direct technical problems associated with the Apotti implementation) was that the system supplier, Epic Software, specialises in building information systems that operate in a mainly privately owned

hospital ecosystem in the United States, which is a very different environment from the Finnish public hospital system. This difference was not sufficiently communicated during the design phase of the system, which is why Apotti has been reported to have many features that are downright useless, or even negative to work performance, because there is simply no use for them.

The software producer must also remember that the customer does not always fully know or understand what they really want. Of course, the customer has some kind of general idea of what is wanted, e.g., a website, a mobile app for X, an information system for product tracking, etc., but this idea may not be very refined at the stage when the customer and the system provider start negotiating. Depending on the client organisation's previous experience in procuring systems and its own level of technical competence, the vision of what is wanted can be very precise — but it can also be very fuzzy due to a lack of these things. On the one hand, the customer may not fully understand what is possible, on the other hand, there may be no idea of how much the desired feature will cost, and on the other hand, there may not even be any idea of what exactly the desired system should do.

For these reasons, it is vitally important that the software supplier knows how to ask the right questions in a consultation situation, and to ask for clarification on details that need to be elaborated upon. If the supplier is satisfied with the customer's initial answer about what is wanted, without trying to clarify the exact need and the desired outcome, it is likely to end up with a situation where the customer gets something slightly different from what they needed.

Sometimes, however, despite all the efforts made to identify the customer's needs, the result is a situation where the customer cannot say exactly what he wants. In such a situation, the software supplier must be able to suggest different ways of implementation on the basis of which the customer will form his final opinion, because if such open questions are not resolved and the software developers themselves decide how to solve the issues, it is very easy to end up in a situation where the customer is not satisfied with the end result.

Of course, when you talk about modern agile development, you don't want to lock everything down at the beginning and then go and implement it, because it's an iterative process. What is being sought here is the so-called co-design approach, i.e., that this communication about what the customer actually wants and what the software should do should not end with a preliminary

agreement on what is to be delivered, but that, as far as possible, customer representatives should be involved in the development process by presenting new system features to them and asking for their opinions on whether these meet the customer's needs.

4.3 Understanding the overall structure of the system

When starting to describe the architecture of a system, it is important to understand how the scale of the project affects the description process. If it is a single website, it may not make sense to start drawing detailed UML diagrams of all its functions. On the other hand, if we are talking about a system with tens or hundreds of thousands of concurrent users, very detailed modelling is extremely important.

Architectural styles are ways of categorising different types of software development architectures. There are several such styles, and it cannot be said that any of them is necessarily standard, but that the right category for a software project depends largely on what you are doing. The most traditional of these is the Monolithic Architecture, where the software package is seen as one big unit. Somewhat inversely, despite its name, it is best suited to describing small and medium-sized software, as it scales somewhat poorly to larger projects. In designing larger software today, for example, the Service-Oriented Architecture is used, where software is built on a set of disconnected units that communicate with each other through interfaces. An even more advanced model is the Microservice Architecture, where the decomposition of the software into separate units is taken to a much more detailed level, making it very well suited to rapidly evolving software. However, these two models should not be confused with the Component-oriented Architecture, which is a similar but somewhat different approach to software decentralisation. The difference between components and microservices could perhaps be seen in the fact that components are traditionally understood as parts of the software but not as separate from it, whereas a microservice that is part of the software can be, for example, a program that is completely external to the software itself. An example of such a service could be Twilio, for example, which can be used to easily implement SMS messaging over the network and can be easily deployed through the API it provides. In general, there are many

other architectural models besides those presented here, but we will not cover them in this course.

4.4 Unified Modeling Language (UML)

Modeling a system, in other words describing its functions and components with sufficient accuracy, is a very essential part of software development. While agile development has tended to move away from this approach and towards a preference for modelling from a representation of already implemented components in UML (or similar systems), the truth is that, especially when building systems of any size, modelling has a very clear purpose in terms of outlining the relationships between the different parts of the software.

Unified Modelling Language (UML) is a "modelling language" for designing software functions. It is perhaps a little counter-intuitive to talk about a language because UML is really just a set of rules that define how the different parts of a graphical diagram describing the operation of software should be structured and how they relate to each other. It is therefore not a "language" in the sense of a machine language, a search language, or any language expressed in words. However, the linguistic nature of UML is most evident in the fact that a correctly constructed UML diagram can be used with modern development tools to generate directly from the code a class hierarchy, e.g., in the sense of object-oriented programming, and conversely, code can be used with the right tools to generate UML diagrams. This process is often used today in the context of agile development practices.

Although UML was originally developed as a tool to help with object-oriented programming, it has generally had a strong influence on other software development paradigms and processes. However, UML as such, in the pure form in which it has been standardised since 1997, is not widely used today, and the diagrams used in practical software development are more free-form. However, these more commonly used free-form diagrams are significantly influenced by UML. The use of pure UML is also limited, not only by the fact that there is rarely a necessary need for it, but also by the fact that it is to some extent linked to the object-oriented programming paradigm, which has lost some popularity in software development.

When talking about UML diagrams proper, several different types of diagrams are referred to, which are described in the following diagram:

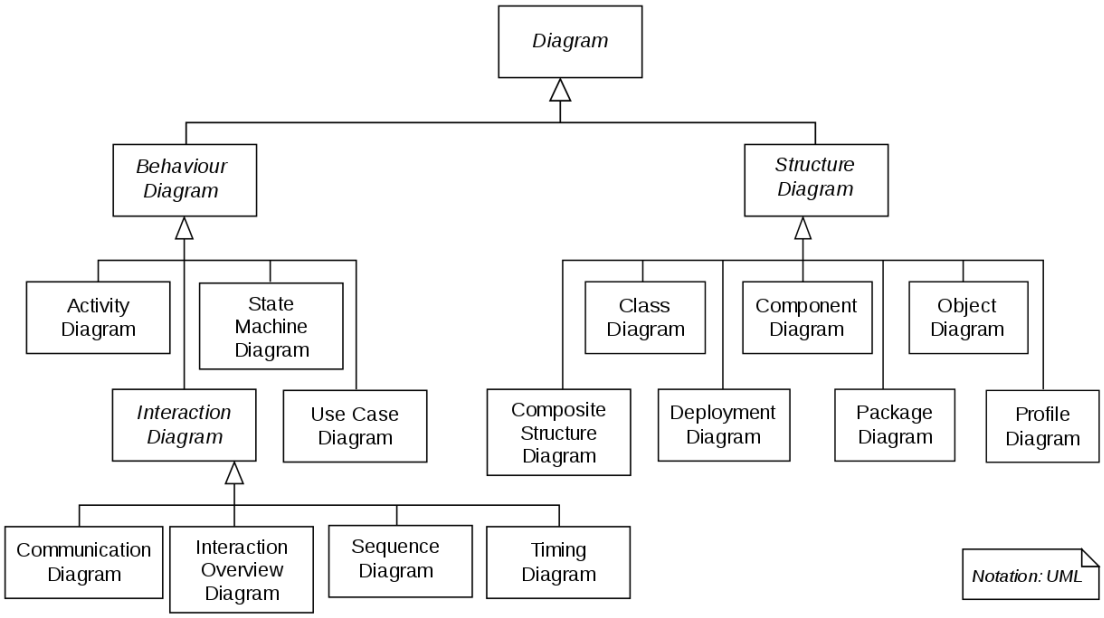

Figure 4.1: UML diagrams of different types

As we can see from the diagram above[1], UML has two main types of diagrams, Behaviour Diagrams and Structure Diagrams. Under these two main types, a number of different subtypes and their subtypes are organized to provide different perspectives on software. Structure Diagrams, and in particular Class Diagrams, are the type of UML that most people think about if they ever think about UML. Namely, a Class Diagram describes the class hierarchy of software and the key properties of classes, and can be converted directly into code using certain tools. However, UML as a whole contains many more different types of diagrams and can in principle be used to describe any aspect of the operation or structure of software.

[1] UML diagram types

UML short history

The Unified Modelling Language, or UML, was not developed from scratch, but was formed by combining ideas and methods from previous similar technologies. In the early 1990s, the golden age of object-oriented programming, a number of different systems were developed for designing software and, in particular, the structures of object-based solutions. Among these systems, the Booch Method, developed by Grady Booch, James Rumbaugh's Object-modelling Technique (OMT), and Ivar Jacobson's Object-oriented Software Engineering (OOSE), were the most important influences on the development of UML. Booch, Rumbaugh, and Jacobson worked at Rational Software in the mid-1990s, where their collaboration resulted in the building of UML on the foundations of the systems they each developed in 1994-96, which has since become a de facto standard for software development design, much like Git has become for version control. In 1997, the development of UML was taken over by the Object Modelling Group (OMG) and its first standardised version was published.

4.4.1 Theory Box: Other Modelling Languages

UML is not the only modelling language relevant to software engineering. Examples include *Business Process Modelling Notation (BPMN)* and flowchart methods.

4.4.2 Theory Box: Model-driven Engineering (MDE)

MDE is another approach to software design, including approaches like low-code and no-code systems. This encompasses different levels such as configuration, graphical application developers, and code-based programming.

4.5 Design the system for security, data protection, safety and ethical aspects

Security, privacy and ethics are not things that can be tacked on to a software product after the fact, but must be built into the system. The challenge for the software professional is therefore to keep these issues constantly in mind, even if they do not necessarily affect his or her daily work. In addition, these are complex and multifaceted issues, which often require the assistance

of specialists. These experts may be specialists working on several projects in the organisation or external consultants. In many cases, an external audit, a 'second opinion', is also obtained, which is sometimes required by regulation.

It is particularly important to note that this is not just a technical or mathematical problem, but that the developer and the development team need to understand how users work, in which environment the (software) product will be used and which stakeholders are affected by the product. It is also worth noting that mainly security-related issues, such as privacy and data integrity, are ethically charged issues, but ethically charged issues may arise in a project beyond security issues. The aim is to use technical and socio-technical measures to bring the system to the desired and 'right' way of working.

Security of use is a key consideration at all stages of the software process. This includes user interface design, error management, documentation and guidelines, and user testing. The aim of user safety is to ensure that the software is easy to use, that users understand its functions and that it works as expected.

User security is also related to the security and privacy of users. This means, for example, protecting user data, securing passwords, ensuring user consent and creating a safe and reliable experience for users when using the software. It is also important to provide users with appropriate training and support on security issues related to the use of the software, so that they understand its security features and potential risks.

Careful security planning is vital for most software systems and includes measures and practices to protect the system and the data it contains. The aim is to ensure the integrity, confidentiality and availability of the services produced and the data they contain. There are several aspects of information security, such as encryption, access control, vulnerability identification and remediation, and continuous monitoring and control.

During the software development process, security is considered at several stages. The requirements specification should take into account security requirements and risk analysis. The design should include security solutions such as strong authentication and access management. The implementation must ensure secure programming practices and minimisation of vulnerabilities. Testing must ensure that the software can withstand security auditing, and maintenance must monitor and update security-related issues.

The appropriate level of security depends on a number of factors, including the nature of the software, its intended use, user needs and stakeholder expectations. Security is not a single absolute level, but a constant balancing act between risks and resources. In general, the aim is to achieve an adequate level of security that protects the software against significant threats and risk situations. An appropriate level of security seeks to minimise risks to an acceptable level, taking into account available resources and time constraints. The level of security may also vary across different software and contexts. For example, critical infrastructure systems or applications handling personal data may have very high security requirements, while less sensitive applications may have less stringent requirements.

The importance of information security is further emphasised when taking into account the activities of the users, the environment in which the software is used and the different stakeholders that are affected by the product. It is also important to recognise that security issues such as privacy and data integrity are of ethical importance. The aim is to develop a system that, through technical and social measures, supports the desired and perceived correct approach to security.

A software product is ethical if and only if it is built on an ethically sustainable basis. In this case, the design solutions of the software product, such as data management, security, user experience design and monetisation models, are designed at the beginning of the project to withstand ethical scrutiny. (Heimo, Kimppa & Nurminen, 2014; Heimo, 2018) For example, a mobile game designed using a monetisation model, where the product is targeted at children and uses psychological addiction mechanisms and micropayments cannot be an ethically sustainable solution. (Heimo et al. 2018) Furthermore, ethical design and analysis should continue throughout the software project.

The software product should reflect not only the ethical values of society and its developers, but also the ethical values of its customers and the industry. For example, developers of software for healthcare should familiarise themselves with medical ethics[2].

[2]see for example Gillon `https://www.ht.lu.se/media/utbildning/dokument/kurser/FPRB01/20132/gillon.pdf`

Legal box: GDPR The EU General Data Protection Regulation (GDPR) is the regulation that governs the processing of personal data in the European Union. From a software engineering perspective, it requires software that processes personal data to implement appropriate safeguards, such as pseudonymisation, encryption, and be able to demonstrate compliance with these measures. This applies to the design, development and maintenance of the software.

Applying the GDPR means designing and developing software according to the principle of "privacy by design". This means that the protection of personal data must be built into the software at the design stage, not as an afterthought. It must also ensure that customers have control over their own data, including the right to delete and transfer data.

Link: GDPR

See also: this link does not go to the right place

Consider these at the beginning of the process and keep them in mind throughout the process!

4.6 Design system usability and use cases

Usability[3] in software design refers to the ability of the software to meet the user's needs successfully, efficiently and satisfactorily. The design of software system usability relates to how easy and efficient it is for the end user to use the system. In this context, ease of use refers to how intuitively the user can navigate through the software and achieve the desired goals. Efficiency, on the other hand, refers to how quickly and flawlessly the user can complete his or her tasks. The International Organization for Standardization (ISO) defines usability through five key components: ease of learning, efficiency, memorability, error rate and satisfaction.[4] In today's world, where software production is very much focused on providing services and products for the everyday needs of ordinary consumers, whether financial, entertainment or hobby, the importance of usability has become much more important than in the early 2000s or before, when software was primarily created and used in very specific work environments and often by people who were familiar with software engineering. As a result, today's design for usability, user interfaces and user experience is much more heavily resourced and virtually

[3] engl. usability

[4] ISO 9241-11:2018

all major and many smaller software companies have UX/UI (User Experience/User Interface) designers dedicated specifically to this area of software development.

Even today, students of computer science and computer engineering are sometimes confronted with harmful preconceptions that this branch of software engineering is somehow useless and less important, but it is an undeniable fact that many of the major software industry success stories of our time have been built very strongly on intuitive usability and a pleasing user experience, the clearest single example being Apple's product ecosystem and, more generally, the de facto standardisation of graphical user interfaces. If we were still living in a situation where computers were mainly used from the command line, computing would never have achieved the societal breakthrough we have seen over the last two decades, first with graphical user interfaces and mouse navigation, and later with touchscreens becoming the norm. Therefore, usability issues should not be ignored in software production, but should be treated with the same seriousness as seemingly more technical areas such as programming and system design. At the University of Turku, usability design can be specialised in the Master's degree in Interaction Design, which offers courses that provide a more in-depth understanding of the subject and its various techniques and theories.

1. Ease of use: How intuitive and easy is it for new users to learn how to use the software? The less time and effort required to learn how the system works, the better its usability.

2. Efficiency: Once a user has learned to use the software, how quickly and effortlessly can they perform the desired tasks? Efficiency can be measured, for example, by the time it takes to perform a function or the number of functions required.

3. Memorability: How easily can users return to the software after a long absence and continue using it without the need for a new learning process?

4. Error rate: How many users make errors while using the software, how serious are these errors, and how easily can they recover from these errors?

5. Satisfaction: Is the software pleasant to use, and are users satisfied with the functionality and appearance of the software?

The design of use cases seeks to determine how the software will be used in real-world situations. This process includes defining the different user roles, their tasks, goals, and needs. These "user stories" help shape the structure and functionality of the software to best serve the user's goals.

It is worth noting that usability depends on the user and the context of use: usability is not the same for all individuals or groups. Knowledge and understanding of the target audience is therefore key to usability design. It is important to remember that both usability and use case design are ongoing processes in software development. User feedback and testing are essential tools for improving these design processes and should be used regularly throughout the software lifecycle. Some methodologies have therefore emerged in usability research, such as the Co-Design Principle[21], which are intended to improve understanding of the end-user's views and to involve users in the design process itself, so that their needs and opinions are sufficiently heard.

4.7 Development coordination and management

In order for modern agile software development to be most effective, the software industry has developed and partially adopted from other industries various production management practices and systems to optimize process flow and efficiency. In this chapter we will discuss three of these methods, Scrum, Kanban and Lean, which are very commonly used and interlinked in certain ways. All of these methodologies preceded the emergence of the agile development paradigm to some extent, but have since become essential tools for its implementation.

4.7.1 Scrum

Effective coordination of the software development process is critical to the progress of the work. Various theoretical frameworks and supporting conceptual tools have been developed, of which Scrum is probably the most widely used today. It is a model of work management, which, simply put, is about meeting and deciding what to do. And then you do it. And then you have more meetings. In the Scrum framework, this meeting-execution-meeting-execution cycle is called a sprint. The length of a sprint can be anything from a day to a month and Scrum as

a technique has been developed to support the hectic and flexible pace of agile development. However, it should be noted that the length of a Sprint is traditionally never more than a month.

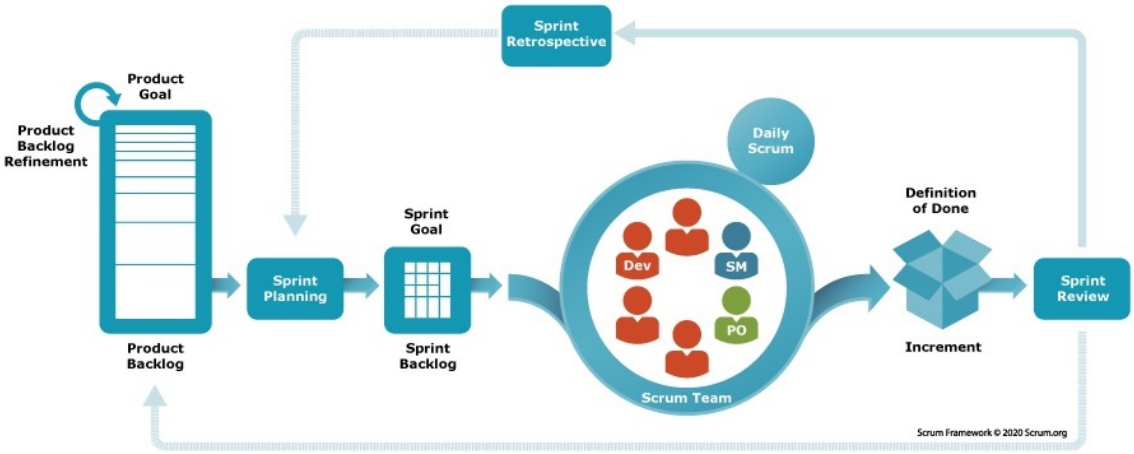

Figure 4.2: Scrum framework (Source: scrum.org)

Scrum has a lot of jargon of its own, which needs to be opened up here in order to deal with the subject at all[5]:

- **Scrum:**

  The term Scrum refers not only to the entire framework, but also to the actual meetings between sprints where Sprint Goals are set and after which the Sprint begins.

- **Sprint:**

  The execution cycle of Scrum is called a "sprint". A sprint can in principle be of any length, but due to the nature of Scrum, a sprint is usually short rather than long.

- **Sprint Goal:**

  This means the development goal that has been set in the previous Scrum, e.g. the implementation of a new feature.

- **Daily Scrum:**

  Daily Scrums are short daily meetings to quickly go over what has been achieved in the last 24 hours.

[5]Scrum's concepts are inspired by those of rugby, where it describes a pre-game situation where players in a team come together to put their heads together and make quick tactical decisions

- **Product Backlog:**

  Product Backlog is a list of work items that need to be completed. The Product Owner is responsible for maintaining this list.

Scrum teams are traditionally divided into three main roles:

- **Developer:**

  This role may not need further explanation, as the work of developers has already been covered earlier in this document.

- **Scrum Master:**

  In Scrum, the role of the foreman or team leader is by default split in two between the Scrum Master and the Product Owner. The Scrum Master is responsible for organizing the team, keeping the Scrum sprints on schedule and otherwise actively managing the work.

- **Product Owner:**

  Product Owner is Scrum jargon for someone who is responsible for the success of the project and assigning work tasks to the backlog. You could say that the Product Owner is more of a background figure than the Scrum Master, whose job is to oversee the success of the whole project and make sure that important work items are written in the backlog so that the developers and the Scrum Master can pick them up. The Product Owner is often a person with technical expertise, although this is not necessary, but rather the Product Owner is required to have an abstract understanding of the big picture and how the team's "product", in other words the piece of software they are working on, integrates into the wider software project.

Planning Poker is a gamified tool developed to assist Scrum, where the idea of poker is applied to the planning of project tasks. In the game, players use cards representing different software development concepts and tasks, and relative values for the time needed to complete them. The numbering of the cards in the game follows the Fibbonacci formula and is not intended to represent an exact workload in terms of days or weeks, for example, but rather the

comparability of the workload of one task to the workload of another. The point of all this is that when each player "blindly" bids his own presentation on the table, without the other participants being able to know in advance what each player has chosen, this has no effect on others' ideas about how to act in a particular context. The game is also a game where no one can win, the aim is to reach a consensus on what to do in the next sprint.

This technique was originally developed by James Grenning in 2002 and popularised in software development by Mike Cohn a few years later. Since it can be a little difficult to understand how this idea works in a free explanation, below is how Planning Poker is broken down into different stages of the game.

- The person acting as moderator, who does not participate in the game, acts as chairman.
- The Product Owner of the team presents a use case of some kind whose implementation should be evaluated. The other team members can ask clarifying questions and discuss the topic with each other, but they are not allowed to compare their own estimates of the resources for the next step. The person acting as moderator may summarise these discussions.
- Each player chooses a card whose value reflects his/her estimate of the resources needed and places it face down on the table in front of him/her. The value in this case could be the number of working days required, for example. Even at this stage, players are not allowed to communicate the values of their cards to each other.
- Everyone reveals their cards at the same time.
- First, the players with the highest and lowest scores are allowed to present their scores, in order to justify their scores. After that, the others may speak freely.
- The discussion on the different options continues until a consensus is reached. The moderator may also try to negotiate a consensus among the players if one cannot otherwise be reached.
- In order to maintain some sort of cohesion in the discussion and not to make the rounds

too long, the Moderator or Product Owner may set a timer at any time, at which point the turn will end and a new round of poker will be played.

As the above description suggests, Planning Poker can be a bit challenging to implement effectively. It has been criticised to some extent, particularly for being built on relative time estimates, since the human mind is not very good at evaluating things in relative terms, and players often start thinking of the game in terms of numerical estimates in days and hours, which are concrete time estimates. Another thing the game has been criticised for is the difficulty and ambiguity of the rules, which makes it difficult to absorb, especially for new users.[6] Planning Poker has also been criticised for being overly time-consuming, especially if consensus cannot be reached on the issues at hand, if some team members do not consider it a reasonable working method or if the scale of the project is too large.[7] Despite these problems, Planning Poker is quite widely used as a tool for Scrum.

4.7.2 Lean

Another essential such system is the so-called Lean thinking, developed by the car manufacturer Toyota. The idea behind Lean is to continuously measure the results of activities, to bring these results into practical work management and to eliminate "waste", i.e. practices that produce negative results, from the organisation. The name of the methodology comes largely from the fact that it "trims out" waste from production processes, thus improving efficiency itself. The major "loss" of Lean is seen as the failure to make the most efficient use of human productive potential, although the system recognises other forms of wasted potential.

From a software development perspective, Lean identifies as wasted potential, slow software performance, communication problems, unnecessary backups, ignoring innovation and new ideas, and developers not doing what they do best. To manage and eliminate these wasted elements, Lean uses a methodology called value-stream mapping, which, as the name implies, presents the steps and materials in the production process as a visual diagram that makes it easy to see the individual aspects of production and their outcomes, and to make decisions about where to trim the process to be more efficient.

[6] Why I stopped using Planning Poker

[7] Planning Poker Common Challenges and Solutions

In addition to waste, two other key concepts for understanding Lean are flow and "unevennes"[8], which are opposites of each other. Flow is the ideal to strive for; that the steps in the production process flow smoothly, and unevenness is what interferes with the flow. However, unevenness is not the same thing as waste, but rather consists of the stages of production that can cause bottlenecks in the flow, while waste, strictly speaking, is understood as things that reduce the strength of the flow. If the focus is only on wastage, it may be that the process asymmetry causes other problems. From a software development perspective, an example of this might be where a team develops a piece of software or a subset of it using a programming language that is as familiar and well understood as possible, with zero waste, but without a good enough API standard for partners and customers to effectively adopt it.

4.7.3 Kanban

Invented in the 1940s, kanban[9], also originally invented for the Toyota car factories, is a system that originally evolved to be part of Lean, but is now also used with Scrum. Kanban was originally developed based on the customer logic of supermarkets: the customer only gets what is available, and the retailer's job is to make sure that more products are added to the shelves as they are bought and that they don't sit in the stockroom unnecessarily. The idea is to manage the inventory so that there is never any unnecessary stock, but at the same time there is enough of everything available when it is needed. When this process is translated into a work-life management method, it means in practice that the amount of work is managed in such a way that tasks are not stacked up waiting in "stock", but as workers perform tasks "off the shelf", new tasks are moved to "shelves" to be picked up.

In practice, when kanban is applied to software development, the "stocks" taken from the shelf are thus the various tasks of the development process, and the "shelf" itself is a kind of labelling system on which tasks are put. In general, we talk about a kanban board[10], which is either a physical or digital board on which tasks and resources are marked with kanban cards. An observant reader may notice at this point a certain similarity to Scrum's operational

[8] jap. mura

[9] eng. visual signal

[10] What is a kanban board?

processes, which is why kanban is used a lot as a tool to assist Scrum. In practice, the Product Owner updates the tasks on the kanban board, which are then assigned to the developers by the Scrum Master.

4.8 Design Smells and Patterns

Design smells is a concept that refers to clues that something inherently wrong has occurred in the design and implementation of a system. This does not necessarily mean that the system is not working, but it always means that the system is operating in a somewhat sub-optimal way and that somewhere along the line either the wrong corners have been cut or at least the communication between the implementation team has not always worked at the required level. In the worst case, design smells cause significant problems at some point in the system's life cycle when various side effects and bugs start to appear due to these design flaws.

The term refers specifically to a kind of intuitive feeling that everything is not as it should be, and it is not always possible to pinpoint these errors to any particular thing, but various methods of analysis have been developed to assess whether the perception of design smells is correct. At their lightest, design smells are essentially sub-optimal code structures that can be easily fixed by refactoring, and which actually require such treatment. At its worst, it is a matter of, for example, different parts of the system treating the same information with different names, sometimes even different units (e.g. one part of the system defines the same value in percentages and another in decimals), which can have catastrophic consequences in the worst case.

Some common design smells include:

- **lack of abstraction level**, i.e. overuse of loose variables and other data blocks instead of having them coherently defined within classes.
- **Multi-level abstraction**, i.e. that a class does many, unrelated things, rather than being focused on providing a specific limited functionality.
- **Multiple abstraction**, i.e. that there are many classes doing the same things.

- **Broken modularization**, i.e. a situation where related data objects should be under the same class, but are distributed over several different classes.

Design Patterns: Design Patterns[11] are structures and recurring patterns that are widely recognized as useful in software production, and whose familiarity is a major contribution to the skills of the software developer. Some of these patterns are such fundamental parts of modern design work, such as the Factory Method, that many developers who use them may not even be aware that they are using a pattern that is considered a Design Pattern.

4.9 Unit and other automatic tests

Unit testing is a working method used as a standard in software development to verify the functionality of code. As the name implies, unit testing involves the automated execution of "units" or lowest-level components of software, which in most languages are methods or functions, by writing another function that is intended to test-run the function under test with a given set of parameters and compare the output it returns with a predefined output that conforms to the logic of the function under test. The unit test returns either TRUE or FALSE depending on whether the return value of the function under test was what it was supposed to be.

Nowadays, unit testing is such an important part of software development that it is commonly used at all scales of software development, from small to massive projects, and especially in larger companies, there are often dedicated software developers for the design and implementation of unit tests. In smaller companies, unit tests are usually written by the same developer who wrote the code to be tested.

Let's look at the logic of unit testing with a simple example. Assume a Python function:

```
def substract_two_numbers(x, y):
    return x - y
```

Tätä funktiota varten kirjoitettaisiin yksikkötestit:

```
def test_substract_positives():
```

[11] 22 Classic Design Patterns

```
    result = substract_two_numbers(5, 40)
    assert result == -35


def test_substract_negatives():
    result = substract_two_numbers(-4, -50)
    assert result == 46


def test_substract_mixed():
    result = substract_two_numbers(5, -5)
    assert result == 10
```

There are a few different ways to go about unit testing, which differ to some extent. For example, "Test-driven development" is a software development methodology where unit testing is actively used to verify functional requirements. The idea is that tests are first written so that their conditions match the functional requirements set for the software, so that when the actual code is written, the test not only tests its functionality, but acts as a proof that the software conforms exactly to the requirements that have been set for it. In the opposite paradigm, unit tests are written immediately after the actual coding work, so that the code can be verified as quickly as possible. Unit testing is also an essential part of CI/CD pipelines, so that unit testing is tightly integrated into the automated release pipeline. Unit testing and other quality assurance methods in software development are discussed in more detail in the course Software Testing and Quality Assurance.[12]

[12] Software Testing and Quality Assurance

V-model

Between Agile and the waterfall model, there are other philosophies of software development, one of which is the so-called V-model[13], which is an application of the V-model for systems design in a broader context.[14] In this broader context, beyond just software development, the V-model is used by federal government departments in Germany and the USA, for example, to design various projects. This model gets its name from the fact that its development phases are arranged in a V-shaped formation when presented in a logical relationship, with the design and development phases on the left side of the V and the correlated testing and maintenance phases on the right side. The idea behind the model is that the tests for each design phase are written at the stage when the design is being done. The V-model is used in software development mainly in the development of health technology devices, because for certain reasons, the agile development philosophy is not optimal for such work.[22]

Other testing methodologies: In addition to test-driven development, other methodologies are used that combine software testing with development in slightly different ways.

4.10 Implement

Sometimes we even code a little. However, since this is not a coding course, in this chapter we will discuss the coding-related working practices of software development that are essential to understand, internalize and adopt as part of one's own work as a software developer.

In previous courses, we have discussed in the context of teaching coding that proper code nesting, commenting, and variable naming practices are essential to the whole. However, the importance of these practices can be difficult to understand if your own coding experience is still limited, and especially if most of that experience consists of working alone on personal projects. When you write code that practically no one else ever reads, it's easy to get used to the fact that indentation can go wherever it wants, commenting may be occasionally present, and naming conventions are mostly lacking in coherence. But the code works and you know why it does what it does because you wrote it yourself from start to finish. Although many programming languages and frameworks have well-defined "best practices" that usually also touch on these things, and although some languages also enforce at least to some extent a certain way of doing

things (e.g. Python indentation), these do not make the code more readable if the coder does not follow these practices.

But let's consider a situation where you're working with a slightly larger team, on a slightly larger project, and with slightly larger stakes, usually because somebody paid you to write that code and that code has to be done by a certain date. In such a situation, if all the developers act as described above, the work will come to nothing. And if even one does act in the way described, he is likely to be fired if he doesn't change his ways, because the other coders who work with the code will suffer undue and unnecessary hardship from having to live with the bad working practices of another coder. If a significant amount of time is spent simply interpreting the other person's poorly formatted, confusingly named and uncommented code, this does not directly contribute to work motivation or efficiency.

For this reason, virtually all software development companies want all the developers working for them to follow the same practices for commenting, indenting and variable naming. Some companies even strive for such a high degree of uniformity that the code would not reveal any stylistic features that identify the coder who wrote it. This is all simply because the impact of such practices, which affect the readability and clarity of the code, on work efficiency and hence on the bottom line, is significant. Therefore, deciding on these practices together with the team, getting all employees to commit to them and sticking to them until they become routines that never even need to be thought about is an important aspect of personal growth in software development.

4.11 Ready-made components and configuration

Much of modern software development is based on the use of ready-made code. Of course, the proportion of this varies from programming language and purpose to programming language, but roughly speaking there are very few things that someone else hasn't already implemented in some way and created a library from that implementation that can be used to save a massive amount of work and time and grey hairs. On the utility side, this usually means using various off-the-shelf modules, but in game programming, for example, it can mean using entire game engines as the basis for your own creation.

Some frameworks and languages are more focused on such use of pre-built libraries, e.g. React and Java, and when using these languages a large part of the whole software project can be about using pre-built blocks or at least creating functionality built on top of these pre-built components. This is largely because there is a huge developer culture around both languages, which has created a considerable number of free libraries and modules that are extremely easy to adopt and deploy.

The more specialised or extensive the use of a library and the more complex the theoretical knowledge required to write it, the more likely it is that you will have to pay something to use that library. Sometimes, for example in the case of big game engines such as Unreal Engine or Unity, the charge is based on how much profit is made by using these tools, so that you can build as many free or near-free games as you like, but as soon as the revenue from the sale of the game exceeds a certain point, the company that created the game engine takes a commission. Another common billing model is that the license fees are scaled directly according to the number of members of the development team. In some cases, payment also provides other services from the developer of the module or library, such as technical support and consultation when problems arise.

4.12 Version Control (GIT)

The central idea of version control systems, as the name implies, is that the development of software code can be managed efficiently, old versions can be reverted to easily if necessary, and multiple developers can easily work on different parts of the same software without even being directly connected to each other. Nowadays, all professional software developers use version control systems, most commonly Git, and therefore knowledge of this technology is essential for all those studying and working in software development. It should also be noted that, in addition to version control itself, building up one's own code repository on GitLab or GitHub also serves as proof that one has development experience, which is why it is a good starting point for novice software developers in particular to upload all their work, from school projects to hobby projects, to one of these services. This makes it easy to showcase your coding portfolio, for example by linking it to your CV.

As we have said before, when people talk about version control systems nowadays, they usually mean Git software[15], which was developed by the Finnish Linus Torvalds in 2005. Prior to this, various version control systems had existed in some form since the 1960s[16], and there is no real reason behind the development of Git other than that Torvalds needed a version control system for his Linux project and did not want to pay the license fees for other such systems.

Over the last 19 years, Git has become the standard for version control, used in virtually all areas of computing, and has grown to include providers such as GitHub and GitLab, which maintain massive repositories of code. By some estimates, Git's share of the total version control market will be 93.3% by 2022, meaning that it has virtually eliminated all competition in this market. The reasons for this very rapid and very total market takeover are open to conjecture, but in general, the free license of the software, the high level of security, the distributed repository structure and the efficiency in dealing with even relatively large code repositories are considered to be the factors behind its success.

Deployment of Git is very simple, and really all that is required is to download the software to the machine where it is to be used. This is largely because Git is based on a distributed repository structure. In other words, Git is designed from the ground up so that each developer maintains his own code repository locally on his own machine, which means that no central repository is necessarily needed. In large software projects, of course, such a central repository is used, to which all developers working on the project add code.

The key utility of Git comes from the fact that when used, code can be forked into different development lines, which can then be merged back into a single line at a later stage. This allows different parts of the code to be edited in complete isolation from each other in separate development branches, effectively allowing, for example, the creation of different parallel versions of the same software and the safe execution of different experiments while keeping the version history of the main line clean. Also, the entire repository can be easily replicated if, for example, two significantly different pieces of software are to be built from the same base code.

[15] Git Download

[16] Version Control Systems

4.13 CI/CD

As we already briefly discussed in a previous chapter, the use of CI/CD pipelines is an essential part of modern software development. With this technology, the process of updating software, i.e. fixing bugs and bringing new features into production, can be almost completely automated, so that the amount of human effort in the whole process can be minimized to virtually nothing more than writing code, although this varies from pipeline to pipeline. When we talk about a Continuous Delivery pipeline, we mean a system where a human ultimately signs off on the changes before they are released, whereas in a Continuous Deployment pipeline this part is also fully automated. In this chapter, we will go into a little more detail about the principles of the CI/CD pipeline and how to deploy such a system. As the name suggests, this technology is divided into two parts, Continuous Integration and Continuous Delivery/Deployment.

Continuous Integration is the part of the system where new changes to the code are tested, incorporated into the main body of the software and compiled into a finished program. The idea of this whole process is that it's completely automatic, so that a human only has to write the code, with the pipeline doing everything else.

Continuous distribution/deployment is the phase that follows continuous integration, so that once the code is assembled into a working program, it is moved directly to either a test or production environment and deployed. As has already been mentioned, there is a subtle difference between delivery and deployment, since in the former case a human being verifies the correctness of the results of the integration phase and manually gives the command to deploy the software, while in the latter case these operations are fully automated.

In practice, such a system can be built using various tools, most likely decided at organisational or team level and configured by someone specialised in this type of work, so that the software developer only has to learn how to use the system. In smaller projects and companies, of course, this work may also be done by a developer on top of his other work. For example, both GitLab and GitHub offer their own tools for this work, which can be fully integrated with the project's version control, but there are other ways of implementing CI/CD pipelines also.

4.14 Communication and coordination

4.14.1 Meeting practices

For teamwork to work well, communication between team members must work. In practice, in all organisations, one of the key ways of working to achieve this is through regular meetings and briefings where much of the work planning takes place and where the relevant issues can be discussed and other team members can be kept informed of schedules and work progress. Regular practices and schedules for holding meetings should therefore be established at the start of the project, and it should be agreed which meetings it is necessary for everyone to attend. For some working methods, such as Scrum, continuous meetings are an essential part of the process, and in some cases short daily meetings (Daily Scrum) can even be held to exchange essential information about the day's agenda. However, even without a Scrum work rhythm, it is quite common to have a meeting at least once a week to report on your progress, and at least once a month for a slightly longer planning session.

4.14.2 Communication and task-assigning softwares

An essential part of effective teamwork is to organise communication so that it consumes as few resources as possible from the work itself. In addition to meeting team management, there are nowadays a variety of tools that allow both continuous communication and the organisation of work tasks so that their status is known in real time by all participants. The former includes a number of messaging applications such as Slack, Teams and Mattermost etc. that are specifically designed for the work context, and the latter includes various tagging systems that allow work tasks to be efficiently distributed within a team.

While messaging systems are now commonplace, there is a separate set of tools among these types of applications that are primarily designed for the communication needs of workplaces and other formal organisations. Unlike systems primarily designed for leisure communication such as WhatsApp or Signal, systems such as Slack or Mattermost have usability features that promote fast communication links with other team members, the ability to branch threads into separate topics and the secure availability of message history, allowing users to return to old topics with reasonable ease. Some of these applications also have built-in video calling

capabilities, allowing all communication to be efficiently centralised in a single application and eliminating the need for separate software for video conferencing, for example.

Most work-based messaging applications charge payments for use, but this is partly why companies prefer them for their internal communications, because, in contrast, applications that are "free" are almost always based on the assumption that the information they carry will be sold to advertisers, for training AIs or other third-party purposes. Therefore, they cannot be used when dealing with trade secrets or other data that must not fall into the hands of third parties, which provides a niche for communication software based on a different monetisation model.

Ticketing software is an application that allows tasks to be allocated efficiently and dynamically and to track the completion of tasks with minimal effort. Generally speaking, these systems were originally used mainly on the customer service side, but they have also become commonplace in the IT sector because of the ease of communication they allow. Many such systems can be integrated with Slack, for example, so that assigned tasks go directly through Slack to the desired user, making the ticketing process even more efficient.

4.15 Understanding customer's needs continuously

Even if the groundwork for a project and its requirements definition has been done properly, it is quite common that as the work progresses and milestones such as sprint results and other goals achieved are presented to the customer, new goals emerge that have not been considered before.

This is largely because at the stage when things are designed on paper and have not yet been seen in practice, the customer may have a very wrong idea of what they actually wanted and then, when the first version of the project is made exactly as the customer described it, the customer thinks that this is not what was wanted.

On the other hand, due to the complex nature of software, things that are essential to the functioning of the software, or at least to a reasonable user experience, may simply be forgotten in the early stages of the project.

4.16 Measuring software development

Measuring software development in this case refers both to the time and resources available, and to measuring the results of the project in some way that is meaningful to the nature of the project. Measuring the progress of the project is important not only to keep the development team informed of how the work is progressing, but also to communicate to stakeholders and other interested parties the progress of the project through clear measurement practices. Various concepts and tools have been developed to measure project progress, such as Burndown Charts and Milestones, which we will outline below.

- **Burndown Chart**[17]**:** A graph intended to show the amount of work still to be done in relation to the time available. In practice, this can mean, for example, comparing the number of instances of the Sprint Backlog with the number of working days available, from which it is possible to estimate how many working hours can be spent on a given task.

- **Milestones**[18]**:** More commonly, milestones are interim goals defined for a software project, the achievement of which indicates that the project has progressed to a certain stage. The use of milestones is nowadays standard way of measuring project progress and is widely used in other industries as well. Milestones are usually defined before the project starts, but can change during the project in line with the requirements of the agile development paradigm. In a sense, many of the software project milestones discussed in this course could themselves be considered generic milestones, achieved in a particular order.

- **Team Velocity**[19]**:** A concept that is intrinsically linked to user stories, i.e., abstractions describing the use cases of software. Team Velocity measures the number of such user stories that a previous development iteration, be it a Scrum sprint or some other work management system cycle, generated an answer to. These completed user stories can then be used to generate estimates of how much longer the project will take.

[17] Burndown Chart

[18] Milestones

[19] Team Velocity

4.17 Retrospectives

As the software project progresses, it becomes clearer whether the chosen approach makes sense for this particular project, or whether the approach needs to be changed in some way to achieve a more optimal result. In principle, this kind of adaptability, i.e. looking at things "retrospectively", is the essence of the whole agile development philosophy; that things are not necessarily locked in advance, but that approaches and working methods can be changed according to what makes the most sense to solve the problems at hand.

The importance of retrospectives to the effectiveness of software development is so widely recognised as a useful fact that they have been codified into the official Scrum routine[20]. In Scrum, the idea of a retrospective is primarily to bring the team together to discuss whether the tools used to implement the previous sprint were fit for purpose and how the issue should be developed in the future. The idea of the retrospective is not to find fault with what went wrong or to assess the quality of the work itself, but only to examine whether the tools used were fit for purpose.

4.18 Identifying and managing risks in system development

There are many risk factors associated with the software development life cycle that need to be monitored and addressed as early as possible so that these risk factors do not become real problems later in the development life cycle. While problems can always be solved, they are much easier to prevent with the right foresight and preparation.

In principle, there are two types of risk, technical and project management. We will discuss some of the most common such risks below.

General risks related to technical implementation:

- **Performance Risks[21]:** In practice, they refer to the possibility of making incorrect choices in the design or implementation of software that lead to suboptimal performance of the software. This means not only that there are bugs in the software, but also that

[20] Sprint Retrospective

[21] engl. Performance Risks

even if the software works correctly in theory, it requires, for example, significantly more computing or data transfer capacity than it would require if it were correctly optimised.

- **Data security risks:** At all stages of software development it is possible to make mistakes that lead to security vulnerabilities that can potentially be used as attack paths.
- **Integration risks:** Integrations with other systems and subsystems must be carefully designed to avoid compatibility problems or access to the wrong data.

General risks associated with project management:

- **Scope Risks:**[22] If the project requirements specifications are not properly defined, there is a risk that new targets will be added continuously as it is discovered during development that some essential aspect of the software's operation has not been addressed. This can very easily lead to a situation where the scale of the project gets out of hand and the workload becomes unfeasible.
- **Resource risks:** If project budgeting and workload estimation are not done properly, it is possible that factors such as money, time or even required hardware, e.g. server capacity, may become scarce at a critical stage.
- **Communication problems:** Effective communication between the different parties involved in a project, i.e. the implementation team, stakeholders, customers and decision makers, is essential for the success of a software project. If there are communication problems and not everyone is on the same page about what is being done, this can have a negative impact on the progress and outcome of the work.

Risk management must be actively pursued throughout the lifecycle of the software. In practice, this means holding regular meetings between all parties to discuss potential risk factors and decide how to combat them most effectively. Other ways of managing risk include various quality assessments, prototyping new features before they are implemented, and involving experienced developers and designers in the project to look at the software as a whole and provide an external expert's assessment of what risks are manageable.

[22] engl. Scope Risks

4.19 System integration

In modern times, all software is usually composed of several different parts, which are somehow integrated with each other so that the end result works like a single system. Even simple web pages are nowadays mostly implemented in such a way that the part visible to the user is implemented with one technique, and the back-end mechanisms invisible to the user with another, and these are "integrated" with each other with some third technique so that they work together. However, this is somewhat different from the "integration" that is talked about when two or more large systems are made to talk to each other and work together.

In practice, larger software companies have at least one, sometimes more, system architects for each such subsystem, who, alongside other design work, work with the other system architects to ensure that the subsystems can talk to each other. In this case, we are talking about a scale where each such subsystem is developed by a team, or even a set of teams. In smaller environments, where subsystems are both simpler and fewer in number, a single person can design the operation of all the integrations.

The definition and design of interfaces has already been discussed from a slightly different perspective earlier in **??**.

4.20 Pre-release testing and Early Access

Before software can be released to end-users, its performance, and in particular its resilience to the right operating volumes, needs to be extensively tested. In practice, this means running the software in a test environment with simulated user volumes that are assumed to correspond to future usage, and a little more, to ensure that all parts of the system can withstand the load that the actual usage will impose.

Pre-release testing can also be done with real users, recruited through their own process, who are given limited access to the test environment to enable them to use the software as it is intended to be used. This is called alpha and beta testing, where the aim is to use the testers to find bugs in the software's operation and generally to determine on a large scale that all the features of the software work as they should.

Such an alpha and beta testing process can also be used as part of the marketing of the

software, giving a selected group of users "privileged" access to the new software, which can have a significant impact on the future commercial success of the software through the publicity it generates. Facebook, which was originally restricted to US university students, is a prime example of this. Facebook spread early between 2004 and 2010, first by limiting its user base to universities, high schools and selected companies, and then, after achieving significant popularity, opened the service to all in 2006. Significantly, however, recruitment for "beta testing" only started in 2010, after the company had already grown into a major player, effectively with software that was still in the development phase.

In the games industry, this testing phase has also become a commercial activity, since games that are often released in *Early Access* mode, but for which users pay for themselves, are in practice in beta-test phase and may not work with any degree of certainty. In practice, these end-users are testers who are paying to do the work. What makes this practice even more dubious is that if a game is not successful at this stage, development can simply stop, and the consumers who paid for it have no guarantee that they will ever get real value for money or a return on their money. The perpetual *Early Access* limbo of games has become somewhat of a phenomenon. Not to sound too harsh, however, it can be argued that, especially for smaller and start-up companies developing games, the financial resources provided by *Early Access* may be the only way to get enough resources to see development through to the end.

4.21 Usability testing

Usability testing is the process of testing software with the help of real users, or their equivalent, to determine how easy and intuitive the software is to use. This helps developers better understand how the system they design meets the needs, expectations and usage patterns of real users.

Usability testing is an empirical, user-centered process that focuses on the user experience. Usability testing typically involves users performing a set of tasks on the software, and their performance is monitored and evaluated. Usability testing can measure such things as the time taken to complete tasks, the number of errors, the success of task completion, and the subjective experience of users.

Usability testing aims to identify potential problems and shortcomings in the software that negatively affect the user's experience or prevent them from performing tasks effectively. It is part of the user-centered design process and aims to ensure that the software is easy to use, effective and satisfying for the user.

Various tools and methods can be used during testing, such as interviews, observations, surveys and eye-tracking, depending on the aspects of usability that are to be investigated. The testers may be actual users of the software, or they may be people selected according to a user profile. It is important that testing takes into account the diversity of users in order to provide the most comprehensive picture of the usability of the software for different users.

Usability testing aims to continuously improve the software and optimise the user experience. When usability testing is integrated into the software development process, each new version of the product is better than the previous one, and the final product is pleasant and easy to use for the user. Usability testing results can show which parts of the software work as intended and which do not. Testing can also help identify user needs and expectations that may not have been clear during the development process.

The results of usability testing help software developers to understand where there is room for improvement in the software user interface. These results can be useful at different stages of software development, from the initial prototype to the final product. The best results are obtained when usability testing is integrated into the continuous development process.

It is also important to remember that usability testing should, as far as possible, take place in a realistic environment and usage context in order to obtain the most reliable results. The different backgrounds, skills and needs of users should also be taken into account when selecting testers in order to obtain a comprehensive picture of the usability of the software.

4.22 Managing technical debt

We discussed the concept of technical debt in general terms earlier, in the chapter **??**. In this chapter, we will focus in more detail on what technical debt is and how it is actually created, and how technical debt should be managed.

When discussing technical debt, it is important to understand the perhaps somewhat para-

doxical phenomenon that, like monetary debt, technical debt can be "acquired" on purpose, and that it can even be a positive phenomenon in some situations. An example of this kind of debt taking is, for example, focusing on getting all the functional requirements done in a given time, but as a result looking at the quality requirements with a bit of a blind eye. However, technical debt also accumulates unintentionally, mainly for reasons we discussed earlier. These reasons can largely be summarised as making decisions that are not based on correct information, e.g. by not having the requirements specifications fully refined at the time the software is implemented. Another reason for the unintended accumulation of technical debt is environmental factors, i.e. in practice the way software ecosystems evolve and therefore, for example, a version of a programming language may become "obsolete" as newer versions bring new and more powerful features to the language.

The concept of financial debt can also be used to describe the characteristics of technical debt.

- **Capital:** From the perspective of technical debt, capital is the amount of effort required to make a system work optimally. In other words, it is the initial implementation that deviates from the perfect situation.
- **Interest:** From the perspective of technical debt, interest is all those negative phenomena and effects that continually increase as time passes since the initial implementation was implemented.
- **Likelihood of realization of interest:** This concept refers to the probability of being in a situation where the effects of a suboptimal solution are reflected more broadly in the system of which the software that accumulated the debt is part.

4.23 Maintaining documentation and manuals

As mentioned earlier in the 4.10 section, documentation of the software is of paramount importance both for the smooth running of the actual development and for subsequent upgrade operations. Poorly documented code can be cryptic, even almost impossible to understand afterwards, especially for developers who did not write the code in the first place. It can be

that way for the author, too, if he or she returns to the same code only after a while, after working on other projects in between. For this reason, consistency in commenting practices is essential, so that all developers in an organisation doing development work commit to using expressions that are clearly understandable to others. Various automated systems have also been developed to assist commenting, which to some extent can generate comments from code without the developer having to spend time on them.

Another essential literary aspect of software development is the writing of user manuals. These are primarily intended for the end-users of the software, and therefore the precision of the technical details and the language used in them should be designed according to the type of software being produced and the type of people likely to read them. What is essential is that the manuals serve as an easy-to-understand reference for users in situations where something unexpected happens or a feature of the software with a low activation rate is used, i.e. where users may not have previous experience or a clear recollection of how to use the feature in question. However, user manuals should also cover general features of the system so that they can be used to train new users.

With regard to user manuals, it should be borne in mind that possible translations into other languages will largely depend on the needs of the software's customer organisation. Since there is a high risk of error, especially with technical terms, if the translation is carried out by someone who has no previous experience of translating user manuals, it may be best to outsource this part of the work to translation service providers specialising in the translation of technical documents. Nowadays, artificial intelligences can also translate texts quite well, though not perfectly, which is why human translators are still needed at least for error correction and content checking, especially for more technical material.

For the software itself, translation work is often needed for localisation, i.e. translating the content and interface into a particular language. Although English has become the universal language of the world and the majority of software users can cope with English-language content and menus, localisation is still an important part of modern software development and its popularity has been growing steadily over the last decade. The practical implementation of localisation depends on the software, and may require that the mechanisms for changing the language require work by the developers, but for many frameworks and content management

systems there are also ready-made localisation libraries and tools which, when used, do not really require anything more than having the translated content and menu texts in a database somewhere. This translation work is already being done and will probably increasingly be done by artificial intelligence in the future, so that the number of people translating texts will be significantly reduced.

4.24 Support systems and activities

Software does not live in a vacuum, but in an ecosystem where it depends on a large number of other programs to support it. These other programs include various marketing channels, customer support and feedback systems, license management systems, and so on. Although from the end-user's perspective these often appear to be part of the software itself, in reality they are mostly software in their own right, communicating with the software product itself.

The development of these support systems is often a separate project, which, depending on the customer's needs, can take place either in parallel with the development of the software product itself, or after the main software product has been released. There are also off-the-shelf solutions to implement these functions, for example feedback boxes, user surveys, chat programs and the like often seen on websites, are usually third-party services. However, off-the-shelf solutions are not always fit for purpose and even when they are, they require their own configuration. In most cases, however, this is much less work than developing the software from scratch yourself.

4.25 User training

At the stage when the software is ready for use, preferably before it is put into final production, training sessions should be held for future users, explaining in great detail how the software works and teaching end-users how to use the software correctly, what to do in case of errors and what functions each group of users has access to. At this stage, it is also useful to have well-written written instructions ready, but this cannot be left to chance; in practice, the training should take the form of a meeting so that the customer's representatives can ask clarifying questions.

Of course, this only applies to software that is tailored to the needs of a particular customer. If you are making games or websites or anything that people can buy as a ready-made product, the training is done by some kind of tutorial or, in the case of websites, by a general familiarity with the interface. However, if the website is to be used to access some other system made to the customer's specifications, then training is pretty much mandatory.

It should be noted that user training is also billable work; the more you have to do, the more you have to charge for it. If the client is unwilling or unable to budget enough money for this part of the project, the most sensible approach is to assemble key people from the client organisation, optimally those with above average IT skills, to be taught how the system works. These can then train other users in their own organisation to use the software.

4.26 Release

In the games industry, it is customary to say that "there is only one release date". By this is meant that the impact of the release's performance on the future success of the software is so significant that the one and only day the software is released must be as successful as possible. Servers must not crash, bugs must not jump out of the woodwork, people must be able to log into the service and create user accounts, and so on. However, despite all the work and testing discussed in the previous chapters, the reality is that no software is perfect when it is finally released to end users; what matters is that it looks perfect, and a lot of manpower and resources need to be devoted to ensuring that any potential problems can be solved on the fly if and when the walls start falling down at the finish line.

Getting a release up and running requires coordination with the marketing and sales departments if you are building a software product for commercial use, or with the organisation for which the software is built if it is a system tailored to the needs of a particular customer organisation.

5 Maintaining the software product and service

The final phase of the life cycle of the finished software is called the maintenance phase. The maintenance phase can last several years, even decades, so the developer's work does not end when the software is finished. It is worth noting that maintenance often accounts for a significant proportion of the total cost and time of software development. It is estimated that up to 80 percent of the total cost and time is spent on maintenance tasks.

In agile projects, it is often better to talk about the continuous development phase rather than maintenance, which better reflects the nature of modern software development. The advantage of continuous development is that new features, fixes and changes are made in smaller chunks, thus avoiding the problems caused by large individual software upgrades. The ability to respond better to changing requirements and technological environments over time is enhanced by continuous updating.

5.1 System Release

System release is a step-by-step process. The main objective of phasing is to allow early identification and correction of bugs and testing of software functionality. The development stages/maturity levels preceding the final release are usually referred to as alpha and beta. The alpha release is usually limited to software developers and internal testers. At this stage, the software may be unstable and contain only some of the intended features. Developers use this phase mainly to fix major bugs and performance problems.

The beta release phase makes use of external testers, such as beta testers or pre-users. In some cases, especially in agile development, there may be multiple beta versions or alternative testing and release cycles. In beta testing, the software is closer to the final version and aims to

gather broader feedback on software functionality, user experience and potential bugs. During the beta phase, significant changes can still be made to the software. The beta release is generally stable, but may contain some unidentified bugs.

The final release marks the software's transition from the development and testing phase to production use. Before the final release, the release process can be further staged by using a release candidate. In the final release, known bugs and problems have been fixed and the software is believed to be stable and functional enough to be used by a wide audience. Developers may continue to update the software with new features or fix problems, but the final release is considered to meet all essential requirements.

5.2 Data collection

A wide range of data can be collected about software and its use. Large datasets without structuring are of little value, but analysis of the collected data can provide meaningful insights into both the operation of the system and the business value it delivers. At its best, the information produced by the software can enable an organisation to make informed decisions and strengthen the interaction between data and knowledge management.

Post-release monitoring focuses on ensuring the performance, availability and security of the software. Bug tracking systems enable reporting, prioritisation and tracking of bugs. The collection and analysis of user feedback provides information on the usability of the software and possible areas for improvement.

Infobox: monitoring of IT infrastructure, performance analysis and network monitoring can be automated using dedicated tools. Each tool has its own specific characteristics and is suitable for different environments and requirements. An example of a commonly used monitoring software is Grafana, which is an open source software mainly for time series data visualisation and alerts. The software allows data from multiple data sources (e.g. Prometheus, InfluxDB, Elasticsearch) to be imported and visualised in real-time. It allows users to create different views of system data in the form of graphs, charts and tables.

Infobox: please also note the legal aspects of data collection during the maintenance phase. Data collection should take into account data protection regulations, such as the EU General Data Protection Regulation (GDPR) or similar national laws in different countries. Data collection must be justified and its purpose clearly defined. Pseudonymised log data can still be used for performance monitoring, behavioural analysis and planning system improvements without violating the privacy of individual users.

5.3 Monitoring

Monitoring the technical performance of the system is a key part of the maintenance phase. The practical steps of monitoring can be represented as a list of tasks, for example:

1. **Design and configuration:** Define the metrics and events to be monitored: this may include server resources (CPU, memory, disk space), network performance, application response times, database query performance, etc. Set alarm thresholds and define action processes in case of alarms.

2. **Data collection and aggregation:** Collect metrics and log data from different components of the system. Aggregate data into a central monitoring system for analysis and overview.

3. **Analysis and visualization:** Analyse collected data to identify trends, anomalies and potential problems. Use visualisation tools such as dashboards and charts to present data in an understandable format.

4. **Alerts and response:** Configure the alarm system to send automatic notifications when critical thresholds are exceeded or anomalies are detected. Organise measures to manage alarms, including automatic corrective actions or manual intervention.

5. **Continuous improvements:** Use information from monitoring as a basis for continuous improvement of system performance and stability. Update your monitoring strategy to reflect system changes and new requirements.

5.4 Ecosystem Monitoring

Ecosystem monitoring is the up-to-date monitoring of the evolution of the underlying technologies used by the system. The preparation and implementation of migrations and upgrades of e.g. third party components are a key part of product management. (Or else something like this might happen: How one programmer broke the internet by deleting a tiny piece of code)

5.5 Maintain Security

labelmaintain Security Software products are composed of many different components of varying degrees of complexity. Especially in larger projects, not all components may be in-house work, but some components may be subcontracted, purchased or exploited from freely available open source products. From a security point of view, it is essential to know the components used in your own product. The components form a product architecture where several security aspects need to be considered. Some components are more security critical than others and some components are more security sensitive. Unfortunately, even software and libraries that are perceived to be secure can have fatal vulnerabilities that the end-user may not be aware of. It makes sense to treat each component as critical to security. From a security point of view, it is best to keep software as simple as possible and to remove components that are not critical to its operation. Customers should also be educated product life cycle management. [23]

The security of components can be assessed, for example, according to a checklist developed by the Cyber Security Centre Finland [23]:

- The components use different technologies and platforms. Are we aware of their impact on the security of the product?
- Different access rights must be defined for different components. How can the principle of least privilege be applied to components?
- Different components have different update intervals. How to synchronize the updates of the components with the product updates?

- The functionality of the components may differ significantly from the core functionality of the product. Are separate test methods and other quality assurance measures required for the components?

5.6 Correct programming errors in a controlled manner

The severity of programming errors can range from an error that prevents the system from working altogether to a cosmetic detail that annoys the user. It is also estimated that about 5 percent of programming errors go undetected and the program still works despite these [4]. It is worth taking a systematic and controlled approach to correcting programming errors. The first step is to find out where and why the error occurred. After corrections and changes have been made, it is necessary to test that the corrective measures do not cause new problems due to dependencies. Repairs are transferred to the production environment in a controlled manner, minimising downtime and other disruption to users where possible.

5.7 Life cycle updates

Software components and configurations change and evolve during the software life cycle as bugs are fixed, new features are added, and new variations of components are made. In some cases, it may also be necessary to continue to develop older versions of components and configurations, which should be available when needed. A key issue in upgrading during maintenance is to understand the impact of the changes on the overall system, which means that new functionality should be added in a controlled manner. Through continuous integration, changes can be forced small enough to ensure that the version controlled entity is always ready to run. From a technical perspective, software product management is the management of components and configurations, which can be divided into three different areas (Figure 5.1 on page 90) [4].

- methods to manage different versions of the same component
- methods to manage configurations and their versions
- the common procedures followed in creating and modifying versions and configurations

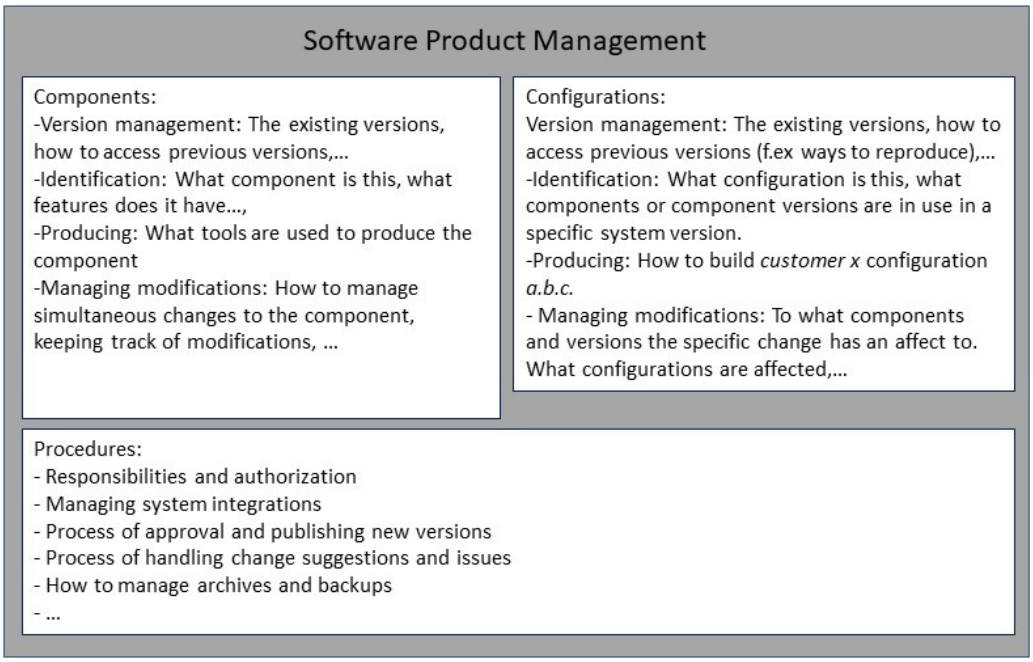

Figure 5.1: Product Management Components
[4]

5.8 Continuous testing

Continuous testing is part of the CI/CD process. During the maintenance phase, continuous testing focuses in particular on ensuring the reliability of existing software, detecting and correcting new errors, and maintaining system compatibility with new upgrades and changes. It is worth noting that the higher we are in the level of V-model of testing, the more expensive it is to correct errors. For example, if a problem is detected during system testing, correcting this error can affect several components. Higher level debugging is complicated by the fact that fixing errors can cause other errors. This is why continuous testing is important.

5.9 Update without disturbing the user

Each project defines its own best practices for running updates and these can be adjusted as the project progresses. It is important to plan and manage updates in a way that causes as little inconvenience to the user as possible. In practice, updates should be done, for example,

at night or outside office hours, taking into account time differences. For more information on life cycle updates, see 5.6 and **??**.

5.10 Maintenance

Maintenance can include updating applications running on different platforms and managing back-end systems such as databases and servers. Maintenance also includes backing up the system, preventing failures and managing change requests detected during operation. During the life cycle, particular attention should be paid to changes in the security environment and to ensuring that the security of the software is kept up to date. An essential part of maintenance is also to keep documentation and support materials up to date and to inform users of changes affecting the system and use.

5.11 End of life

An integral part of software life cycle management is the controlled decommissioning of software, a phase often referred to as end of life (EOL). A systematic decommissioning process aims to minimise the potential negative impact on customer operations and security. The first step in the decommissioning process is to identify services that are at the end of their life cycle. Reasons for decommissioning may include reduced utilisation, high maintenance costs, outdated technology or changes in the business requirements of the customer.

Once a service has been identified for decommissioning, the next step is to plan the practical implementation of the decommissioning. This includes a detailed plan on how to decommission the service with minimum disruption. The plan should include a timeline for the process, tasks and responsible parties. It is also important to ensure that all stakeholders are aware of the plan and its implications.

The measures for the downtime process depend on the system. They may include data migration or archiving, dependency management and the deployment of possible replacement services. Security should be taken into account, as often the end of support, patches and updates creates security gaps that can leave the software vulnerable. Care must also be taken when transferring sensitive data from one system to another. It is also a good idea to ensure

that all licensing and contractual issues are properly addressed in the event of a shutdown. Software decommissioning steps are documented as part of life cycle management process practices.

Case examples (in finnish) related to end-of-life and security:

National Cyber Security Centre: Decommissioning of outgoing services must be carried out carefully

References

[1] H. W. Rittel and M. M. Webber, "Dilemmas in a general theory of planning", *Policy sciences*, vol. 4, no. 2, pp. 155–169, 1973.

[2] J. F. Dooley, *Software Development, Design and Coding*. Apress, 2017. DOI: 10.1007/978-1-4842-3153-1.

[3] J. F. Dooley, *Software design problems: Wicked or tame?*, 2018. Accessed: Jun. 18, 2024. [Online]. Available: https://www.apress.com/gp/blog/all-blog-posts/software-design-problems-wicked-or-tame/15558942.

[4] I. Haikala and T. Mikkonen, *Ohjelmistotuotannon käytännöt*. Talentum Media Oy, 2011, ISBN: 978-952-14-1754-2.

[5] J. Desjardins, *How many millions of lines of code does it take?*, 2017. [Online]. Available: https://www.visualcapitalist.com/millions-lines-of-code/.

[6] O. Celkee Tivia, *Tietojärjestelmien hankinta Suomessa 2013*. Tivia, 2013.

[7] W. W. Royce, "Managing the development of large software systems: Concepts and techniques", in *Proceedings of the 9th international conference on Software Engineering*, 1987, pp. 328–338.

[8] T. Mäkilä, *Software development process modeling. Developers perspective to contemporary modeling techniques*. Turku: TUCS Dissertations No 148, 2012, ISBN: 978-952-12-2790-5.

[9] S. E. Biable, N. M. Garcia, D. Midekso, and N. Pombo, "Ethical issues in software requirements engineering", *Software*, vol. 1, no. 1, pp. 31–52, 2022, ISSN: 2674-113X. DOI: 10.3390/software1010003. [Online]. Available: https://www.mdpi.com/2674-113X/1/1/3.

[10] S. Vallor, A. Narayanan, B. Regnell, C. Jones, and R. Skipper, "An introduction to software engineering ethics", in *Applied Ethics*. Santa Clara, CA, USA: Santa Clara University, 2015, pp. 1–60.

[11] B. Cook. "Engineering: Everything you need to know about software engineering ethics", Accessed: Apr. 17, 2023. [Online]. Available: https://fellow.app/blog/engineering/engineering-everything-you-need-to-know-about-software-engineering-ethics/.

[12] F. Scale, *Software developer ethics: Avoiding ethical issues in software development*, Full Scale, 2024. Accessed: Jun. 18, 2024. [Online]. Available: https://fullscale.io/blog/ethical-issues-in-software-development/.

[13] M. Murgia. "Google pauses ai image generation of people after diversity backlash", Accessed: Apr. 17, 2024. [Online]. Available: https://www.ft.com/content/979fe974-2902-4d78-8243-a0cff68e630a.

[14] A. C. M. de Souza, "Social sustainability approaches for a sustainable software product", *ACM SIGSOFT Software Engineering Notes*, vol. 48, pp. 38–43, 2023. DOI: 10.1145/3573074.3573085.

[15] L. Lannelongue, J. Grealey, and M. Inouye, "Green algorithms: Quantifying the carbon footprint of computation", *Advanced Science*, vol. 8, no. 12, p. 2 100 707, 2021. DOI: https://doi.org/10.1002/advs.202100707. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/advs.202100707. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/advs.202100707.

[16] L. A. Wright, S. Kemp, and I. Williams, "'carbon footprinting': Towards a universally accepted definition", *Carbon Management*, vol. 2, no. 1, pp. 61–72, 2011. DOI: 10.4155/cmt.10.39. [Online]. Available: https://doi.org/10.4155/cmt.10.39.

[17] M. Z. Hauschild, R. K. Rosenbaum, and S. I. Olsen, *Life cycle assessment*. Springer, 2018.

[18] G. G. Protocol, "Greenhouse gas protocol", *Sector Toolsets for Iron and Steel-Guidance Document*, 2011.

[19] Apriorit, *The importance of accessibility & inclusiveness in ui/ux design*, https://www.apriorit.com/dev-blog/design-accessibility-in-ui-ux, 2023. Accessed: Jun. 18, 2024.

[20] WCAG. "WCAG 101: Understanding the Web Content Accessibility Guidelines", Accessed: Jun. 18, 2024. [Online]. Available: https://wcag.com/resource/what-is-wcag/.

[21] E. B.-N. Sanders and P. J. Stappers, "Co-creation and the new landscapes of design", *Co-design*, vol. 4, no. 1, pp. 5–18, 2008.

[22] M. McHugh, F. McCaffery, and V. Casey, "Barriers to adopting agile practices when developing medical device software", in *Software Process Improvement and Capability Determination*, A. Mas, A. Mesquida, T. Rout, R. V. O'Connor, and A. Dorling, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 141–147, ISBN: 978-3-642-30439-2.

[23] Kyberturvallisuuskeskus. "Turvallinen tuotekehitys: Kohti hyväksyntää", Accessed: Jun. 18, 2024. [Online]. Available: https://www.kyberturvallisuuskeskus.fi/sites/default/files/media/publication/Turvallinen_tuotekehitys_Suomi_J003_2018.pdf.