The Impact of Software Development Practices on Quality

Lauri-Mikael Pakkanen

Master of Science (Tech) Thesis
May 2025

Supervisors:
Prof. Ville Leppänen
Lauri Oikarinen

Department of Computing

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

UNIVERSITY OF TURKU
Department of Computing

Lauri-Mikael Pakkanen: The Impact of Software Development Practices on Quality

Master of Science (Tech) Thesis, 96 p., 30 app. p.
Software Engineering
May 2025

The quality of software projects is affected by a myriad of factors. One way of improving software quality is to utilise software development practices that promote or enforce software quality. The goal of this thesis is to discover software development practices that may be impactful on quality through a literature review, and to ascertain the impactfulness of the discovered practices through a survey and a case study. The research question of this thesis is: **Which software development practices are the most impactful on code quality and project success**?

The literature review resulted in the discovery of 38 software development practices. The survey analysis was able to find positive correlation between the utilisation of the software development practices and software project quality for 14 software development practices in Chapter 4. The analysis results were used to ascertain the relative impactfulness amongst the different practices in Chapter 6. The results indicate a relatively high level of correlation with quality for many software development practices, considering how many factors affect quality in software engineering. The three most impactful software development practices were found to be establishing and enforcing code style, considering higher level design issues in code reviews, and writing unit tests with the build-operate-check pattern. There were 6 software development practices whose impactfulness was rated as 5 or 4 (out of 5) by at least 75% of the survey respondents. Notably, two of them were also the most impactful on quality according to the correlation analysis. The survey response data also revealed interesting results with respect to how surprisingly many professional software projects do not require reviewing code changes before merging them, nor reviewing code generated by generative AI tools.

The results of this thesis were utilised in developing the software development life cycle (SDLC) of Evitec Solutions, the employer of the author of this thesis. The results may be utilised by individual software engineers to improve their output quality, or by organisations to develop their SDLC or similar guidelines. Moreover, they may be utilised as basis for further research – for example, the software development practices may be implemented in different ways, which could affect their impactfulness.

Keywords: Software Development Practices, Impact, Quality, Software Engineering, Software Development, Software Development Life Cycle, SDLC

Contents

List of acronyms

AI Artificial Intelligence

ASDT Author's Software Development Team; the software development team which the author of this thesis is a member of

CD Continuous Delivery

CI Continuous Integration

CI/CD Continuous Integration / Continuous Delivery

DPS Development Practice Suggestion

ETL Extract, Transform and Load

FP Functional Programming

IDE Integrated Development Environment

ISO International Organization for Standardization

LLM Large Language Model

MR Merge Request

OOP Object-Oriented Programming

PQS Project Quality Standard, the quality standard used in a particular SP

PR Pull Request

QS Quality Standard, the particular quality standard that is used in this thesis, defined in Section 2.1

RQ Research Question

RS Rating System

SDLC Software Development Life Cycle

SM Survey Method

SP Software Project

VC Version Control

VCS Version Control System

List of Figures

1 Introduction

Software engineering is an extremely complex field where almost any given business goal can be achieved in a multitude of ways. Programming, the act of writing software, is often thought of as being the core discipline in software engineering. In reality, the act of writing software is a small part of a software engineer's workflow. The field requires not only technical knowledge in terms of knowing how to write code, but also the ability to plan and design complex software projects (SP). While developing software projects alone is sometimes feasible, complex software projects often require multiple engineers and other personnel working together as a team. Adding the complexity of engineers working together with other technical and business personnel makes developing complex software projects successfully a form of art.

"Quality must be built in as a continuous process. Indeed, the type of errors and the moment at which they are detected have a major influence on the overall cost of a project."[1] Software development practices are a major contributor to code quality [2]. They may refer to a vast set of practices, however this thesis focuses particularly on practices that specify and enforce how code should be written, controlled, and documented. Because simply too many practices exist for a single thesis to cover exhaustively, the goal of this thesis is to find software development practices that are the most impactful on code quality and the probability of success in the delivery of SPs through a theory-based literature review. See Chapter 3 for the software

development practices that are discovered in this thesis.

Quality is particularly important in the financial technology (Fintech) sector in which the author is employed at the time of writing.[3] Because the term "success" in software engineering can be defined in various ways, the term is defined through the use of a quality standard in Section 2.1. While this thesis focuses mostly on Fintech software projects, the results are applicable in most web-oriented software projects. The results are likely not directly applicable to e.g. video streaming, embedded programming, and other low-level software projects.

The successful delivery of a software project is never guaranteed, however one way of increasing the likelihood of succeeding can be improved through the use of software development practices.[2] This thesis aims to analyse development practices currently suggested in literature in order to find out which are likely to be the most impactful and which are perhaps not.

The goal of this thesis is to answer the following research question (**RQ**):

- **RQ**: Which software development practices are the most impactful on code quality and project success?

1.1 Motivation

One of the primary motivations of this thesis is that the results are utilised in the process of developing the Software Development Life Cycle (SDLC) of the author's employer, Evitec Solutions, which is a Nordic software company that specialises in IT consulting and software product development in the finance and insurance sector. The company values quality in its software projects and aims to standardise the software development methodologies and practices used by all its software engineering teams through the SDLC.

The author of this thesis is also interested in the subject and chose this subject to better understand how high-quality software should be written.

1.2 Research Design and Methods

Quality is an ambiguous term that may be defined in many ways. The meaning of quality defined in Section 2.1 is used to limit the scope of this thesis by excluding certain quality characteristics from the definition. Due to the challenge of finding software development practices that are objectively impactful on quality, several research methods were chosen in order to answer the research question with sufficient depth and accuracy.

1.2.1 Literature Review

The literature review in Chapter 3 is basic research that utilises inductive reasoning to find suggestions for software development practices that are likely to be highly impactful on code quality. The review is based on two approaches: first, the analysis of the well-known book Clean Code [4] is analysed to form a baseline for development practice suggestions. Second, scientific literature is researched to augment the baseline suggestions. The review results serve as a basis for the later chapters, which attempt to validate the results empirically through a quantitative survey in Chapter 4. A failed software project is studied in Chapter 5 through a qualitative case study to find out if the identified software development practices were used in the project, and whether the lack of using certain practices led to the failure of the project.

The results of the literature review could also be used as a basis for more research subjects, e.g. furthering research on the specifics of particular software development practices. Similar research could also be conducted, for example, on the subject of information security using similar methodology and sources as this thesis.

1.2.2 Survey

The development practice suggestions found through the literature review are validated (or refuted) through a survey that aims to assess how impactful the found software development practices are in real-life software projects. The target audience of the survey is software developers and other technical personnel in professional software development projects of medium to high complexity. The respondents are asked about if and how the software development practices identified in the literature review of this thesis are applied in their software development project(s). Finally the respondents are asked to evaluate the code quality of the project(s) and the expected probability of successful delivery of the project (or if the project has already been delivered, whether the delivery was successful or not). The responses are then analysed through correlation to find out which development practices are likely to be the most impactful and which are perhaps less so in order to aid in answering the research question of this thesis.

1.2.3 Case Study

The case study in Chapter 5 is about a complex software development project in the Fintech sector that was abandoned after over a year of development. The development practice suggestions from the literature review are used to analyse the abandoned project and to test the hypothesis that the project likely failed, at least in part, due to code quality issues through deductive reasoning.

1.2.4 Limitations

Due to the comprehensive design of this thesis the realised results of the thesis should be generally applicable to most software projects above a certain threshold of complexity. Less complex projects are not considered because their development process is often quite different and because quality issues may not be as readily

apparent in them. The survey and case study of this thesis may also exhibit cultural bias as the survey is only distributed to Finnish software development companies and individuals, while the case study concerns a Finnish software project. Different cultures around the world view certain aspects of software engineering differently, e.g. the relationship between software engineers of different seniority, team dynamics and other similar factors.

1.3 Thesis Structure

The background Chapter 2 introduces the categories to which the software development practices identified through the literature review in Chapter 3 were assigned to, along with core concepts and terminology used throughout this thesis. Chapter 3 explains the literature search methodology used in the literature review in more detail and includes a list of the sources and their respective query keywords that are used in this thesis.

To answer the RQ, the results of Chapter 3 are used as a base and are validated (or refuted) by the survey in Chapter 4 and a case study of a failed software project in Chapter 5. Finally, the list of the most impactful software development practice suggestions discovered in this thesis is presented in Chapter 6, where the results are

2 Background

This thesis focuses on the impact of software development practices on code quality. The core concepts and terminology used in this thesis are briefly explained in this chapter, as well as the categories to which the software development practices identified through the literature review in Chapter 3 were assigned to. It should be noted that some software engineering experience is expected of the reader due to the technical nature of this thesis.

The Section 2.1 introduces the quality standards that were utilised to evaluate software development practices and to limit the scope of this thesis to certain aspects of quality in software engineering. Section 2.2 introduces the core concepts, terminology and software development categories that were used in this thesis.

2.1 Quality Standards

Quality is important in software engineering, especially in security-critical applications where mistakes and quality issues may cause significant harm, such as the financial technology (Fintech) sector where sensitive information and financial transactions are often handled in addition to critical societal functions. Quality in software engineering not only reduces the likelihood of bugs, vulnerabilities, and altogether failed projects but often reduces the cost associated with projects over the long term [4][1]. Quality is a multi-faceted concept and may mean different things depending on various factors. [1][5][3] Quality standards concretise the meaning of

quality, often through characterising aspects of quality. Quality standards are considered important and are widely adopted in scientific literature[1], although they are probably not often utilised in the industry by the average software engineer. Quality standards are utilised in two ways in this thesis. First, they are used to define what is meant by quality in this thesis, to limit the aspects of quality that this thesis addresses, and to base the analysis of the literature review in Chapter 3, the survey in Chapter 4 and the case study in Chapter 5. Second, software development practices regarding quality standards that may have a positive impact on software quality are suggested in Section 3.3.

The following sources were used to analyse the usage of quality standards in literature:

- Product-Focused Software Process Improvement, 2023 [6]
- Matching terms of quality models and meta-models: toward a unified meta-model of OSS quality, 2023 [7]
- Software Product Quality Metrics: A Systematic Mapping Study, 2021 [8]
- The certified software quality engineer handbook, 2009 [9]
- Software Process and Product Measurement, 2008 [10]
- Exploring Quality Attributes Using Architectural Prototyping, 2005 [11]
- Software Quality - ECSQ 2002, 2002 [12].

Additionally, the other scientific literature sources were indirectly used in the analysis because most of them also referenced quality standards and models.

In recent literature (between 2020 and 2025), various iterations of International Organization for Standardization's (ISO) quality standard ISO/IEC 25010 appears most often, along with other ISO standards which focus on different aspects of quality. An example of other ISO standards that appeared often are the ISO 9000-series standards which focus on managing quality. Quality standards and models authored

by others than ISO did appear in literature, however at much lower frequency. For example, one source compared ISO/IEC 25010 to McCall's classic model for software quality factors and other alternative standards and models [13].

In literature that was published before 2020, most sources used earlier iterations of the same ISO standards that appeared in literature published during or after 2020, or older ISO standards which the current ones are derived from. A common example of older literature using the original standard whose derivative was used in newer literature is the standard ISO/IEC 9126 (and 9126-1), which preceded ISO/IEC 25010. Quality standards and models authored by others than ISO did appear in older literature as well, however again at much lower frequency. The study "Software Product Quality Metrics: A Systematic Mapping Study" concluded that in their research, ISO/IEC 9126 appeared the most often, followed by ISO/IEC 25010 [8].

The standard ISO/IEC 25010:2023 was chosen to be used in this thesis as the ISO/IEC 25010 standard appeared most often in literature. This standard will be referred to as "QS" in this thesis. The QS defines a model for software product quality, which defines quality as consisting of the following nine characteristics: functional suitability, performance efficiency, compatibility, interaction capability, reliability, security, maintainability, flexibility and safety.[14] Because code quality is the primary focus of this thesis and the QS is quite comprehensive, some of the characteristics are excluded in the interest of limiting the scope of this thesis. The following four characteristics were chosen to be included in this thesis: functional suitability, reliability, maintainability, and flexibility.

It is acknowledged that defining quality this way may result in the results of this thesis not being applicable to all audiences. The broader quality standard may also be used as a reference for developers to ensure all aspects of quality are considered in all phases of the software development life cycle.

Functional Suitability

Functional Suitability		Included?
Capability of a product to provide functions that meet stated and implied needs of intended users when it is used under specified conditions.		
Functional completeness	Capability of a product to provide a set of functions that covers all the specified tasks and intended users' objectives.	✗
Functional correctness	Capability of a product to provide accurate results when used by intended users.	✓
Functional appropriateness	Capability of a product to provide functions that facilitate the accomplishment of specified tasks and objectives.	✓

Figure 2.1: Functional Suitability characteristic, ISO/IEC 25010:2023 [14].

Functional suitability ensures that the project provides functionality that meets the stated and implied needs of its intended users. **Functional completeness** was chosen to be excluded from this thesis due to its focus being primarily on code quality, while functional completeness relates more to testing strategies, which are excluded from this thesis.

Reliability

Reliability		Included?
Capability of a product to perform specified functions under specified conditions for a specified period of time without interruptions and failures.		
Faultlessness	Capability of a product to perform specified functions without fault under normal operation.	✓
Availability	Capability of a product to be operational and accessible when required for use.	✓
Fault tolerance	Capability of a product to operate as intended despite the presence of hardware or software faults.	✓
Recoverability	Capability of a product in the event of an interruption or a failure to recover the data directly affected and re-establish the desired state of the system.	✓

Figure 2.2: Reliability characteristic, ISO/IEC 25010:2023 [14].

Functional suitability is not enough on its own if the project does not handle faults well or is not available in the first place. Fault tolerance and availability are covered by the reliability characteristic.

Maintainability

Maintainability		Included?
Capability of a product to be modified by the intended maintainers with effectiveness and efficiency.		
Modularity	Capability of a product to limit changes to one component from affecting other components.	✓
Reusability	Capability of a product to be used as assets in more than one system, or in building other assets.	✓
Analysability	Capability of a product to be effectively and efficiently assessed regarding the impact of an intended change to one or more of its parts, to diagnose it for deficiencies or causes of failures, or to identify parts to be modified.	✓
Modifiability	Capability of a product to be effectively and efficiently modified without introducing defects or degrading existing product quality.	✓
Testability	Capability of a product to enable an objective and feasible test to be designed and performed to determine whether a requirement is met.	✓

Figure 2.3: Maintainability characteristic, ISO/IEC 25010:2023 [14].

Just like most mechanical systems, software requires regular maintenance. Maintainability is important for upholding quality throughout an SP's life cycle as it makes long-term maintenance easier, faster and cheaper which aids to ensure the long-term relevancy of the project as the business requirements of the SP change over time. [2] If the project does not possess maintainability qualities such as modularity, changing code without necessitating a complete rewrite of the software becomes difficult. Changing code also often introduces technical debt which accumulates over time if regular maintenance and technical debt reduction are not carried out. Having a well defined and strictly followed maintenance plan is crucial to ensuring that known security vulnerabilities are patched in a timely manner and that software components are kept up to date on security updates.

Flexibility

Flexibility		Included?
Capability of a product to be adapted to changes in its requirements, contexts of use, or system environment.		
Adaptability	Capability of a product to be effectively and efficiently adapted for or transferred to different hardware, software or other operational or usage environments.	✓
Scalability	Capability of a product to handle growing or shrinking workloads or to adapt its capacity to handle variability.	✓
Installability	Capability of a product to be effectively and efficiently installed successfully and/or uninstalled in a specified environment.	✓
Replaceability	Capability of a product to replace another specified product for the same purpose in the same environment.	✗

Figure 2.4: Flexibility characteristic, ISO/IEC 25010:2023 [14].

Flexibility is similar to the maintainability characteristic, however maintainability refers specifically to the "degree of effectiveness and efficiency" of making changes while flexibility refers to the overall ability to make changes.[14]

Replaceability is a sub-characteristic that is not usually a requirement in complex, custom SPs, because often a similar product does not exist. Therefore it is excluded from this thesis.

2.2 Software Development Practices

This section presents software development practices that may have a positive impact on the code quality of an SP. These software development practices are derived from the literature review and are not an exhaustive list. Most of the practices focus on how source code should be written and on managing source code over its life cycle. Quality standards may also be considered to be software development practices, however they were separated to their section as was explained in Section 2.1.

2.2.1 Documentation

Information regarding a software project's requirements may be stored as written documentation, which shall be referred to as "requirements documentation" in this thesis. Requirements documentation enables access to the information for all stakeholders of the project and may reduce conflicts in e.g. what has been agreed between the supplier and customer, especially if a process is in place where all involved parties agree on the accuracy of the documentation. Requirements documentation is often used in requirements engineering, wherein solution architects create technical specification documents for the software engineers of the project based on the requirements. Software engineers may also refer to this documentation to e.g. see what the business requirements are or the context surrounding the particular functionality. In some software projects the software engineers may even be doing the requirements engineering themselves.

Internal documentation refers to specific technical implementation information about an SP. It may be stored apart from source code, however it is possible to embed documentation in code as most programming languages support code comments. Software development practices regarding code comments are addressed in Section 3.4.3, while a few different types of internal documentation are addressed in Section 3.4.1.

2.2.2 Programming Paradigms

Programming paradigms refer to the high-level approach to writing software. They influence how SPs are to be structured and how data should flow between components of the software. Paradigms may be considered to be part of software architecture, however this thesis approaches them from the viewpoint of how the practice of adapting parts of paradigms in projects that primarily adhere to another paradigm may impact code quality of the overall project. Three main paradigms exist in soft-

ware engineering at the time of writing: procedural, object-oriented and functional programming.

The **procedural paradigm** is the simplest of the three. Simply put, a procedural SP consists of one or more instructions or *procedures* that are executed one after another until the program halts.[15] It can be thought of as a catch-all paradigm in that any code that does not adhere to the object-oriented programming (OOP) or functional programming (FP) paradigms can be thought of as procedural code.

The **object-oriented** programming paradigm is based on the "concept of wrapping pieces of data, and behavior related to that data, into special bundles called objects, which are constructed from a set of 'blueprints', defined by a programmer, called classes"[16]. OOP is largely based on a well-known set of design patterns that shape how object-oriented code should be structured and composed. OOP is most prominent in backend software development – e.g. the popular backend-focused programming languages Java- and C# can be considered inherently object-oriented due to their design.

According to Stack Overflow's 2023 Developer Survey Java and C# are among the most popular fully fledged programming languages, surpassed in popularity only by JavaScript and Python. [17] The popularity of the languages is not directly comparable but it can be deduced that object-oriented languages are very popular at the time of writing. The book Design Patterns is one of the most well known books on the subject of software engineering theory. At over 300 pages of OOP theory, Design Patterns covers OOP design patterns thoroughly and therefore it is not necessary to repeat its contents in this thesis.[16] In addition to Design Patterns, another great source of information on design patterns is the website Design Patterns by Refactoring Guru [18].

Design patterns are the core of OOP programming – in Design Patterns, Grady Booch mentions that "one of the ways that I measure the quality of an object-

oriented system is to judge whether or not its developers have paid careful attention to the common collaborations among its objects. Focusing on such mechanisms during a system's development can yield an architecture that is smaller, simpler, and far more understandable than if these patterns are ignored". [16]

The **functional paradigm** (functional programming) is a "paradigm in which the mathematical function evaluation is the main block in building the software"[19]. Pure functional programming means strictly adhering to rules laid out by the paradigm. A purely functional SP has no state nor side effects and always produces the same output for a given input. The paradigm is based in mathematical theory and functional code is often expressed using the lambda calculus through lambda expressions.

Functional programming is addressed more thoroughly than other paradigms in this thesis due to the fact that it requires certain approaches to writing code that are similar to development practice suggestions found through the literature review in Chapter 3. [4][20][19] The theory behind FP is vast and complex enough to warrant writing a paper of its own, however this thesis addresses FP on a fairly high level, focusing on the requirements of functional programming that may be applied as a development practice while using a different primary paradigm. The following section explains these particular requirements.

```
function divide(divident: number, divisor: number) {
  if (divisor === 0) {
    throw new Error("Division by zero is not allowed");
  }
  return divident / divisor;
}

function addToDivision(divident: number, divisor: number, value: number) {
  let quotient = divide(divident, divisor);
  quotient += value;
  return quotient;
}

function handle(divident: number, divisor: number, value: number) {
  try {
    const result = addToDivision(divident, divisor, value);
    console.log(`'(${divident}/${divisor}) + ${value}' = ${result}`);
  } catch (err) {
    console.error(err);
  }
}

handle(10, 0, 100);
```

Figure 2.5: Impurely functionally implemented program.

```
type MaybeErr<T> = Just<T> | Err;

class Monad<T> {
  private value: T;

  constructor(value: T) {
    this.value = value;
  }

  public unwrap(): T {
    return this.value;
  }
}

class Just<T> extends Monad<T> {}
class Err extends Monad<string> {}

function divide(divident: number, divisor: number): MaybeErr<number> {
  if (divisor === 0) {
    return new Err("Division by zero is not allowed");
  }
  const quotient = divident / divisor;
  return new Just(quotient);
}

function addToDivision(
  divident: number,
  divisor: number,
  value: number
): MaybeErr<number> {
  const quotientMaybe = divide(divident, divisor);

  if (quotientMaybe instanceof Err) {
    return quotientMaybe;
  }

  const quotient = quotientMaybe.unwrap();
  const result = quotient + value;
  return new Just(result);
}

function handle(divident: number, divisor: number, value: number) {
  const resultMaybe = addToDivision(divident, divisor, value);

  if (resultMaybe instanceof Just) {
    const result = resultMaybe.unwrap();
    console.log(`'(${divident}/${divisor}) + ${value}' = ${result}`);
  } else {
    console.error(resultMaybe.unwrap());
  }
}

handle(10, 0, 100);
```

Figure 2.6: Purely functionally implemented program.

Figure 2.5 represents the source code of a simple SP written in TypeScript that

first gets the quotient from a division, adds a value to it and logs the resulting value to the console. The program is impure because the function `divide` throws an exception while the function `addToDivision` modifies (mutates) a variable's value. The source code of the same program re-implemented purely functionally is represented by Figure 2.6. The function `divide` uses a `MaybeErr`-monad (see definition in Figure 2.7) instead of throwing an error while the function `addToDivision` no longer mutates the `quotient` variable. Even in this simple example the purely functionally implemented program requires more lines of code and complexity than the impure implementation.

```
Monads are mathematical concept that may be simplified in
functional programming as "increasing the representational
power of their underlying type".
```

Figure 2.7: Definition of a monad. [21]

2.2.3 Coding Convention Practices

Coding conventions refer to software development practices chosen for the development of an SP which define the formats that written code should adhere to. The respective literature review Section 3.4.3 focuses particularly practices regarding code readability, continuous quality improvement and generative artificial intelligence tooling.

2.2.4 Version Control Practices

Version control (VC) in software engineering refers to the practice of managing source code and its version history. It enables recoverability by making it possible to switch between different versions of the source code. Additionally it enables accountability and traceability of changes made to the code base, which is particularly

useful in a team setting where multiple persons may be interacting with the source code.

There are multiple version control systems (VCS) for implementing version control. The most popular one at the time of writing is Git, which was originally developed by Linus Torvalds who is also known for developing the Linux kernel.[22] Git is considered the de-facto VCS in software engineering at the time of writing and therefore this thesis focuses solely on it. The VC practices explained in Section 3.4.4 may not be applicable if another VCS is used in place of Git.

2.2.5 Merge Request Practices

Software engineering is a team activity. Everyone in the team must agree with - and be dedicated to - the chosen software development methodologies and practices. One method of improving consistency across a code base is utilising code merging tools such as pull requests (PR) or merge requests (MR). Both of these terms refer to an asynchronous process where a set of changes is submitted for review to be merged to a target Git branch. This thesis uses the term MR as it describes the process more clearly than PR.

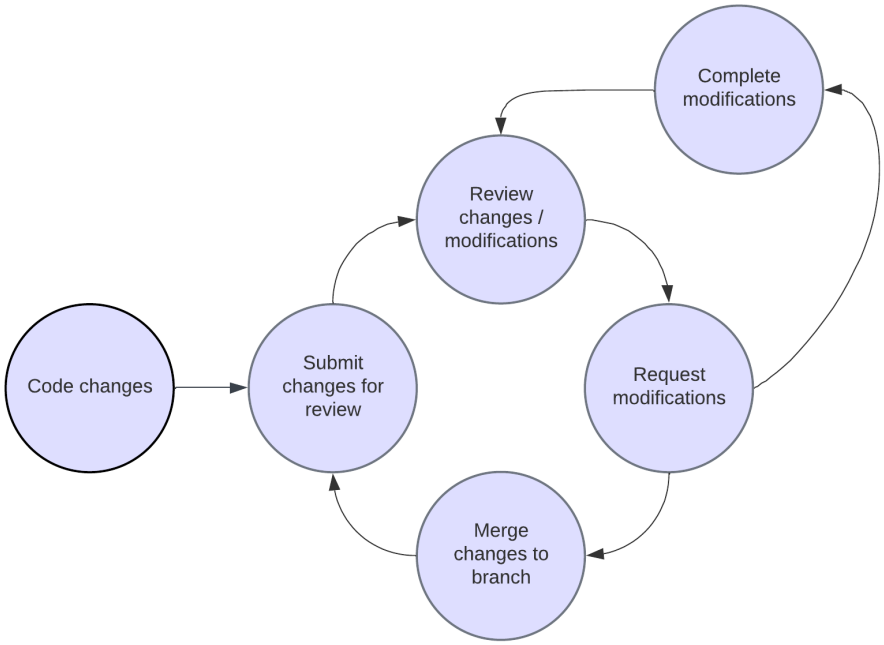

Figure 2.8: The merge request process.

The process in Figure 2.8 describes the basic MR cycle. When a set of changes is ready, a software engineer submits them for review through an MR. One or more other software engineers review the changes and request modifications if deemed necessary. When the modifications have been completed by the submitter, the modifications are reviewed. If there are issues with the modifications or some new issues are found, the request-complete-review cycle is repeated. Once there are no more modification requests the changes are merged into the target branch and the entire MR cycle is repeated. The MR cycle is simple and often the basic implementation between different teams is similar. Usually the review process is shallow and aims to catch obvious mistakes.

Merge requests promote traceability as code changes may be traced back to a specific MR, which in turn usually relates to a specific issue. Through the use of MRs, discussion about a particular set of changes is kept in a centralised place instead of being spread over e.g. instant messages and emails.

2.2.6 Semi-Automated Process Practices

A semi-automated process refers to a process which is initially established (e.g. by configuring a tool) and thereafter applied continuously while editing source code. These processes are manual by default and must be explicitly invoked to apply changes to code, however they may be triggered by e.g. saving a file or a keyboard key bind. This thesis focuses on two semi-automatic tools that are used to analyse code and to suggest changes to it: formatters and linters. These were chosen because they are considered to be fundamental in modern software development, and are likely to be impactful on quality. There are many commonly utilised static analysis tools that focus e.g. on information security and dependency analysis, however such tools were not considered in this thesis because they are less likely to have a direct impact on the quality characteristics that were chosen in this thesis.

Formatters reformat code according to a given set of rules such as maximum print width, spacing and line breaks. The rules are often fairly simple and focus only on code style. The chosen rules often aim to improve readability while the formatter is used to automatically flag code that violates the rules of the configuration. Many formatters additionally support automatically fixing trivial rule violations. Automatic flagging and fixing reduces manual work required to make a given component match the project's code style.

Linters function similarly but serve a different purpose, which is primarily to analyse code and suggest changes which may aim to improve different aspects of code quality. Their purpose varies from one linter to the next, although often they aim to improve issues in code such as readability, information security, or execution performance. Linter configuration is often more complex than formatter configuration.

2.2.7 Automated Process Practices

Maintaining high quality in an SP is very difficult to achieve by manual means and processes. Software engineers are only able to focus on a few things at best at once and they are bound to make mistakes. Various automated processes help maintain quality in addition to reducing the likelihood of causing regression in functionality while making changes to existing code. Some of the most well known automated processes are automated testing and static analysis tools, which are the focus of this section of this thesis. These automated processes (and most semi-automated processes) may be considered to be building blocks of continuous integration (CI), which is often associated with continuous delivery (CD) in the commonly used abbreviation "CI/CD". Due to the focus of this thesis, software development practices related to continuous delivery are excluded. This thesis focuses particularly on practices relating to how automated tests should be written, and on choosing automated process tools to aid in improving and maintaining code quality.

Static analysis tools are used to analyse the source code of an SP without actually executing the code. Formatters and linters may be considered static analysis tools, however often the term refers specifically to tools that focus on security aspects of software such as unpatched vulnerabilities, outdated dependencies and other security flaws. Often static analysis tools are passive, meaning they run in the background without requiring manual invocation by the software engineer.

2.2.8 Continuous Learning Practices

Software engineering is a vast, dynamic field that changes rapidly and requires software engineers to learn continuously. The respective literature review Section 3.4.8 addresses practices that may be employed by software development teams to support continuous learning among software engineers. The practices mainly focus on using code reviews and pair programming as mediums for learning.

3 Literature Review: Software Project Quality

The majority of the concepts and terminology used in this chapter were explained in more depth in Chapter 2; however, the key concepts will be briefly reiterated here when necessary. The suggestions for software development practices (labelled as Development Practice Suggestions, DPS) that may have an impact on software quality will be identified in this chapter through a literature review. The impactfulness of each DPS will be further validated (or disproved) through a survey in Chapter 4 and a case study in Chapter 5.

3.1 Methodology

3.1.1 Literature Sources

The primary literature database that was used in this thesis is the University of Turku's (UTU) literature database Volter (https://utuvolter.fi). This tool was selected due to being recommended by UTU guidelines and the availability of versatile international research papers, conference proceedings and books. The literature database ResearchGate was used to find additional sources that the search through Volter did not yield sufficient results for. In addition to scientific literature, the well-known books "Clean Code" [4] and "Design Patterns" [16] were used to find

and evaluate software development practices, because these books are considered to be among the most often recommended and highly acclaimed sources for literature on software development practices. The results of the analysis of these sources is presented in Section 3.2.

3.1.2 Keywords and Search Criteria

The search criteria were split into two categories: those used to find literature on software quality standards and models, and those used to find literature on software development practices. The purpose of the first category is to find out if recent literature (published between 2020 and 2025) suggests utilising the same quality standards (or their derivatives) as older literature (published before 2020). ISO's standards are an example of quality standards and models that have been updated over the years – for example, the standard ISO/IEC 25010:2011 is derived from the earlier standard ISO/IEC 9126-1:2001. The literature from the first category was also used to find out if the frequency of suggesting to use quality standards overall has changed over time. The second category was used to find software development practices that may be impactful on software quality, and to ascertain their impactfulness if such data was available in the literature.

After conducting the literature review it turned out that most literature in one category also included suggestions applicable to the other category. Therefore, literature was analysed for both purposes regardless of the category for which it was originally intended.

Keywords

The final keywords and date filters used to search for literature were as follows:

Software Quality Standards:

- "software AND quality AND (model* OR standard*) AND (comparison OR

review)" in the title, abstract or full text, 2020-2025

- "software AND quality AND (model* OR standard*) AND (comparison OR review)" in the title, abstract or full text, -2019

Software Development Practices:

- "software AND quality AND practice* AND (efficacy OR effectiveness OR impact* OR review)" in the title, without a date filter
- "literature AND review AND software AND quality" in the title, abstract or full text, without a date filter
- "functional AND programming AND impact" in the title, abstract or full text, without a date filter
- "combining AND programming AND paradigms" in the title, abstract or full text, without a date filter

Search Criteria

The primary focus field or industry of the literature was evaluated based on the title and abstract. Literature relevant to the primary software project types (closed source, web-based application consulting and product development) was selected to ensure the contents of the literature are relevant to this thesis. For example, projects focusing on technologies such as blockchain development are too dissimilar to the kind of software development that this thesis focuses on. Furthermore, sources that focus on software development in a particular industry were excluded to avoid possible bias on any one industry. Whenever a given source focused on a particular subject, the source was included if the subject aligned with the scope and goals of this thesis, e.g. development methodology literature was not included because it is not in the scope of this thesis.

The full literature text was evaluated for quality. Due to the quantity of available literature, sources estimated to be of low quality or low scientific merit (through e.g. inadequate methodology) were excluded. Literature that focuses on developing new

practices or quality standards are excluded because this thesis focuses specifically on evaluating existing software development practices and quality standards.

The search was limited to literature written in English. When a particular author or organisation appeared multiple times in the resulting set of literature, the literature was further filtered to reduce duplicate sources from the same author or organisation to ensure the literature would be representative of the industry as a whole.

3.1.3 Limitations

Software development is a complex process, particularly as the size and complexity of the project increases. For that reason, the DPSs may not take into account less complex projects or smaller development teams as the negative effects of poor quality may not be as readily apparent as they are in more complex projects and larger development teams.

Because the primary focus of this thesis is on web-based application consulting and product development, the results are likely not directly applicable to software projects that do not fit this description. Various aspects of software development were not taken into account – such as software architecture and development methodologies – because they were excluded from the scope of this thesis.

3.1.4 Deriving Software Development Practice Suggestions

The scientific and non-scientific literature was reviewed to find suggestions for software development practices that may be impactful on quality. Each DPS was assigned quality characteristics that it may have an impact on. The quality characteristics were previously introduced in Section 2.1.

3.2 Literature Search Results

The resulting literature from the search in the previous section and the general findings from each source are presented in this section. The literature sources are divided into sections first by the source database, and secondly by the keywords used to find the given literature. The keywords and search criteria from the previous section are used for all the sources.

3.2.1 Non-Scientific Literature

The book Clean Code [4] is one of the most well-known and highly acclaimed software engineering books and was therefore selected as a non-scientific source for this thesis. Another well-known book, Design Patterns[16], was chosen as a source for the same reasons, however the role of was significantly less significant as this thesis does not focus on software architecture specifically. Clean Code [4] was thoroughly read and analysed – the DPSs in this thesis were partially derived from suggestions in it. The scientific literature was used to supplement the baseline and to verify that scientific literature agreed with the derived DPSs.

3.2.2 Scientific Literature

The literature databases Volter and ResearchGate were used to search for scientific literature sources for this thesis. As mentioned in Section 3.2.1, these sources were used to supplement the baseline suggestions derived from Clean Code [4] and to validate the baseline suggestions. Particularly, quality standards were not mentioned in Clean Code [4], however the use of quality standards and models was heavily present in scientific literature. Clean Code [4] also does not directly address programming paradigms, which were often present in scientific literature.

The following sections reiterate the chosen keywords and the search results fil-

tered with the search criteria as was explained in Section 3.1.2. The results are presented in the same order as in the result of the search.

Volter

The UTU literature database Volter was chosen as the primary source of scientific literature. Originally the plan was to find sources for comparing the suggestions on quality models and suggestions on software development practices separately, however the found literature often mentions suggestions for both categories. Therefore, there is some overlap between the two categories in the literature review. The keywords and the resulting sources from them are:

- "software AND quality AND (model* OR standard*) AND (comparison OR review)" in the title, abstract or full text, 2020-2025. Used to find literature on quality standards and models that was published between 2020 and 2025. This search yielded 14,894 results, of which the first 20 results were considered. They were filtered down to three after combining articles from the same conference to the whole conference proceedings.
  - The proceedings of the PROFES conference titled "Product-Focused Software Process Improvement" (multiple articles as sources), 2023 [6]
  - The article "Matching terms of quality models and meta-models: toward a unified meta-model of OSS quality", 2023 [7]
  - The article "Software Product Quality Metrics: A Systematic Mapping Study", 2021 [8]
- "software AND quality AND (model* OR standard*) AND (comparison OR review)" in the title, abstract or full text, -2019. Used to find literature on quality standards and models that was published before 2020. The suggestions on quality standards in these sources were compared to the sources of the previous keywords to find out the previous sources suggested quality standards derived from the suggestions in these sources, and to compare how often quality standards were suggested overall. This search yielded 20,045 results, of which the first 30 results were considered. They were filtered down to four after combining articles from the same conference to the whole conference proceedings.

  - The proceedings of the ECSQ conference titled "Software Quality - ECSQ 2002" (multiple articles as sources), 2002 [12]
  - The proceedings of the conferences IWSM 2008, Metrikon 2008, and Mensura 2008 titled "Software Process and Product Measurement" (multiple articles as sources), 2008 [10]
  - The article "Exploring Quality Attributes Using Architectural Prototyping" in the proceedings of "Quality of Software Architectures and Software Quality", 2005 [11]
  - The book "The certified software quality engineer handbook", 2009 [9]

- "software AND quality AND practice* AND (efficacy OR effectiveness OR impact* OR review)" in the title, without a date filter. Used to find meta reviews on software quality literature and literature on software development practices, their efficacy and impact. This search yielded 19 results, which were filtered down to eight.
  - The article "An empirical study of the impact of modern code review practices on software quality", 2016 [23]
  - The article "The impact of software process improvement on quality: in theory and practice", 2003 [24]
  - The article "Omission of Quality Software Development Practices: A Systematic Literature Review", 2019 [2]
  - The article "Unraveling the code: an in-depth empirical study on the impact of development practices in auxiliary functions implementation", 2024 [5]
  - The article "Do code review practices impact design quality? A case study of the Qt, VTK, and ITK projects", 2015 [25]
  - The article "Investigating effectiveness and compliance to DevOps policies and practices for managing productivity and quality variability", 2024 [26]
  - The article "Impact of methods on productivity & quality: Panel: Impact of software engineering research on industrial practices", 2024 [1]
  - The book "Software Quality - Concepts and Practice", 2018. One of the book's chapters appeared in the search. [13]

ResearchGate

The literature database ResearchGate was primarily used to find literature on the topics of specific DPSs, for which literature did not exist (or was not found) in Volter. One meta-review article was also found on ResearchGate, which was used to ascertain if the findings of the literature review of this thesis are in line with similar research. The keywords and the resulting sources from them are:

- "literature AND review AND software AND quality" in the title, abstract or full text, without a date filter. Used to find reviews of the existing literature on software quality. This search yielded 100 results, of which the first 10 were considered. Only the most relevant one was selected because the results from searching in Volter yielded sufficient results.
  - The article "Review of Literature on Software Quality", 2018 [3]
- "functional AND programming AND impact" in the title, abstract or full text, without a date filter. Used to research the functional programming paradigm and the impact on code quality of using it or applying aspects of it with other paradigms. The search yielded 100 results, of which the first 10 were considered. Only the most relevant one was selected because the topic is not the focus of this thesis.
  - The article "An Overview of Practical Impacts of Functional Programming", 2017 [19]
- "combining AND programming AND paradigms" in the title, abstract or full text, without a date filter. Used to research the practice of combining aspects of programming paradigms after this suggestion was found in other literature. These keywords were used to further research the impact of functional programming specifically in combining multiple paradigms instead of using pure functional programming. The search yielded 100 results. Because this topic is not the primary focus of this thesis, only the following source was selected based on its title.
  - The article "A Perspective on Combining Different Programming Paradigms", 1995 [20]

3.3 Quality Standards

As was briefly mentioned in Section 2.1, most of the literature analysed in this thesis employed quality standards, even in literature which focus on other topics regarding software quality.

DPS 1.1. Establish a Quality Standard

A well documented project quality standard (PQS) refers to a quality standard that is selected for a particular project. A PQS should be selected early in the development life cycle of an SP. The PQS should be based on a formal definition of quality (e.g. ISO/IEC 25010:2023's quality model, optionally including the modifications made in this thesis). The PQS may be modified and extended to fit the needs of the development team and project. The PQS should be used primarily as a reference for making quality-related decisions. It could also be used as a checklist of sorts for software engineers to ensure that code considers all relevant aspects of quality.

This DPS is a general suggestion for utilising quality standards in software engineering, however since it does not directly impact any one characteristic of quality in particular, it is not assigned to any quality characteristic in this thesis.

DPS 1.2. Assign Responsibility for Quality

The structure of software engineering teams varies from organisation to organisation and often even between projects within an organisation, however there is always someone who is ultimately responsible for the project. In a traditional software engineering team the responsibility lies with the project manager, and in smaller teams the responsibility may lie with a particular software engineer or even a non-technical team member, which is problematic for quality. It has been suggested that someone in the team should be assigned responsibility for quality.[2] It should then be the person's responsibility to ensure all software engineers in the team are aware

of and apply e.g. the chosen software development practices. Literature suggests penalising personnel for not following the chosen practices. Such an approach may be warranted, albeit such an approach seems a bit excessive in the opinion of the author of this thesis. Nonetheless, some mechanism for ensuring conformance should probably be in place [2].

As with DPS 1.1., this DPS is a general suggestion for ensuring that someone is responsible for the quality of the project. It is therefore not assigned to any quality characteristic in this thesis.

3.4 Software Development Practices

3.4.1 Documentation

This section addresses various documentation practices, both for external documentation that is often written for the customer or end user of the SP, and internal documentation which often consists of information about the requirements, technical implementation details and maintenance instructions. Well-defined internal documentation is important to the maintainability of an SP because it enables persisting and sharing information efficiently. Persisted information eliminates the risk of a person forgetting any given piece of information or losing information when stakeholders leave the SP. Public documentation may be equally important depending on the specifics of the project in question, e.g. user instructions for a publicly available service may be beneficial both to the end user as well as the customer.

DPS 2.1. Write and Maintain Requirements Documents

The functional and non-functional requirements of an SP should be well documented as no-one is able to remember every detail of what is agreed upon [3]. Clearly documented requirements not only reduce the risk of lost work and knowledge but also

make development more efficient as software engineers are able to view the specific requirements independently at any time. It is also easier for software engineers to understand the requirements when they are well documented – "[a] lack of adequate understanding of requirements among software stakeholders [...] often leads to rework as developers make design assumptions that later need to be changed "[2]. It is not unusual for disagreements to arise during the development process in regards to what is included in the project scope. It may be beneficial to go over the requirements documentation with the customer to ensure both parties agree on the scope of the project, which also reduces the probability of major changes being made in the requirements during the development process [3].

This DPS is considered to be related to functional suitability and flexibility because requirements are fundamentally what define the intended needs of the project's users and non-functional requirements. It is also considered to be related to maintainability because modifying software without knowing its purpose is difficult, particularly when maintenance is performed by someone who is not intimately familiar with the project.

DPS 2.2. Write an Initial Project Plan Document

A rough estimate of the project schedule should be documented to ensure everyone involved in the project is aware of milestone deadlines [3]. The documented schedule is beneficial to the customer in that they are able to monitor the progress and to have an idea of how the development process is planned and executed.[13] An initial project plan gives everyone involved rough milestones as to when features should be ready and helps understand the development process overall.

It should be noted that the estimated schedule should be adjusted as necessary during development, especially when utilising iterative development methodologies, such as frameworks based on Agile. Accurate schedule estimation at the beginning

of a project is likely more difficult in traditional Waterfall projects.

In well-planned projects, the milestones laid out by the initial project plan document may help software engineers in designing maintainable software components by giving them an overview of the components to be developed and the relationships between them. This DPS is therefore considered to be related to maintainability.

DPS 2.3. Write and Maintain Specification Documents

One of the first steps in software development is requirements engineering, which refers to the process of creating technical specifications based on the functional and non-functional requirements of an SP. The resulting specifications should be well documented for similar reasons as the requirements documentation mentioned in DPS 2.1. The code implementation of features should align very closely with their respective specification documents.[3][2]

This DPS is considered to be related to functional suitability and flexibility because specifications define the functionality and non-functional requirements that should be implemented by software engineers. Assuming that the specifications are well made and implement the requirements of the projects accurately, they can be used to ensure that code is functionally concrete (and in some cases, functionally appropriate). It is also considered to be related to reliability, because specifications often also specify non-functional requirements – such as handling common faults. Finally, it is considered to be related to maintainability since specifications suggest (or outright define) the relationships between software components, inherently affecting characteristics of maintainability.

DPS 2.4. Write and Maintain Technical Design Documents

Technical design documents refer to documents that describe technical implementation, such as project architecture and the functionality of services. The technical

design documents of services should include e.g. its inputs, outputs and the major logic flow of the service. The practice of documenting architecture this way has been suggested in literature [2], and the practice of documenting services has been found to be impactful by the software development team which the author of this thesis is a member of (which shall be referred to as Author's Software Development Team, ASDT).

This DPS is considered to be related to functional suitability and flexibility because technical design documents describe the actual functional and non-functional implementation of software components. Technical design documents can be utilised by personnel other than software engineers, such as solution analysts, to ensure that the implementation implements the specifications – and, by extension, the requirements – of the project. This practice is also considered to be related to maintainability because technical design documents provide maintainers descriptions of the software components, their intended implementations and the relationships between software components.

DPS 2.5. Create Draft Design Documents

Draft design documents refer to documents that were used to design a particular feature of an SP. Their purpose is to document the thought process and explored ideas of the persons who designed the feature. Draft design documents should be free form and may consist of e.g. flow charts, notes and/or bullet points. They should not be updated over time because they do not describe an existing feature but rather the design process and decisions. This practice did not appear in the literature review, but has been found to be impactful by ASDT.

This DPS is considered to be related to functional suitability because draft design documents may depict the steps that were taken to arrive at the eventual solution for a given software component, possibly showcasing attempted solutions that were

rejected for some reason – such as not being functionally correct or appropriate. It is also considered to be related to maintainability for the same reasons.

DPS 2.6. Require at Least One Reviewer For Documents

The aforementioned documents should always be reviewed by someone other than the author [13]. The documents are not much use on their own if they lack necessary information, contain incorrect information or are e.g. difficult to understand by other persons than the author.

Requiring reviewing documents impacts the characteristics of quality that the documents themselves have an impact on. This DPS is therefore not considered to be related to any one quality characteristic.

3.4.2 Programming Paradigms

Because software architecture is outside the scope of this thesis, programming paradigms were not focused on either. However, it has been suggested in literature that SPs should adhere primarily to one paradigm but it may be beneficial to partially adopt practices from other paradigms. [20] This thesis considers the practice of combining parts of multiple paradigms as a way to potentially increase overall quality, however it does not address particular programming paradigms in great detail.

The main paradigms presented in Section 2.2.2 represent different philosophies and there are advantages and disadvantages to each of them

Functions may exhibit side effects, which means that the function changes non-local state or its output value is nondeterministic. In Clean Code [4], side effects are aptly described as being "lies"[4] that "often result in strange temporal couplings and dependencies"[4]. Pure functional programming prohibits side effects, and according to the conference paper by Abdullah Khanfor and Ye Yang, "some studies claim that writing programs using functional languages increases the security by enforcing the

programmers to a particular method of writing the code that mitigates security risks by forbidding the state changes in pure FP"[19]. The same is true for the functional correctness characteristic.

Some programming languages (such as Haskell) are purely functional by design, however pure functional programming is very restricting. Real-world software projects are rarely purely functional (excluding projects where the programming language itself is purely functional). It has been suggested that "the complexity and the cost of software design tasks using this paradigm plays a significant role in the lack of adoption in the industry". [19] This complexity is showcased in the figures of Section 2.2.2.

The functional paradigm may be implemented impurely, which means that aspects of functional programming are partially or fully ignored. A possible use case for the functional paradigm is to use it impurely in projects where the primary paradigm is something other than FP. In literature it has been said that "[P. Henderson] proposed a particular mixture of FP, formal specification, and rapid prototyping as an effective methodology for software design"[19]. Particularly functional programming concepts (e.g. monads) and practices such as avoiding mutability and side effects may be adopted with relative ease in projects where FP is not the primary paradigm [4].

Java is an example of a programming language where this practice has been adopted natively, to a degree. For example, the monad-esque class `java.util.Optional` (which strictly speaking breaks some rules of monads) may be used to manage nullable values to avoid `NullPointerException`s by wrapping the possibly nullable value in itself. The `Optional` variable can never be null, therefore accessing its properties is always safe whereas accessing the properties of a variable that has the value `null` is not (with certain exceptions). In addition to better null-safety, the monad provides additional methods for executing further code that will

be executed only if the wrapped value is not null, simplifying syntax and potentially reducing cognitive load when reading complex code.

```
class Node {
  private Object value;

  public Node(Object value) {
    this.value = value;
  }

  public Object getValue() {
    return this.value;
  }
}

// Handling nullable values with `java.util.Optional`.

Optional<Node> nullNodeOpt = Optional.ofNullable(null);

int nullNodeOptValue = nullNodeOpt
    .map(Node::getValue)
    .map(Integer.class::cast)
    .orElseThrow(() -> new IllegalArgumentException("Node is null"));

// Traditional handling of nullable values.

Node nullNode = null;

if (nullNode == null) {
  throw new IllegalArgumentException("Node is null");
}

int nullNodeValue = (int) nullNode.getValue();
```

Figure 3.1: Nullability handling with modern Java's `java.util.Optional`.

The Figure 3.1 illustrates the benefits of the monad `java.util.Optional` in Java. The first example could also simply use `.orElseThrow()` (without an argument) to throw a default exception, simplifying the code even further. Furthermore, the monad forces the programmer to handle nullability with e.g. `Optional#get()`, `Optional#orElse()` or `Optional#orElseThrow()` – otherwise the type of the variable is incorrect and would lead to a compile-time error. On the contrary, if the `if`-clause of the second example was to be forgotten, the program would throw a `NullPointerException` at runtime.

Many of the design patterns presented in Design Patterns are OOP-specific and

can not be easily adopted in SPs where other paradigms are used, however adopting some design patterns, particularly behavioural patterns such as the "template method" pattern, may be beneficial in certain situations if the chosen programming language supports it. [18][16][20] An example of where this pattern may be useful is generic classes that share structure, e.g. automations with methods for starting, querying and stopping the automation.

When multiple paradigms are partially combined, compatibility between the paradigms should also be ensured such that the chosen secondary practices do not contradict the primary paradigm. It should be noted that practices in OOP often require mutation, however it is possible to use non-mutable code alongside mutable code. It has been suggested that variables should be immutable by default and where mutability is required, the name of the variable should be e.g. prefixed with "mutable" to clearly denote its mutability [4], especially in programming languages where the mutability of variables cannot always be guaranteed (such as object instances in Java). An exception to the suggestion of combining multiple paradigms is that purely functional SPs should adhere solely to the functional paradigm as the other paradigms directly contradict the rules of pure FP, and purely functional code differs greatly from procedural- or object-oriented code.

DPS 3.1. Utilise Conventions From Other Paradigms

Based on the literature review it appears that it may be useful to partially combine secondary paradigms with the primary paradigm of the project, particularly concepts of functional programming seem to promote generally positive software development practices even when primarily using another paradigm.[20]

It is difficult to assign this DPS to any one particular quality characteristic, because the effects of utilising it vary depending on which particular conventions are utilised from other paradigms. For the sake of clarity, this DPS is not considered to

be related to any one particular quality characteristic in this thesis.

DPS 3.2. Avoid Mutability and Side Effects

Separate from DPS 3.1., the practice of avoiding mutability and side effects may be particularly impactful because it has been suggested in literature separately from functional programming, and because by the author's experience mutability and side effects indeed often cause hard-to-debug problems and readability issues. The logic flow of such code is also often very difficult to follow.[4][19]

This DPS is considered to be related to reliability and maintainability because mutability and side effects may cause unintended changes to program state, and because they make code harder to interpret, thus increasing the likelihood of introducing bugs when creating of modifying code which induces mutability or side effects.

DPS 3.3. Lexically Encode Mutable Variable and Method Names

When using a programming language that does not restrict variable mutability by default, such as Java, it has been suggested that mutable variables names and the names of functions which mutate variables or have side effects should be lexically encoded by e.g. prefixing the variable or function name with `mutable`. An example of using this practice would be to change a hypothetical mutable variable `counter`'s name to `mutableCounter`.[4] The author of this thesis has also found this practice to be helpful, although perhaps not altogether impactful.

This DPS is considered to be related to reliability and maintainability because lexical encoding may reduce the likelihood of introducing bugs when creating or modifying code that induces mutability or side effects.

3.4.3 Coding Convention Practices

DPS 4.1. Avoid the Use of Null Values

When a programming language implements null values, their usage should be avoided and some other value should be used as the default.[4] Tony Hoare, the inventor of null-pointer references, has said that they "may be perhaps a billion dollar mistake"[27]. They could be replaced by monadic values such as an instance of Java's `java.util.Optional` or a sensible default, such as `0` for integer type variables.

This DPS is considered to be related to reliability because avoiding null values may reduce the likelihood of mishandling variables that may be null, potentially causing problems in fault handling. It is also considered to be related to maintainability because it is likely easier to modify code that does (generally) not allow null values – otherwise, the author would need to always check whether something may be null or not. This is particularly time-consuming and error-prone when dealing with complex logic chains.

DPS 4.2. Use English in Written Technical Disciplines

English is the de-facto language when it comes to software development. Being able to read and write English is essentially mandatory for software engineers regardless of country due to e.g. all major programming languages using English keywords. In countries where English is not the primary language, some software projects have opted to use their native language for e.g. variable names and comments in code, which inevitably leads to very unreadable and messy code, particularly for anyone for whom the language is not their native language. Languages with special characters are particularly troublesome as programming languages may not even support those special characters at all.

Because of these reasons, English should be the primary language in code, including code comments and anything else in close proximity to code (such as technical

documentation that is included in version control) [4]. By extension it makes sense to primarily use English for all technical disciplines, such as documentation and the names of infrastructure components. However, at times exceptions to this rule must be made. For example, the customer of the project may require the use of a particular language in certain situations, such as customer-facing documentation.

This DPS is considered to be related to maintainability because it is easier to read text – and by extension, code – that is written in one language rather than a mix of languages. Furthermore, it may be impossible, or at the very least difficult, to modify code for someone who is not a native speaker of the the other languages.

DPS 4.3. Establish and Enforce Code Style

Clean Code [4] emphasises the readability of source code, particularly for "permanent" code such as test- and production code, although the readability of "temporary" code is also emphasised as being important, but to a lesser degree. Readability is critical for the modifiability of an SP – "You cannot write code if you cannot read the surrounding code"[4]. The functionality and intent of a component should be readily apparent from reading the code. Outside context and documentation should not be required to understand how any given component is structured and how it functions. Analysability requires readability, since it is either difficult or practically impossible to analyse a component if it is messy or convoluted, especially if the original author of the component is no longer present. It is suggested (through literature and the author's personal experience) to prioritise readability except in circumstances where e.g. execution performance is critical for the particular component and more readable code is not performant enough.[4]

The following stylistic suggestions do not warrant their own DPSs, however they were suggested in literature (where cited) and/or were found to be impactful by ASDT.

1. Use established style guides, such as Google's style guides, to avoid code style opinion conflict
2. Group variables by their purpose or category and separate distinct groups from one another with whitespace [4]
3. Utilise whitespace to improve readability of dense code
4. Utilise expressive naming so that the intent is clear to other software engineers [4]
5. Organise code such that minimal jumping around is necessary. [4]

This DPS is considered to be related to maintainability because well-styled code is easy to read – and interpret, given that the code itself is not particularly complex – enabling persons other than the original author to understand and modify the code more easily. Moreover, unrelated to quality characteristics directly, readable and consistent code is also easier to review, thus likely improving the efficacy of merge request reviews and code reviews.

DPS 4.4. Refactor Existing Code During Feature Development

Because the development of SPs generally takes a long time, existing code should be refactored to meet the quality standards whenever changes are made or new code is added to an existing file. The Boy Scout of America's rule "Leave the campground cleaner than you found it"[4] is often used as an analogy for this practice of continuous improvement. Refactoring should however be limited in scope and small improvements are likely enough to help avoid the code base "rotting".[4] This practice may be generally impactful, however it should be noted that unrelated changes in merge requests may prove harmful during reviews, especially if the refactoring is not atomic. ASDT has found that it may be beneficial to further separate pure refactoring into distinct branches to reduce noise in MRs, such as renaming a class separately from the rest of the code changes – however there comes a point where the refactoring is too small to merit a separate MR.

This DPS is considered to be related to reliability, maintainability, and flexibility because constantly improving code quality through continuous refactoring may reduce complexity – thus likely reducing likelihood of bugs (faults) – and may improve all sub-characteristics of maintainability and flexibility, depending on how code is refactored.

DPS 4.5. Use POSIX Timestamps for Point-In-Time Variables

POSIX timestamps are also known as e.g. Unix or Epoch timestamps. They refer to variables that indicate how many milliseconds (or nanoseconds) has elapsed since 1.1.1970. In the opinion of the author of this thesis, POSIX timestamps should be favored over data types that are internally implemented with some other technique than a POSIX timestamp, because such variables deal with date and possibly time values which are easy to make mistakes with, particularly if time zones are also used. This only applies to variables that are point-in-time by nature – that is, e.g. a time-zone-less simple date without time information could be stored as a `java.time.LocalDate`. It is probable that a majority of these kinds of variables should use POSIX timestamps instead.

Note that this suggestion only applies to backend code – frontend code should format dates as it sees fit.

This DPS is considered to be related to reliability because temporal variables that utilise some other strategy for handling dates and times are likely more error-prone, particularly when modifying dates and times, or when enacting comparisons between multiple temporal variables.

DPS 4.6. Create Variables for Magic Numbers

There is often a need to supply static number-type arguments to methods – for example, when defining the page size in a list request to the backend. Instead

of supplying these arguments directly in method calls, it has been suggested that intermediate variables be created for them in order to be able to give the number a meaningful name.[4]

```
// In class A.java
public List<T> getItems(int pageSize) { ... }

// In class B.java
final T items = getItems(1000);
```

Figure 3.2: Magic number example.

While editing the class `B.class` (Figure 3.2), the number argument `1000` could mean multiple things, such as the page size, a timeout in seconds or a timeout in milliseconds. The argument itself does not give any information to the reader. The argument can be assigned a meaning by creating a variable with a descriptive name:

```
// In class A.java
public List<T> getItems(int pageSize) { ... }

// In class B.java
private static final int PAGE_SIZE = 1000;
final T items = getItems(PAGE_SIZE);
```

Figure 3.3: Magic number example with an intermediate variable.

The benefits of this approach are more clear when there are multiple parameters, or even multiple parameters of the same type. It should be noted that while some integrated development environments (IDE) automatically display the parameter's name inline, this approach also makes reviewing the code outside of a development environment easier.

This DPS is considered to be related to maintainability because separate variables are easier and faster to understand than magic numbers, particularly for someone who is not intimately familiar with the project.

Code Comments

Code comments are subject to change just as the code itself is. Therefore, they should be kept to a minimum and as succinct as possible to make editing them easier.[4] However, they may be utilised in a few different ways that are explored in this section.

DPS 4.7. Use Code Comments to Explain Critical Code

It may be beneficial to write comments that warn future readers that making changes to logic may cause issues, e.g. when a given piece of code is critical for functionality and it is not readily apparent when reading the code.[4] An example of such code may be e.g. a particular variable type or algorithm that was chosen for execution performance reasons.

This DPS is considered to be related to maintainability because such comments may prevent someone from changing critical code without understanding the implications of the change. Additionally, such comments make the code more readable, particularly when the code is convoluted by necessity (for example, the code must be implemented in a way that is not very readable because of performance considerations).

DPS 4.8. Explain Regex Patterns With Code Comments

Complex Regex patterns are difficult to read, even for experienced software engineers. Explaining the pattern's meaning and logic with code comments has been found to be impactful by ASDT.

This DPS is considered to be related to maintainability because such comments make the patterns much easier to understand, thus saving time in figuring out what some software component does. Moreover, such comments make it easier to modify the given pattern.

DPS 4.9. Avoid Leaving "TODO" Code Comments

Code comments that express a need for changes are often referred to as "TODO" comments. Code bases are often littered with todo-comments because problems may be discovered during activities unrelated to the specific code. The use of todo-comments is controversial in that they may be aid in remembering to make some change, however often they are either forgotten or time is not allocated for making the changes. Therefore, it may be beneficial to strive to make the required changes immediately if the changes are relatively small in scope. Otherwise it may be better to create new tickets for them instead of leaving todo-comments, as they may easily be forgotten.[4][26] On the other hand they may be helpful in marking a feature as incomplete so that other software engineers do not have to spend time looking up if a ticket has already been created for the problem.

This DPS is considered to be related to functional suitability, reliability and maintainability because it may be difficult to ascertain the severity of shortcomings in "TODO"-commented code if the code base is littered with "TODO" comments. Therefore, known shortcomings may impact the aforementioned quality characteristics of the given project.

Artificial Intelligence Tools

Artificial intelligence (AI) tools have exploded in capability and popularity in recent years. Because generative AI models such as OpenAI's ChatGPT and GitHub's Copilot are so new, not much is mentioned about them in the literature that was searched for in this thesis. The following suggestion is based solely on the subjective experience of the author.

DPS 4.10. Review Code Generated By AI Tools Exceptionally Carefully

Because current generative AI tools are large language models (LLM), they do not posses a capability to think. They merely predict what text the user is expecting of them. They have been used to generate code and the author of this thesis also uses them to reduce manual typing by accepting their suggestions if the code is the same as they would have written anyway. Others rely on generative AI tools far more, to generate code or to even skip thinking entirely. Such code should be reviewed exceptionally carefully, because LLMs are bound to make mistakes and they do not actually understand what they are outputting.

This DPS is considered to be related to all four quality characteristics because AI tools may generate almost any kind of code, thus impacting all characteristics of quality in a given project.

3.4.4 Version Control Practices

The following suggestions are based solely on the subjective experience of the members of ASDT. Because Git is the de-facto VCS at the timing of writing, this section only applies if Git is used.

DPS 5.1. Establish a Branching Strategy

A branching strategy dictates how branches should be created and how they should be merged. When developing new features, the feature branches should be merged to a development branch, usually `develop`. It is advisable to keep the changes of a particular branch highly concentrated and minimal because large branches make managing changes cumbersome. If a feature requires substantial changes, it should be broken down into smaller features and thus branches which are merged to a common feature trunk. A feature trunk refers to a git branch that serves as an intermediate target branch for a particular feature that consists of multiple smaller

features. This way the feature may be developed incrementally without releasing an unfinished branch to the development branch. Usually the development branch is eventually merged to the main branch, usually called `master` or `main`.

This DPS is considered to be related to reliability – in particular, recoverability – because the branching strategy directly impacts how a certain state of the software project may be restored. It is also considered to be related to maintainability because a consistent branching strategy makes it easier to trace changes in code later – given that the chosen merging strategy preserves information about the branching strategy.

DPS 5.2. Establish a Branch Naming Strategy

Branches should follow a well-defined naming convention to make navigating the Git history fluid and to improve reliability in connecting a given set of code changes to a particular ticket. GitLab, a Git repository platform [28], suggests that branches should only consist of numbers, hyphens, underscores and lowercase letters from the ASCII standard table. GitLab also suggests that forward slashes and emojis may be used in branch names, however they may not be compatible with all software packages and their usage is thus discouraged [28]. ASDT has settled on a naming convention, which is used as a basis for the following suggestions.

The branch name should always be prefixed with the ticket number that the branch relates to [28]. In a situation where a ticket does not exist, the ticket number may be omitted, however it is always encouraged to have every branch correspond to a ticket. It is encouraged to create a new ticket when changes must be made to a ticket whose changes were already merged. However if the same ticket is reused, the branch name should be suffixed with the "index" of the branch after the ticket number, for example `31165-1`. Branch indexing aims to ensure that the Git history never contains two or more merge commits with an identical ticket number.

Branch hierarchy may be formed by prefixing the branch name with a broad category followed by a forward slash to easily separate e.g. features from hotfixes. Some Git repository platforms, such as Azure DevOps, automatically categorise branches in their user interface by their hierarchical prefix, however most repository platforms do not offer such functionality. Branch hierarchy may be beneficial especially in later stages of development due to intertwining of feature development and fixes, however the impact on code quality is likely not significant.

This DPS is considered to be related to reliability – in particular, recoverability – because a consistent and informative branch naming strategy directly impacts how a certain state of the software project may be restored. It is also considered to be related to maintainability because a consistent and informative branch naming strategy makes it easier to trace changes in code later – given that the chosen merging strategy preserves information about the branch names.

DPS 5.3. Establish a Commit Message Format

In addition to branch naming, the commit message format may affect traceability and comprehension of the changes by other software engineers. Commit messages should convey why the changes were made and what the primary change is. When using such a format, the high-level contents of a commit may be inferred from the message, finding a specific commit is easier and the commit history self-describes the development progression.

ASDT has settled on a message format, which is used as a basis for the following suggestions.

The commit message should start with the ticket number followed by the main change formatted as an order to do something, e.g. `31165 Implement REST interface` or `52802 Fix file upload handling`. In some implementations of this format the ticket number is surrounded by parenthesis to emphasise the ticket number. The

specific format chosen likely has limited impact on code quality, however it may be beneficial to include the ticket number so that changes are easier to trace to a particular ticket, and the message should be informative in order to understand the contents of the commit at a glance.

As the aforementioned commit message format intrinsically implies, each commit should consist of a limited set of changes all relating to a single actionable subject. This way even larger issues may be divided into small, distinct "packets" of changes, which may be beneficial if a ticket is unintentionally too broad or cannot be easily split into smaller tickets. Most merge request tools allow viewing changes commit-by-commit, which reduces cognitive load and fatigue in reviewing changes of a broad set of changes. Improving the reviewer's experience indirectly improves code quality as the reviewer is less likely to glance over the changes to get the review over with and more likely to find problems and make suggestions as a result.

This DPS is considered to be related to reliability – in particular, recoverability – because a consistent and informative commit message format directly impacts how a certain state of the software project may be restored. It is also considered to be related to maintainability because a consistent and informative commit message format makes it easier to trace changes in code later – given that the chosen merging strategy preserves information about the commit messages.

DPS 5.4. Establish a Merging Strategy

The commit history of an SP may be managed in various ways. Some software engineers may prefer to include all commits in the history while others may prefer to squash commits to keep the history clean. Some may prefer merge commits while others prefer rebasing. As this topic is highly opinion-based, it is not discussed in this thesis. However it should be noted that regardless of specific preference in commit history management, it is likely that the commit history is readable and

traceable enough if the above recommended branch naming and commit message formatting practices are established. The most important thing is that the chosen practices are used throughout the project.

This DPS is considered to be related to reliability – in particular, recoverability – because the merging strategy directly impacts how a certain state of the software project may be restored. It is also considered to be related to maintainability because being able to restore previous versions of code is an essential tool in software maintenance.

3.4.5 Merge Request Review Practices

DPS 6.1. Establish a Code Review Process

The merge request review (merge review) process is often informal and lax, but it may also be implemented more stringently to improve quality through adherence to the PQS. Improving quality as early as possible is important because mistakes and technical debt propagates as the project evolves [29]. Code that depends on low-quality code may also end up being of low quality because it has to work around the flaws of the low-quality code. This reduces overall code quality and increases project costs – "Unsuccessful development is mainly attributed to the fact that by the time problems are identified, it is too late to rectify them." [3]. Merge requests are one of the first steps in improving quality during the development of an SP and are therefore essential in improving quality.

This DPS is considered to be related to all four quality characteristics because unreviewed or poorly reviewed code may impact all characteristics of quality in a given project.

The Focus of Merge Request Reviews

DPS 6.2. Consider Higher Level Design Issues in Code Reviews

Most merge review tools display changes within its context, i.e. a few lines of code above and below the changed lines of code. Often code reviews focus mostly on stylistic issues or glaring mistakes, however it has been suggested that higher-level code architecture and design issues should be focused on during code reviews.[23]

This DPS is considered to be related to functional suitability, reliability and maintainability because higher level design issues impact all three quality characteristics of a given project.

DPS 6.3. Require at Least One Reviewer For All Code Changes

"By submitting a review request, the original author already believes that the code is ready for integration. Hence, changes that are only approved by the original author have essentially not been reviewed."[23] Therefore, all code changes should be subject to review by at least one other reviewer. Higher levels of code review coverage and reviewer seniority have also been linked to reducing anti-patterns and other negative effects in code [25].

This DPS is considered to be related to all four quality characteristics because unreviewed code may impact all characteristics of quality in a given project.

3.4.6 Semi-Automated Process Practices

DPS 7.1. Establish Formatter and Linter Tools

Because the purpose of formatters is to point out and fix code style, using a single formatter is often preferred to avoid conflicts between different formatters. There are situations where multiple formatters may be used, e.g. the formatter/linter ESLint is fairly often paired with Prettier, which only functions as a formatter. In such cases it is important to ensure the configurations of the tools are not in conflict with

one another and that they do not flag issues that are already flagged by another tool. It should be noted that multiple complex processes running semi-automatically may cause performance issues on the development device as every tool analyses code independently.

Both formatters and linters often offer the ability to automatically apply changes to the source code. Automatic applying reduces manual work, however using automatic applying should be carefully considered because it may interfere in the writing process, e.g. when a rule automatically removes unused code but the component is not yet ready and the unused code is in truth needed later. Automatic applying is often possible to enable or disable on a rule by rule basis. In the experience of the author of this thesis, it is often best to disable automatic fixing by default and enable it for certain rules after careful consideration.

This DPS is considered to be related to maintainability for the same reasons that were mentioned with respect to code readability in DPS 4.3. (see Section 3.4.3).

DPS 7.2. Establish Static Analysis Tools

Static analysis tools often point out issues in code. Various static analysis tools focus on different aspects of code, however such tools are commonly used to improve code quality and to point out information security issues. In the projects of Evitec Solutions, the static analysis tool Sonar is frequently used and has proven to be effective in improving overall code quality. In SPs where the programming language C# is used, the static analysis tool ReSharper is also used.

This DPS is considered to be related to functional suitability and reliability because static analysis tools notify of issues related to these quality characteristics. It is also considered to be related to maintainability because static analysis tools may notify of issues related to maintainability, and because they are as useful when maintaining code as they were when initially developing the project.

DPS 7.3. Centralise Configurations of Tools

Regardless of the chosen tools, it is important to ensure that all team members use identical configurations at all times. Tools such as ESLint save their configuration in files, which should be included in version control. Tools such as SonarLint use configuration files but also allow customising behaviour through user-defined settings that are usually not easily included in VC. In such a case the agreed upon configuration should be managed centrally and it should be ensured all team members update their configuration whenever changes are made to it.

This DPS is considered to be related to maintainability because centralising configurations ensures that all contributors use the same configurations, and because making changes to the configurations is faster and less error-prone than when configurations are user-specific.

3.4.7 Automated Process Practices

DPS 8.1. Run Semi-Automatic Tools as Part of CI/CD

Semi-automated processes may also be integrated to be an automated part of the merge request process and/or the continuous integration and continuous delivery process to ensure all committed code has been handled by all the established tools [26]. The downside to automatically running tools is the additional time required to run them and the cost of computing resources. Historically the execution performance of some tools, particularly linters and formatters, on a work station have been poor due to limited computing power of laptop computers and thus they have not been very popular in the past. Technology has improved dramatically in the past decade and the execution performance is usually no longer a concern on modern hardware.

This DPS is considered to be related to functional suitability, reliability and

maintainability because running the semi-automatic tools as part of CI/CD automatically ensures that they are utilised before code changes are merged, and that they are consistently run.

DPS 8.2. Run Automated Tests Periodically

In order to avoid regressions in existing code, automated tests should be run periodically. This should include most tests, as the test run can be scheduled to run when software engineers are not actively working on code – for example, at night.[26]

This DPS is considered to be related to functional suitability, reliability and maintainability because automated tests themselves may impact these characteristics of quality in a given project (depending the test coverage), and running them periodically ensures that the tests are actually being run.

Writing Automated Tests

Code cannot be considered high quality if its functional correctness can not be verified. Therefore, high quality code must be accompanied by manual or automated tests. Furthermore continuous improvement of code is much more likely to occur when the code is well tested, because the software engineer may feel confident in making changes without risk of breaking things.[4] Because testing is a vast subject in of itself, this thesis does not address testing in itself, but rather how code in automated tests should be written.

DPS 8.3. Write Tests Like Production Code

Code quality matters not only in production code but also in tests, because low-quality tests are hard to reason about and change. This is particularly important in tests that change often, such as unit tests. Having poorly written tests may be equal to, or even worse than, having no tests at all [4]. This is because it may be difficult to figure out what the test is testing, which may lead to gaps in testing

coverage, resulting in a false sense of security. Poorly written tests are also harder and more tedious to update and may therefore end up being left in a state where they no longer pass or may be discarded entirely [4].

This DPS is considered to be related to maintainability for the same reasons that were mentioned with respect to code readability in DPS 4.3. (see Section 3.4.3). Moreover, readable and well-structured tests are more likely to be kept up to date than poorly written tests, which may end up being scrapped or ignored due to difficulty in keeping them up to date.

DPS 8.4. Write Unit Tests With the Build-Operate-Check Pattern

As was established in DPS 8.3., code quality is important in tests as well. One way to improve it is to use the build-operate-check pattern to separate different concerns within a testing method. When using this pattern, the testing method should start with building any necessary variables, followed by operating (calling) the code and end with checking the results of operating the code.[4] The three groups should be separated by whitespace, similar to what was suggested in DPS 4.3.

This DPS is considered to be related to maintainability for the same reasons that were mentioned in DPS 8.3. (see Section 3.4.7).

3.4.8 Continuous Learning Practices

DPS 9.1. Conduct Pair Programming

Pair programming is a process in which two or more software engineers work together on a given component synchronously. It is often used as an educational tool whereby a software engineer learns from a more experienced software engineer. The benefit of leveraging this process as an educational tool is that coding choices and related information may be discussed synchronously while writing code whereas usually the act of writing code occurs in relative isolation and separate reviews are conducted

after the code has been written. Due to the distracting nature of the process it may not be efficient in terms of writing velocity, however the efficiency cannot be directly compared to normal circumstances due to the educational value of the process.[5]

The process may also be useful for brainstorming wherein two or more software engineers figure out which approach should be taken for a given component. Utilising the process in such a way may yield efficiency in decision making and better decisions being made overall. The process may be used to multiply the velocity of writing code if the software engineers are experienced enough to be able to efficiently write code synchronously with another software engineer while also maintaining consistency and quality overall.[5]

Literature suggests that pair programming is likely more effective as a learning tool for less experienced software engineers as it is between experienced engineers.[5]

This DPS is considered to be related to all four quality characteristics because pair programming may impact all quality characteristics of a given project when utilised by experienced software engineers for simultaneous software development. It is indirectly related to all four quality characteristics when utilised as a learning tool in that a higher skill level is likely correlated with positive impact on all quality characteristics.

DPS 9.2. Conduct Retrospective Code Reviews

Merge request reviews are often asynchronous in nature as the participants usually do not complete the review through a meeting. In addition to merge reviews, it may be beneficial to utilise code reviews, which refer to a process where the software development team discusses a given component during a synchronous meeting. The component may be e.g. a module of the SP or code changes related to one or more merge requests, however the subject should be narrow and the number of lines of code should be limited.

Code reviews should be primarily used as an educational tool by discussing why certain choices were made during the development or the merge review of the topic component [30]. The choices discussed in a code review may be e.g. architectural patterns, design patterns, code style, implementation details or modifications requested to the original changes. Discussion about modification requests may reduce how often a given problem reappears in later merge reviews while the other topics aim to be generally educative [29]. Additionally, code reviews may be useful in team members absorbing the chosen software development practices and code style [29]. Continuous learning by members of the software development team may improve code quality in the long run and additionally new issues may be found during code reviews.

This DPS is considered to be related to all four quality characteristics because additional code reviews may positively impact all characteristics of quality in a given project through finding issues in code after initial merge request reviews.

3.5 Summary

The following development practice suggestions were derived from literature sources or have been found to be impactful by the author of this thesis or the software development team which they are a member of.

DPS	**Source**	**Quality Attribute**
1.1 Establish a Quality Standard	Literature	None
1.2. Assign Responsibility for Quality	Literature	None
2.1. Write and Maintain Requirements Documents	Literature	Functional suitability, maintainability, flexibility
2.2. Write an Initial Project Plan Document	Literature	Maintainability
2.3. Write and Maintain Specification Documents	Literature	Functional suitability, maintainability, flexibility

DPS	**Source**	**Quality Attribute**
2.4. Write and Maintain Technical Design Documents	Literature	Functional suitability, maintainability, flexibility
2.5. Create Draft Design Documents	Author	Functional suitability, maintainability
2.6. Require at Least One Reviewer For Documents	Literature	None
3.1. Utilise Conventions From Other Paradigms	Literature	None
3.2. Avoid Mutability and Side Effects	Literature	Reliability, maintainability
3.3. Lexically Encode Mutable Variable and Method Names	Literature	Reliability, maintainability
4.1. Avoid the Use of Null Values	Literature	Reliability, maintainability
4.2. Use English in Written Technical Disciplines	Literature	Maintainability
4.3. Establish and Enforce Code Style	Literature	Maintainability
4.4. Refactor Existing Code During Feature Development	Literature	Reliability, maintainability, flexibility
4.5. Use POSIX Timestamps for Point-In-Time Variables	Author	Reliability
4.6. Create Variables for Magic Numbers	Literature	Maintainability
4.7. Use Code Comments to Explain Critical Code	Literature	Maintainability
4.8. Explain Regex Patterns With Code Comments	Author	Maintainability
4.9. Avoid Leaving "TODO" Code Comments	Literature	Functional suitability, reliability, maintainability
4.10. Review Code Generated By AI Tools Exceptionally Carefully	Author	Functional suitability, reliability, maintainability, flexibility
5.1. Establish a Branching Strategy	Author	Reliability, maintainability
5.2. Establish a Branch Naming Strategy	Author	Reliability, maintainability
5.3. Establish a Commit Message Format	Author	Reliability, maintainability
5.4. Establish a Merging Strategy	Author	Reliability, maintainability
6.1. Establish a Code Review Process	Literature	Functional suitability, reliability, maintainability, flexibility
6.2. Consider Higher Level Design Issues in Code Reviews	Literature	Functional suitability, reliability, maintainability
6.3. Require at Least One Reviewer For All Code Changes	Literature	Functional suitability, reliability, maintainability, flexibility
7.1. Establish Formatter and Linter Tools	Author	Maintainability

DPS	**Source**	**Quality Attribute**
7.2. Establish Static Analysis Tools	Author	Functional suitability, reliability, maintainability
7.3. Centralise Configurations of Tools	Author	Maintainability
8.1. Run Semi-Automatic Tools as Part of CI/CD	Literature	Functional suitability, reliability, maintainability
8.2. Run Automated Tests Periodically	Literature	Functional suitability, reliability, maintainability
8.3. Write Tests Like Production Code	Literature	Maintainability
8.4. Write Unit Tests With the Build-Operate-Check Pattern	Literature	Maintainability
9.1. Conduct Pair Programming	Literature	Functional suitability, reliability, maintainability, flexibility
9.2. Conduct Retrospective Code Reviews	Literature	Functional suitability, reliability, maintainability, flexibility

Table 3.1: The development practice suggestions that were derived from literature sources or have been found to be impactful by the author of this thesis or the software development team which they are a member of.

4 Quality Survey

The software development practices introduced in Chapter 3 were validated through a survey, which was also used to rank the DPSs by their impactfulness on quality. The survey is quantitative and targets an audience of professional software engineers who have participated in medium to high complexity projects. The survey was distributed within Evitec Solutions, on the LinkedIn, various contacts and software engineering groups. These specific channels were chosen in order to reduce the likelihood of receiving low-quality answers (e.g. students without work experience).

The following criteria were stated in the survey to limit the pool of respondents:

- "You are a professional software engineer"
- "You are, or have been, involved in the development of a software project of moderate (or higher) complexity as part of a software engineering team".

Moreover, the respondents were asked to answer whether they were a professional software engineer or not, and if they answered "No", the survey was terminated. This was done to minimise the chance that non-professional people answered the survey.

The survey was implemented with an online form to maximise the potential pool of respondents, because online forms are easy to distribute to the target audience. The full survey may be viewed in Appendix B. Please note that the PDF version of the survey has a different visual look than the online version that was distributed.

Within the two weeks between 4 April 2025 and 17 April 2025 that answers were collected, 44 answers were received.

4.1 Methodology

The respondents were asked to choose a project which they have been involved in the development of to base their answers concerning a specific project on. This was done to ensure all answers by a given respondent were based on the same project. This project will be referred to as "the selected project".

The respondents were asked about their current work experience in years out of pre-determined options. Similarly, they were asked about the currently elapsed length of the selected project in months or years out of pre-determined options. The answers to these questions were not directly used in the analysis. Rather, they were used to get an idea of what the respondents pool looked like.

To correlate the use of software development practices to quality, the survey consisted of two primary rating systems (RS):

- RS1: For each DPS, the respondents were asked whether the software development practice was used in the selected project. They were also asked to give an estimate as to what the level of quality was in the selected project on a scale of 1 (lowest) to 10 (highest) in their opinion, which was used to calculate the correlation factor of the software development practice to quality.
- RS2: The respondents were asked to give their opinion as to how high of an impact each development practice has on quality in general – not necessarily related to the selected project – on a scale of 1 (lowest) to 5 (highest).

All questions related to the use of specific DPSs or rating their impactfulness had the option to select "I don't know", in which case that answer was excluded from RS1 and RS2 analysis for that particular DPS. This option was offered so that respondents would rather pick one of those options instead of giving a random rating if they did not know what to answer or did not understand the question, to ensure that the scores are not skewed by such answers. Similarly, questions had the option to select "Not applicable" if the DPS was not applicable, such as when another VCS was used instead of Git.

The DPSs were assigned to the following high-level categories in the survey:

- Practices Related to Documentation
- Practices Related to Programming Paradigms
- Practices Related to Coding Conventions
- Practices Related to Version Control
- Practices Related to Merge Request Reviews
- Practices Related to Semi-Automated and Automated Processes.

Some of the identified DPSs were excluded from the survey in the interest of limiting the number of questions in the survey to increase the response rate. The number of questions in the survey was high even after excluding some of the DPSs. The excluded DPSs are presented in the following table.

DPS	**Exclusion Criteria**
DPS 1.1 Establish a Quality Standard	The author's estimate is that it is unlikely that quality standards are used this way widely
DPS 1.2. Assign Responsibility for Quality	In practice, the responsibility for quality is almost always assigned directly, or indirectly through assigning the responsibility for the project in general
DPS 3.1. Utilise Conventions From Other Paradigms	The author's estimate is that it is likely that respondents would not know what is meant by the DPS, without explaining the concept in length. The practice is also indirectly included in DPS 3.2., which was included in the survey

DPS	**Exclusion Criteria**
DPS 6.1. Establish a Code Review Process	Excluded from the survey by mistake
DPS 9.1. and DPS 9.2.	Excluded because these DPSs have more to do with continuous learning than project quality

Table 4.1: The DPSs that were excluded from the survey.

The responses to RS1 were used to calculate whether a given DPS correlated to a higher project quality score. The responses to RS2 questions were used to calculate the mean score for each DPS, which were directly used to evaluate their impact on quality.

4.2 Analysis

Spearman's correlation factor was used to calculate the level of correlation of each DPS to the project quality estimate (RS1) using the formula

$$\rho = \frac{\sum_{i=1}^{n}(R_{di} - R_{dm}) \cdot (R_{qi} - R_{qm})}{\sqrt{\sum_{i=1}^{n}(R_{di} - R_{dm})^2} \cdot \sqrt{\sum_{i=1}^{n}(R_{qi} - R_{qm})^2}} \tag{4.1}$$

where n is the number of applicable answers for the particular DPS, R_{dm} is the mean rank of the score for the particular DPS, R_{qm} is the mean rank of the quality scores, R_{di} is the rank of the ith response by score for the particular DPS and R_{qi} is the rank of the corresponding ith response's quality score.[31] This formula was used because ties between ranks exist in the data. Note that the formula is identical between Pearson's correlation coefficients and Spearman's correlation coefficients.

The following scheme was used to assign ranks to the DPS questions.

- 2 = Completely positive
- 1 = Partially positive
- 0 = Not applicable or answered "I don't know"
- -1 = Partially negative

- -2 = Completely negative

When a particular respondent answered that the question is not applicable to them or they do not know the answer (score of 0), that answer would be excluded from the rankings of that DPS question. The project quality scores were ranked individually for each question so that excluded answers do not also affect the ranking of the project quality scores. The probability of obtaining test results at least as extreme as the result actually observed (P-value) were calculated using the Student's t-distribution with a Student's t-test value that was calculated using the formula

$$t = \frac{\rho \cdot \sqrt{(n-2)}}{1-\rho^2} \tag{4.2}$$

where ρ is the Spearman's correlation factor for the particular DPS and n is the number of applicable answers for the particular DPS.[32] The DPSs were then sorted in ascending order by their Spearman's correlation factor ρ.

The impactfulness scores of each DPS were simply summed across responses (RS2) without normalisation, because analysing the impact of work experience and project duration to the responses is outside the scope of this thesis. The mean impact ratings were calculated with the formula

$$\bar{x}ImpactRating = \frac{1}{n} \cdot \sum_{i=1}^{n} y_i \tag{4.3}$$

where the symbol $\bar{x}$ represents arithmetic mean, `n` is the number of applicable responses and y_i are individual applicable responses.[33]

Additionally, the total mean impact rating of all DPSs was calculated using the formula

$$Global\ \bar{x}ImpactRating = \frac{1}{n} \cdot \sum_{i=1}^{n} y_i \tag{4.4}$$

where `n` is the number of DPSs and y_i are the individual mean impact ratings.[33]

4.3 Limitations

It is acknowledged that RS2 results are based on opinions, which means that the results do not definitely prove the level of impact of the DPSs. RS1 establishes a correlation between the use of DPSs and the project quality score. The correlations indicate that a relationship exists between the software development practices and quality, but they do not directly imply causation. Moreover, the quality of SPs depends on a myriad of other factors besides software development practices. It should be noted that the results of this survey depend on the interpretation of how the software development practices are utilised in the target projects, and on the opinions of the respondents with respect to the impact ratings and project quality scores. The survey responses would likely be different between different respondent pools.

It should be noted that many aspects of software engineering were excluded from the scope of this thesis and that the list of software development practices is not exhaustive – therefore there are probably other software development practices that are also impactful on quality besides the ones presented in this thesis.

The number of respondents to the survey is relatively low, however the responses are likely to be of very high quality because the survey was distributed among professional software engineers. Therefore it is reasonable to expect the results to be representative. On the other hand, many of the respondents work at Evitec Solutions – this could introduce a bias in how software development practices are utilised when compared to the wider industry because many of the respondents follow the same SDLC and general guidelines at Evitec Solutions.

The analysis of the responses was automated with a custom-written program, which leaves a possibility for mistakes to appear in the calculations. The source code of the analysis scripts is available online (see Appendix C.1).

4.4 Results

The survey responses are presented in this section, followed by the analysis results.

4.4.1 Analysis Source Data

The frequency of answers was analysed with respect to each question in the survey. The responses to survey questions with interesting answer frequencies are displayed through graphs in this section, depicting a subset of the source data that will be used in the analysis in the next section. The responses to the rest of the survey questions are available as graphs in Appendix A.

Background Information

A majority of the respondents were highly experienced with almost half of the respondents having more than 10 years of experience and a small minority having less than 2 years of experience – signifying that the survey distribution method was indeed successful (Figure 4.1). A possible explanation for the abnormally high experience levels is the companies and groups that the survey was specifically distributed to.

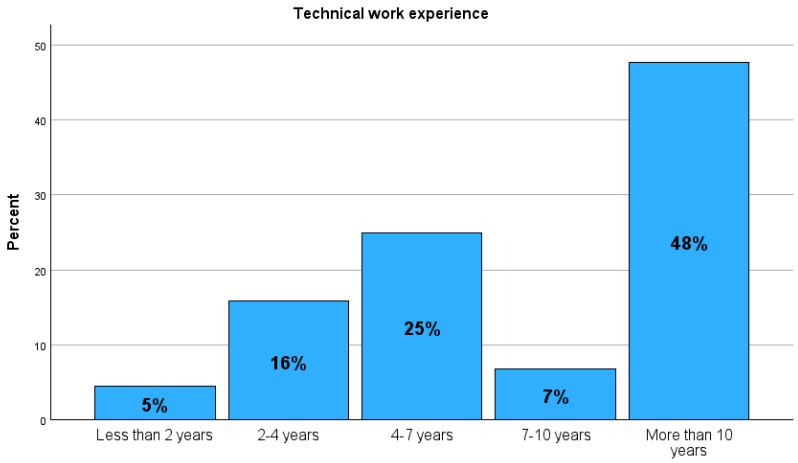

Figure 4.1: Technical Work Experience

Documentation Practices

Most projects utilised specification documents, however they were not maintained in 57% of projects (Figure 4.2). Project documents were not reviewed in only 11% of projects (Figure 4.3).

Initial project plan documents and draft design documents were considered to be significantly less impactful than other documentation practices, which were rated as very impactful (Figure 4.4). These results align with the expectations of the author of this thesis.

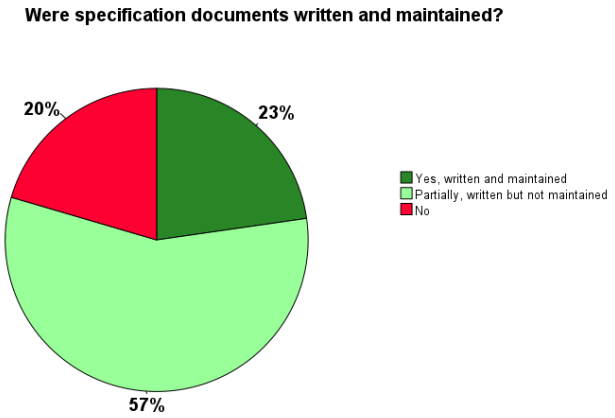

Figure 4.2: Specification Documentation (DPS 2.3.)

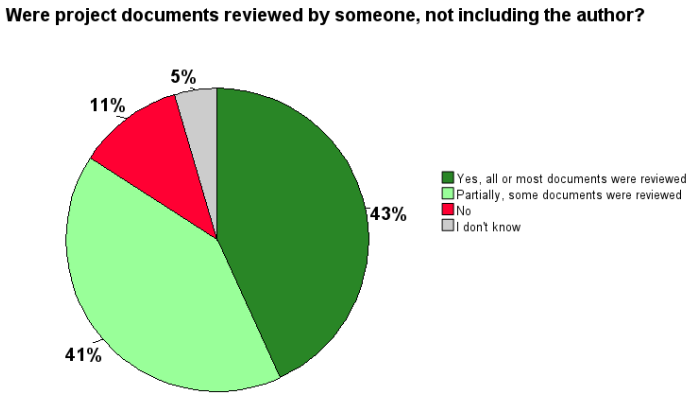

Figure 4.3: Documentation Reviewing (DPS 2.6.)

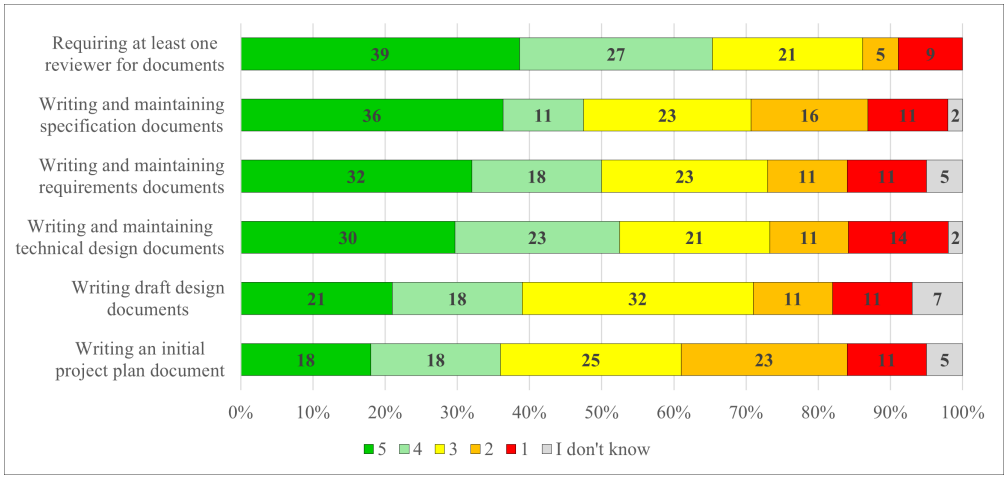

Figure 4.4: Documentation Practices Impact Ratings

Programming Paradigm Practices

Unexpectedly, almost all projects avoided mutability and side effects to some degree (Figure 4.5). The author of this thesis was under the expectation that this was not a widely utilised practice. Lexical encoding of mutable variable and method names was utilised in only 29% of projects, while it was not utilised in 68% of projects (Figure 4.6). This could be caused by the fact that mutability was avoided in most projects, which would mean that such variables and methods may not be used at all in the first place.

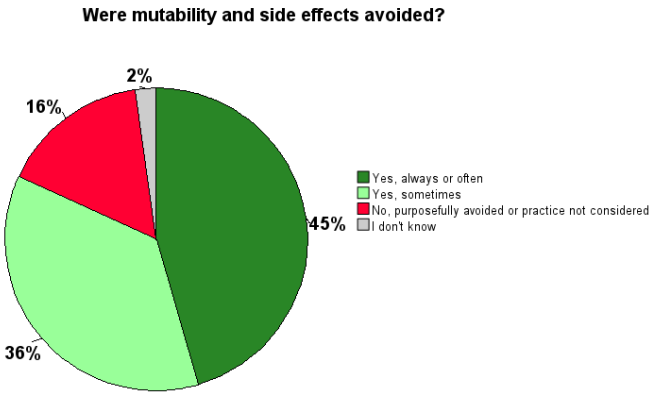

Figure 4.5: Avoiding Mutability and Side Effects (DPS 3.2.)

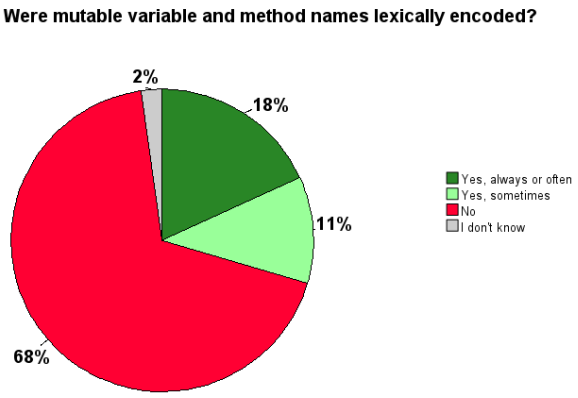

Figure 4.6: Lexical Encoding (DPS 3.3.)

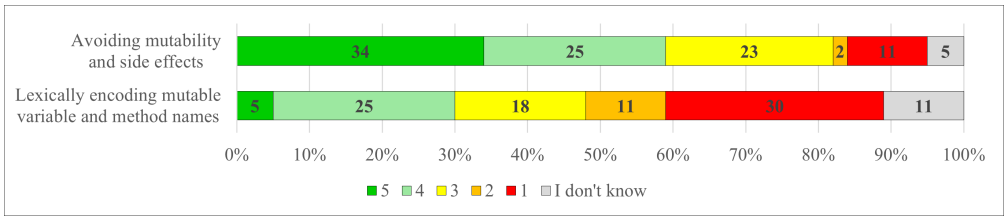

Figure 4.7: Programming Paradigm Practices Impact Ratings

Coding Convention Practices

Null values were not purposefully avoided in 61% of the projects (Figure 4.8). This is somewhat unexpected, although it is likely that nulls would be more often avoided in e.g. embedded systems rather than the target project types of this survey. Languages other than English were used to a considerable degree in 20% of projects, which is a surprisingly high percentage (Figure 4.9).

Code style was established in a vast majority of projects, however it was enforced in only 55% of projects (Figure 4.10).

Refactoring was done in separate branches in 66% of projects, while code was not refactored during feature development at all in only 14% of projects (Figure 4.11).

Variables were created for magic numbers in 82% of projects (Figure 4.12). Considering that most modern IDEs display parameter argument names when literal parameters are used, this is a high percentage.

Critical code was explained in almost all projects (Figure 4.13), which was unexpected because the author of this thesis was under the impression that this practice was rarely utilised. A surprisingly high percentage of projects (25%) did not utilise Regex patterns at all (Figure 4.14).

"TODO" comments were not avoided in most projects, while they were always avoided in only 14% projects (Figure 4.15).

Alarmingly, generated code was not reviewed carefully in 9% of the projects (Figure 4.16), which is a significant percentage considering the number of "Not applicable" answers. Note that the high number of "Not applicable" answers is to be expected because generative AI tools are relatively new.

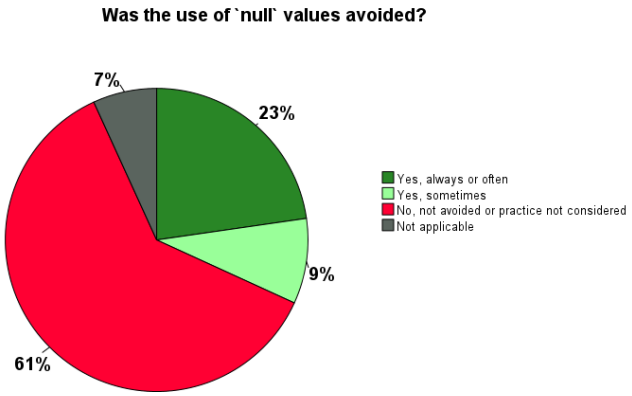

Figure 4.8: Avoiding Null Values (DPS 4.1.)

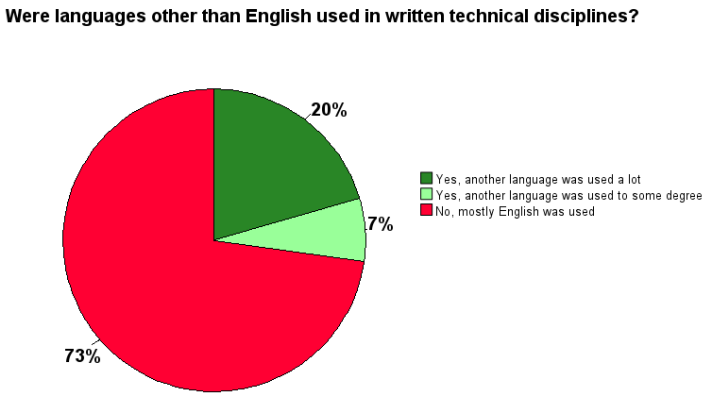

Figure 4.9: Using Languages Other Than English in Written Disciplines (DPS 4.2.)

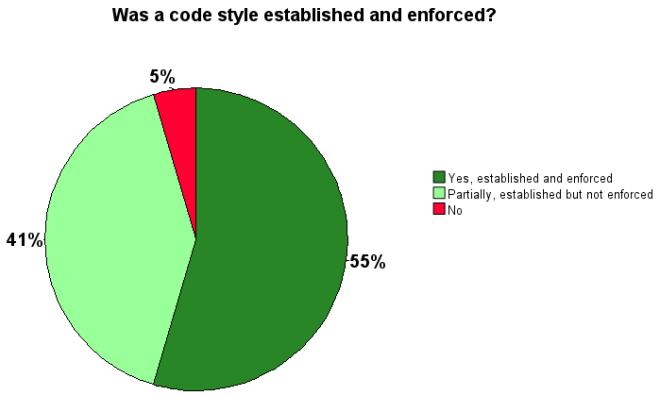

Figure 4.10: Establishing Code Style (DPS 4.3.)

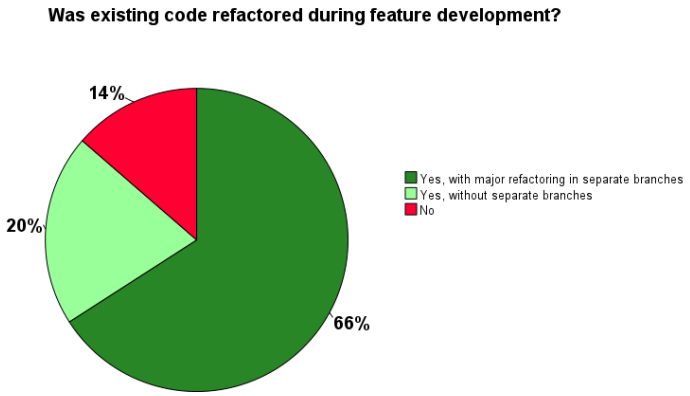

Figure 4.11: Refactoring Code During Feature Development (DPS 4.4.)

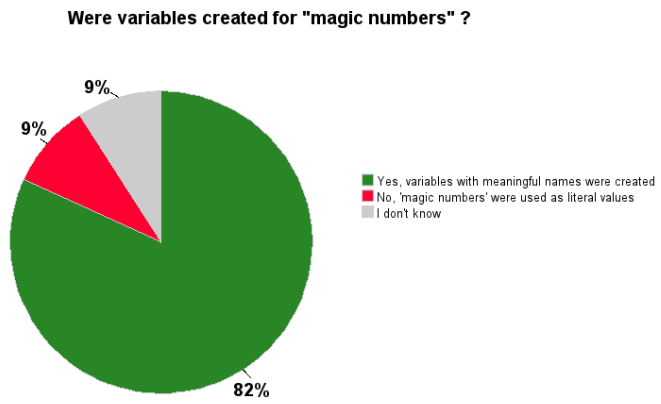

Figure 4.12: Using Variables for Magic Numbers (DPS 4.6.)

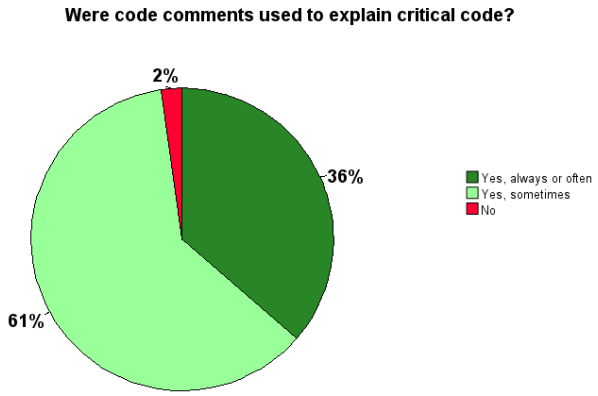

Figure 4.13: Explaining Critical Code With Code Comments (DPS 4.7.)

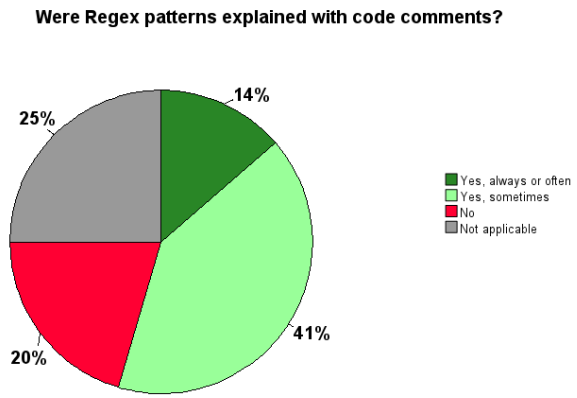

Figure 4.14: Explaining Regex Patterns With Code Comments (DPS 4.8.)

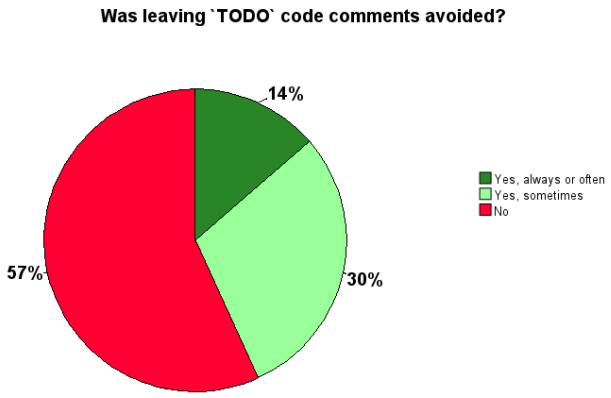

Figure 4.15: Avoiding TODO Comments (DPS 4.9.)

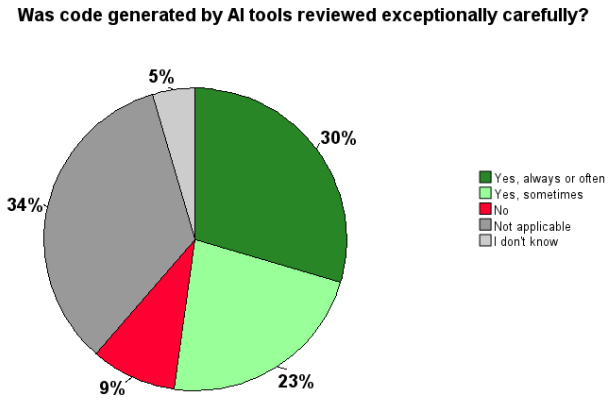

Figure 4.16: Reviewing Code Generated By AI (DPS 4.10.)

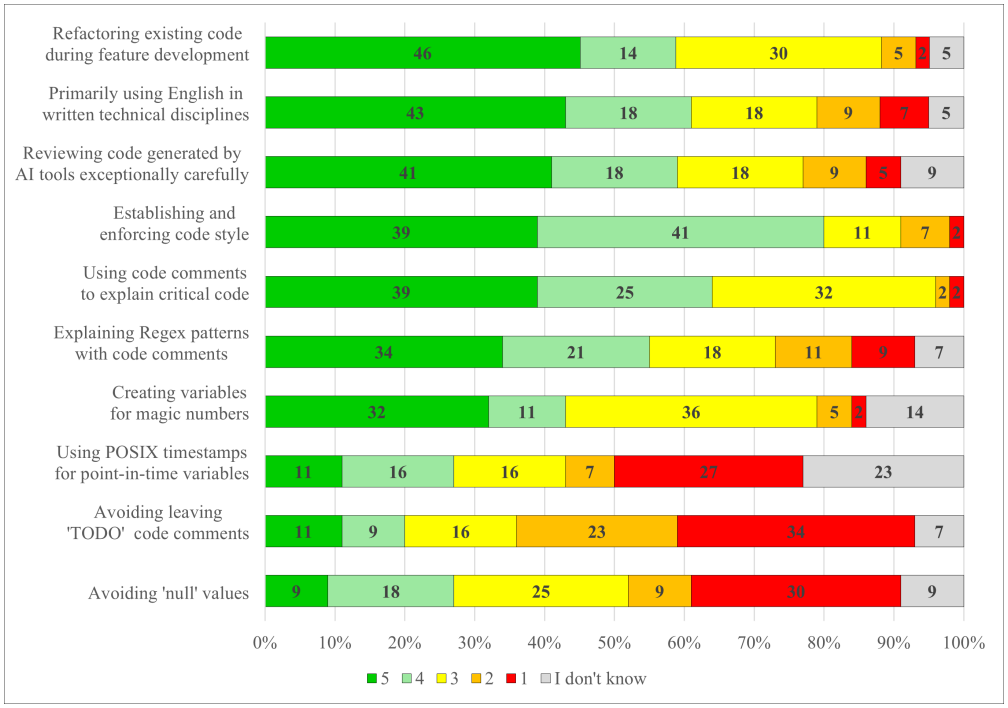

Figure 4.17: Coding Convention Practices Impact Ratings

Version Control Practices

The impactfulness of the practices in this category were relatively highly rated, with the exception of commit message formats and branch naming (Figure 4.18). This matches the expectations of the author of this thesis, because branching and merging strategies directly impact how code changes are managed, whereas the naming of branches and commit messages primarily affect recoverability. Moreover, previous versions of code are not reverted or recovered all that often.

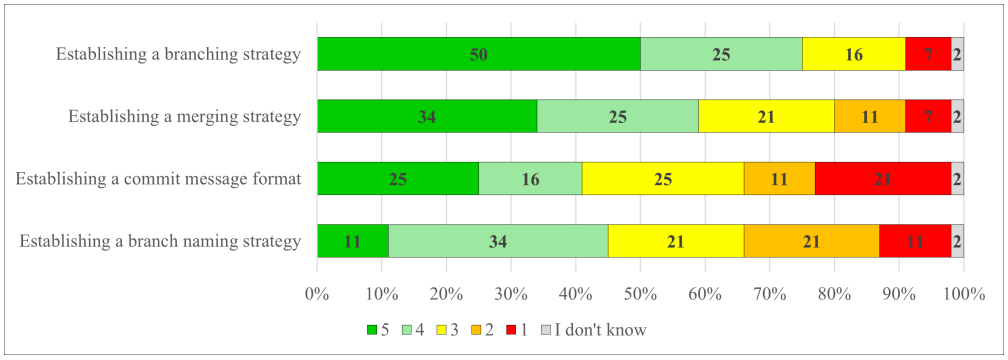

Figure 4.18: Version Control Practices Impact Ratings

Merge Request Review Practices

Higher level design issues were not considered in only 23% of the projects (Figure 4.19). In the author's experience, most code reviewers do not consider them and rather glance over code changes, although this could be due to the personalities of the reviewers or other factors related to the author's particular environment.

In 20% of the projects, reviews for code changes were not required (Figure 4.20). This is an alarmingly high percentage, which poses the question of how well these projects are protected against vulnerable code that was introduced by a malicious actor or through legitimate mistakes.

Somewhat contradicting the answers to RS1 questions, the impactfulness of requiring reviews for all codes are extremely highly rated (Figure 4.21). Moreover, the implications of not reviewing code could be further researched because many projects allow merging unreviewed code.

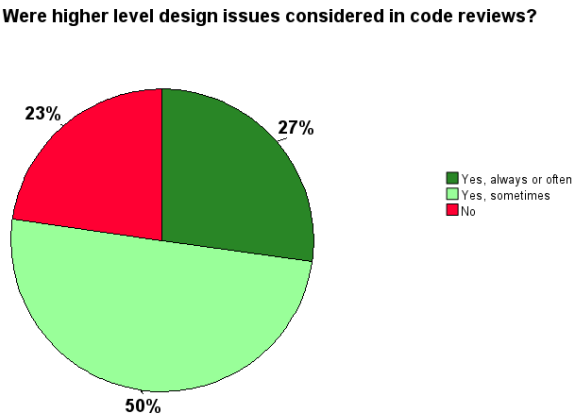

Figure 4.19: Considering Higher Level Design Issues in Code Reviews (DPS 6.2.)

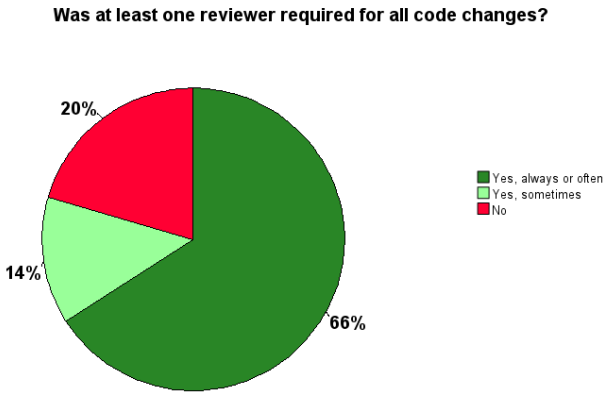

Figure 4.20: Code Change Reviewing (DPS 6.3.)

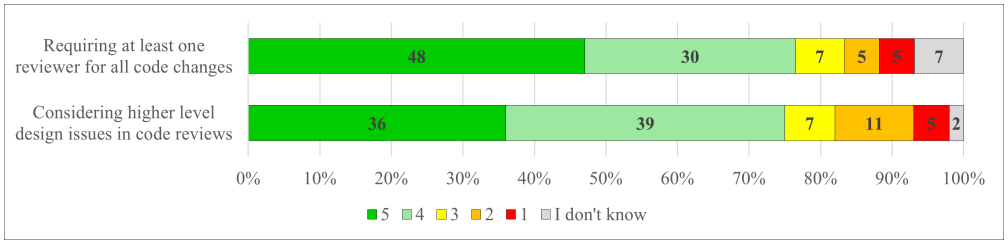

Figure 4.21: Merge Request Review Practices Impact Ratings

Process Practices

Formatter and linter tools were established in a vast majority of projects (Figure 4.22). Still, worryingly many professional software projects do not utilise them. Similar to formatter and linter tools, most projects utilise static analysis tools – however worryingly many projects do not (Figure 4.23).

Automated tests were not run periodically in only 23% of projects, which was an unexpectedly high percentage in the opinion of the author of this thesis (Figure 4.24).

The answers to the questions regarding writing tests like production code and using the build-operate-check pattern when writing tests were almost identical, which would suggest that one is utilised when the previous other one is also utilised (Figures 4.25, 4.26).

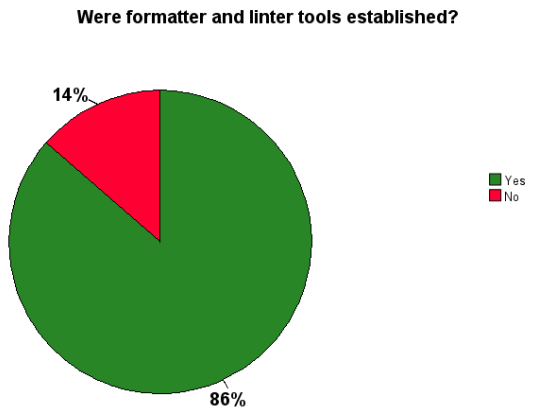

Figure 4.22: Establishing Formatter and Linter Tools (DPS 7.1.)

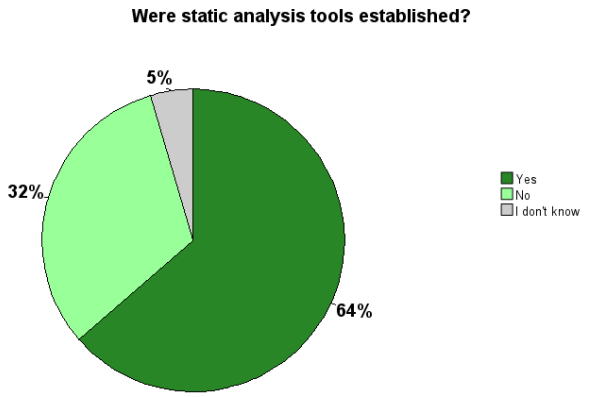

Figure 4.23: Establishing Static Analysis Tools (DPS 7.2.)

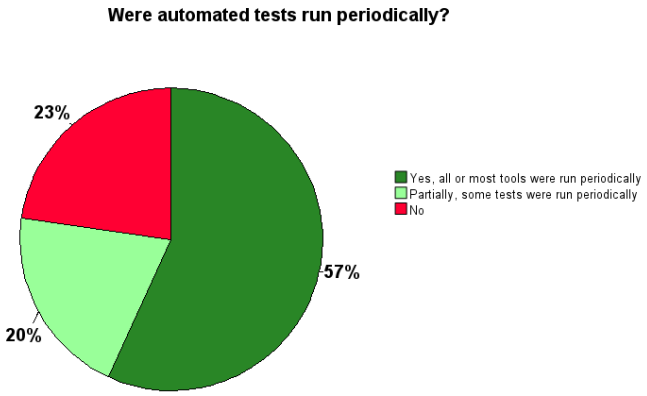

Figure 4.24: Running Automated Tests Periodically (DPS 8.2.)

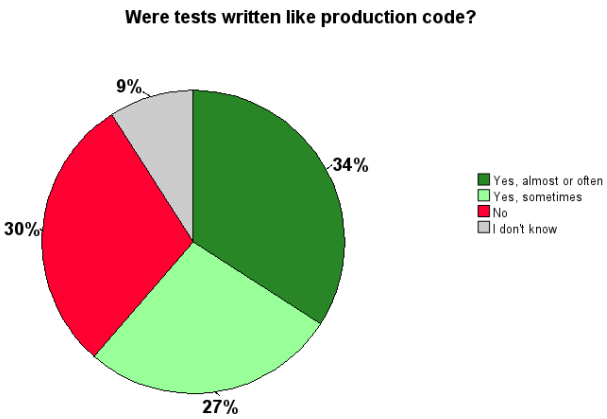

Figure 4.25: Writing Tests Like Production Code (DPS 8.3.)

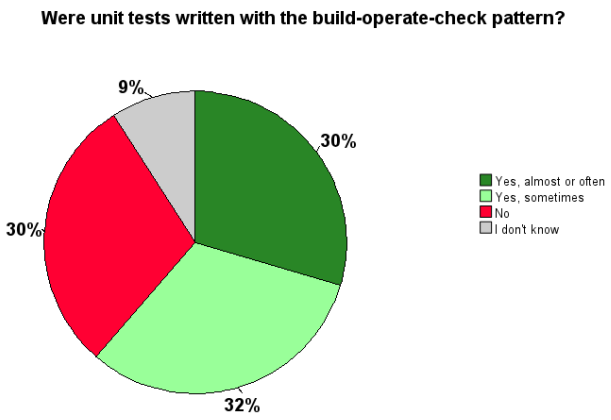

Figure 4.26: Writing Tests With the Build-Operate-Check Pattern (DPS 8.4.)

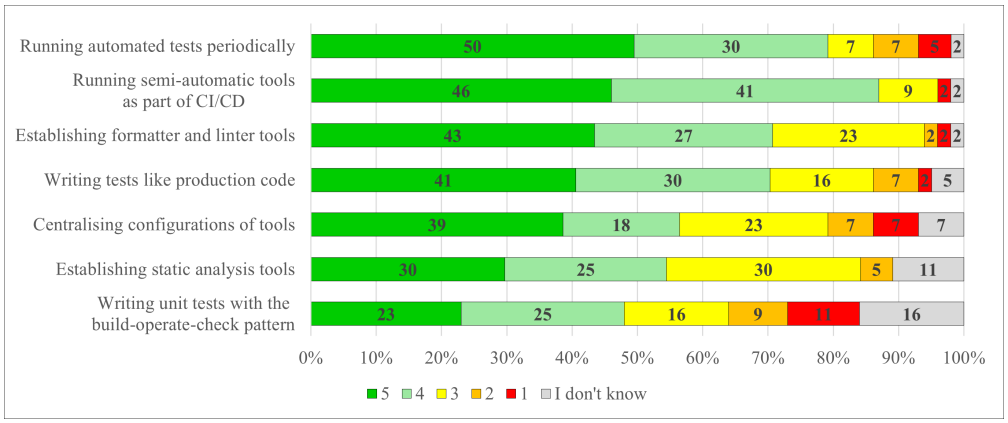

Figure 4.27: Process Practices Impact Ratings

Selected Project Quality

The projects were generally rated as possessing high quality (Figure 4.28). The project quality ratings represented a sufficient range of quality levels in terms of being able to correlate the utilisation of practices to the project quality levels, although it would have been better to have more projects of low to medium quality.

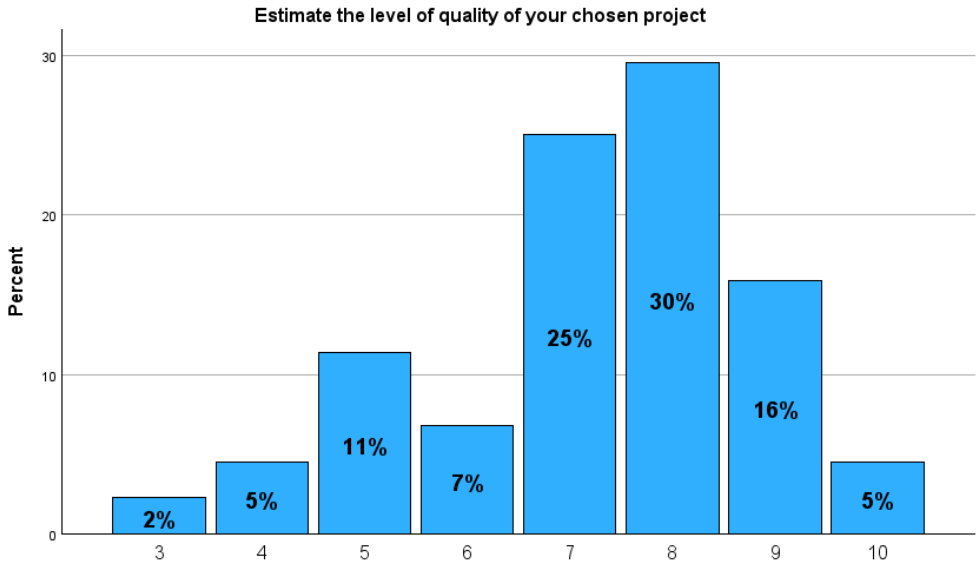

Figure 4.28: Selected Project Quality

Global Mean Impact Rating

The global mean impact rating of the DPSs was 3.57406 / 5.0, which suggests that the DPSs were considered to be fairly impactful in general. However, it should be noted that a few DPSs that were rated as having low impact on quality disproportionately lower the global mean impact rating – many of the DPSs received very high impact ratings.

4.4.2 Analysis

The results of the analysis are presented in the following table, where ρ is the Spearman's correlation factor and $\bar{x}ImpactRating$ is the mean impact rating that was calculated through RS2. Note that the DPSs will be sorted in descending order by their ρ values in Chapter 6 – no sorting is used in the following table.

DPS	ρ	P-value	$\bar{x}$ Impact Rating
DPS 2.1. Write and Maintain Requirements Documents	0.33169	0.01706	3.50000
DPS 2.2. Write an Initial Project Plan Document	0.15816	0.17491	3.09524
DPS 2.3. Write and Maintain Specification Documents	0.28941	0.02836	3.46512
DPS 2.4. Write and Maintain Technical Design Documents	0.37040	0.00666	3.44186
DPS 2.5. Create Draft Design Documents	0.17122	0.15201	3.26829
DPS 2.6. Require at Least One Reviewer For Documents	-0.04849	0.38019	3.81818
DPS 3.2. Avoid Mutability and Side Effects	0.18015	0.12384	3.71429
DPS 3.3. Lexically Encode Mutable Variable and Method Names	0.14954	0.16927	2.58974
DPS 4.1. Avoid the Use of Null Values	0.17967	0.13049	2.65000
DPS 4.2. Use English in Written Technical Disciplines	0.33672	0.01271	3.85714
DPS 4.3. Establish and Enforce Code Style	0.49965	0.00028	4.06818
DPS 4.4. Refactor Existing Code During Feature Development	0.09316	0.27377	4.00000
DPS 4.5. Use POSIX Timestamps for Point-In-Time Variables	-0.19422	0.11490	2.70588
DPS 4.6. Create Variables for Magic Numbers	0.18906	0.12133	3.76316
DPS 4.7. Use Code Comments to Explain Critical Code	0.07501	0.31421	3.95455
DPS 4.8. Explain Regex Patterns With Code Comments	0.19625	0.13685	3.63415
DPS 4.9. Avoid Leaving "TODO" Code Comments	0.37452	0.00613	2.36585
DPS 4.10. Review Code Generated By AI Tools Exceptionally Carefully	-0.20475	0.15280	3.90000
DPS 5.1. Establish a Branching Strategy	0.02033	0.44981	4.13954
DPS 5.2. Establish a Branch Naming Strategy	0.10005	0.26685	3.13954

DPS	ρ	P-value	$\bar{x}$ Impact Rating
DPS 5.3. Establish a Commit Message Format	0.23352	0.07086	3.13954
DPS 5.4. Establish a Merging Strategy	0.18427	0.12439	3.69767
DPS 6.2. Consider Higher Level Design Issues in Code Reviews	0.39914	0.00364	3.93023
DPS 6.3. Require at Least One Reviewer For All Code Changes	0.29316	0.02672	4.19512
DPS 7.1. Establish Formatter and Linter Tools	0.27780	0.03394	4.09302
DPS 7.2. Establish Static Analysis Tools	0.26259	0.04647	3.89744
DPS 7.3. Centralise Configurations of Tools	0.12130	0.22207	3.80488
DPS 8.1. Run Semi-Automatic Tools as Part of CI/CD	0.30293	0.02558	4.30233
DPS 8.2. Run Automated Tests Periodically	0.32322	0.01618	4.16279
DPS 8.3. Write Tests Like Production Code	0.35315	0.01270	4.04762
DPS 8.4. Write Unit Tests With the Build-Operate-Check Pattern	0.38008	0.00778	3.45946

Table 4.2: The DPS analysis of the survey responses.

The resulting ρ values fall in the range [-0.20475, 0.49965], while the maximum range for ρ values is [-1, 1]. A positive value signifies a positive correlation (using the given DPS correlates with higher quality), while a negative value signifies a negative correlation (using the given DPS correlates with lower quality). To ensure that the correlations are statistically significant, any DPS where `P-value > 0.05` are considered to be statistically insignificant. A likely explanation for high P-values is an insufficient number of responses to the survey, or that the particular question received too few applicable responses. For such DPSs, there is insufficient evidence to prove a correlation to quality, however it does not mean that they are not correlated – the results for DPSs with a high P-value should not be used as an inclusion criteria nor an exclusion criteria when using these results.

Because a myriad of factors affect the quality of software projects, it is expected that no single software development practice is strongly correlated with quality. Indeed, the results indicate that at most moderate correlation can be found, at around $\rho = 0.5$. However, there result ρ values are relatively high because of the fact that no single software development practice can be very impactful on their own. The results also suggest that there are some software development practices that are likely not correlated to project quality (where `P-value <= 0.05` and ρ is close to 0), and that many of the DPSs are likely correlated with higher quality (where `P-value <= 0.05` and ρ is significantly above 0).

By the estimate of the author of this thesis, it appears that most statistically significant correlations roughly follow the respective mean impact ratings with a few significant exceptions. For example, DPS 4.9's mean impact rating is 2.36585/5.0 while its ρ value is 0.37452/1.0, which would imply that the respondents do not think that the DPS is impactful, while the correlation analysis implies that it is moderately correlated with higher quality.

The data that was acquired from this survey enables further analysis of the responses, such as normalisation with respect to work experience and/or project duration. Moreover, the free-form responses contain justifications and even development practice suggestions that were not present in this thesis. This survey utilises the aforementioned analysis rating systems to simplify the analysis process, however the data could be utilised in further research. Moreover, it would be interesting to compare the results of this analysis to the mean impact ratings to see whether the results of the analysis match the mean impact ratings.

5 Failed Software Project Case Study

As part of attempting to validate the software development practice suggestions that were identified in Chapter 3, a case study was conducted on the source code of a failed software project. The project is a complex web application that is centered around a process akin to an Extract, Transform and Load (ETL) process which handles vast quantities of data. The core process has strict execution performance requirements.

The customer of the project chose Evitec Solutions as the vendor for this project after another major software vendor failed to develop the project. After more than a year of development by the previous vendor with unsatisfactory results, the project implementation was audited by a third party and ultimately abandoned by the customer. The main reasons for abandoning the project were issues with code quality and the vendor achieving little tangible progress, even if quite a lot of code was already written.

The case study's goal was to test the hypothesis that the project failed at least in part due to code quality issues, and to see if the DPSs identified in this thesis were used in the project. As a consultant in the rewrite of the failed project, the author of this thesis had access to the source code delivered in the failed project, however this source code is not publicly available.

5.1 Methodology

This case study plays a minor role in this thesis and was mainly conducted out of interest in why the previous implementation failed. The usage of the DPSs identified in this thesis was analysed based on the source code by the author of this thesis with the aid of GitHub Copilot Business (Copilot) through the following steps for each DPS:

1. Copilot was asked to analyse whether the given DPS was utilised in the project
2. Copilot's answer was verified by the author of this thesis through checking the code references that were provided by Copilot
3. The author of this thesis checked three relevant source code files through random spot checks
4. If Copilot's answer was found to be incorrect in steps 2 or 3, the analysis for the given DPS was re-done manually.

With respect to DPS 3.1. and 3.2., Copilot answered that mutability and side effects were avoided while practices from functional programming were utilised. The code references provided by Copilot did not match its answer, and through spot checking files it appeared that mutability and side effects were heavily utilised in the source code. Some sections of code seemed to utilise functional programming concepts, however those same sections utilised mutation and side effects which are not permitted in pure functional programming. With respect to DPS 8.1. and 8.2., Copilot did not seem to understand the questions and answered that those practices were utilised, when in fact they were not. With respect to other DPSs, Copilot was able to accurately ascertain whether they were used or not.

It was not possible to evaluate all practices because the author of this thesis did not have access to the development team or documentation, only the source code of the failed implementation. Moreover, the source code was available as simple files, without any version control history.

5.2 Analysing Delivered Source Code

The following table presents the findings of analysing the use of the DPSs identified in this thesis in the source code of the failed project implementation. For the software development practices that were not possible to analyse through the source code alone, the answer "Unknown" is used.

DPS	**Was used**
DPS 1.1 Establish a Quality Standard	Unknown
DPS 1.2. Assign Responsibility for Quality	Unknown
DPS 2.1. Write and Maintain Requirements Documents	Unknown
DPS 2.2. Write an Initial Project Plan Document	Unknown
DPS 2.3. Write and Maintain Specification Documents	Unknown
DPS 2.4. Write and Maintain Technical Design Documents	Unknown
DPS 2.5. Create Draft Design Documents	Unknown
DPS 2.6. Require at Least One Reviewer For Documents	Unknown
DPS 3.1. Utilise Conventions From Other Paradigms	No
DPS 3.2. Avoid Mutability and Side Effects	No
DPS 3.3. Lexically Encode Mutable Variable and Method Names	No
DPS 4.1. Avoid the Use of Null Values	No
DPS 4.2. Use English in Written Technical Disciplines	No
DPS 4.3. Establish and Enforce Code Style	Yes
DPS 4.4. Refactor Existing Code During Feature Development	Unknown
DPS 4.5. Use POSIX Timestamps for Point-In-Time Variables	Yes
DPS 4.6. Create Variables for Magic Numbers	No
DPS 4.7. Use Code Comments to Explain Critical Code	Partially
DPS 4.8. Explain Regex Patterns With Code Comments	No
DPS 4.9. Avoid Leaving "TODO" Code Comments	No
DPS 4.10. Review Code Generated By AI Tools Exceptionally Carefully	Unknown
DPS 5.1. Establish a Branching Strategy	Unknown
DPS 5.2. Establish a Branch Naming Strategy	Unknown
DPS 5.3. Establish a Commit Message Format	Unknown
DPS 5.4. Establish a Merging Strategy	Unknown
DPS 6.1. Establish a Code Review Process	Unknown

DPS	Was used
DPS 6.2. Consider Higher Level Design Issues in Code Reviews	Unknown
DPS 6.3. Require at Least One Reviewer For All Code Changes	Unknown
DPS 7.1. Establish Formatter and Linter Tools	Yes
DPS 7.2. Establish Static Analysis Tools	Yes
DPS 7.3. Centralise Configurations of Tools	Yes
DPS 8.1. Run Semi-Automatic Tools as Part of CI/CD	No
DPS 8.2. Run Automated Tests Periodically	No
DPS 8.3. Write Tests Like Production Code	Yes
DPS 8.4. Write Unit Tests With the Build-Operate-Check Pattern	Yes
DPS 9.1. Conduct Pair Programming	Unknown
DPS 9.2. Conduct Retrospective Code Reviews	Unknown

Table 5.1: The DPS analysis of the failed project implementation source code.

5.3 Results

As is evident in Table 5.1, many of the DPSs that were identified in this thesis are not analysable solely through the source code. One unsure aspect of the results is the software development practices related to documentation – the source code included markdown documents which had instructions for setting up and running the project, however no documentation related to the DPSs was present. It could be that there was documentation that was placed in an external service (such as Confluence), which is why it was decided to mark the documentation-related practices as "Unknown". It could also be that there simply was no documentation for the project.

The project was written with the object-oriented programming paradigm, and did not seem to utilise conventions from other paradigms (such as avoiding mutability in functional programming). The use of `null` values was not avoided and magic variables exist in the code. Regex patterns were not explained at all through code comments, nor was the use of "TODO" comments avoided. However, critical code

was explained through code comments in many places.

A major problem with the project was that the Finnish language was liberally used in e.g. variable names and comments, often inconsistently. For example, business terms related to the Fintech sector were not translated at all, instead their Finnish names were used (without special characters), and even abbreviated. This poses a major problem to maintainability and code readability overall. Not only is it practically impossible to understand the code if one's native tongue is not Finnish, the abbreviated business terms are difficult to understand if the reader is not a native speaker or does not have domain knowledge (which is often the case with software engineers).

On the other hand, many of the DPSs were used well in the project. Code style was established and enforced, POSIX timestamps were used for point-in-time variables, formatters and linters were used, static analysis tools were used (although just one tool), and tool configurations were centralised. The tools were not run automatically as part of the project's CI/CD pipelines, nor were tests run periodically.

There were few tests in the project, however the few that were written, were written with decent code quality and utilised the build-operate-check pattern.

Of the 14 software development practices for which statistically significant correlation coefficients were found in Chapter 4, 5 were utilised in the project, the utilisation of 5 practices could not be determined, and 4 were not utilised.

In conclusion, most of the basic software development practices identified in this thesis were used in the project, with major exceptions being the use of Finnish language in written disciplines and documentation practices. However, it is possible that documentation did exist, but was stored separately from the source code. The use of Finnish language in written disciplines is very problematic, particularly because abbreviations of business terms were used without any explanation as to their meaning. In the opinion of the author of this thesis, it is likely that the lack

of utilising the software development practices suggested in this thesis negatively impacted the quality of the project, however it would appear that the project had other major flaws – such as poor architecture and poor scalability – that contributed more to the demise of the project.

6 Discussion and Conclusions

Researching and analysing the factors affecting quality in software engineering is not a simple feat due to a myriad of variables that affect the quality of the end product. It is almost impossible to point out individual factors that would affect quality significantly more than other factors. However, the results of the statistical analysis in Chapter 4 indicate that some of the DPSs positively correlated with higher project quality more than others. The analysis results could be used to directly compare the impactfulness on quality of the DPSs. Additionally, the mean impact ratings could also be used to gauge how a representative group of software engineers in the industry feels about the impact of these DPSs.

Based on the analysis of the survey responses in Chapter 4, the following table presents the DPSs that were found to be impactful on quality based on correlation. The practices are sorted in descending order by the correlation coefficient ρ values and filtered such that only practices with `P-value <= 0.05` are included.

DPS	ρ	P-value	$\bar{x}$ Impact Rating
DPS 4.3. Establish and Enforce Code Style	0.49965	0.00028	4.06818
DPS 6.2. Consider Higher Level Design Issues in Code Reviews	0.39914	0.00364	3.93023
DPS 8.4. Write Unit Tests With the Build-Operate-Check Pattern	0.38008	0.00778	3.45946
DPS 4.9. Avoid Leaving "TODO" Code Comments	0.37452	0.00613	2.36585

DPS	ρ	P-value	$\bar{x}$ Impact Rating
DPS 2.4. Write and Maintain Technical Design Documents	0.37040	0.00666	3.44186
DPS 8.3. Write Tests Like Production Code	0.35315	0.01270	4.04762
DPS 4.2. Use English in Written Technical Disciplines	0.33672	0.01271	3.85714
DPS 2.1. Write and Maintain Requirements Documents	0.33169	0.01706	3.50000
DPS 8.2. Run Automated Tests Periodically	0.32322	0.01618	4.16279
DPS 8.1. Run Semi-Automatic Tools as Part of CI/CD	0.30293	0.02558	4.30233
DPS 6.3. Require at Least One Reviewer For All Code Changes	0.29316	0.02672	4.19512
DPS 2.3. Write and Maintain Specification Documents	0.28941	0.02836	3.46512
DPS 7.1. Establish Formatter and Linter Tools	0.27780	0.03394	4.09302
DPS 7.2. Establish Static Analysis Tools	0.26259	0.04647	3.89744

Table 6.1: The DPS analysis of the survey responses sorted in descending order by ρ, where `P-value <= 0.05`.

The following table presents the DPSs for which the impactfulness on quality could not be determined due to insufficient statistical significance. The practices are sorted in descending order by ρ values and filtered such that only practices with `P-value > 0.05` are included. The practices in this table may still be impactful on quality – the high P-values simply signify that this particular survey could not reach sufficient statistical significance for the practices. The mean impact ratings could also be useful in determining if a given practice should be used.

DPS	ρ	P-value	$\bar{x}$ Impact Rating
DPS 5.3. Establish a Commit Message Format	0.23352	0.07086	3.13954

DPS	ρ	P-value	$\bar{x}$ Impact Rating
DPS 4.8. Explain Regex Patterns With Code Comments	0.19625	0.13685	3.63415
DPS 4.6. Create Variables for Magic Numbers	0.18906	0.12133	3.76316
DPS 5.4. Establish a Merging Strategy	0.18427	0.12439	3.69767
DPS 3.2. Avoid Mutability and Side Effects	0.18015	0.12384	3.71429
DPS 4.1. Avoid the Use of Null Values	0.17967	0.13049	2.65000
DPS 2.5. Create Draft Design Documents	0.17122	0.15201	3.26829
DPS 2.2. Write an Initial Project Plan Document	0.15816	0.17491	3.09524
DPS 3.3. Lexically Encode Mutable Variable and Method Names	0.14954	0.16927	2.58974
DPS 7.3. Centralise Configurations of Tools	0.12130	0.22207	3.80488
DPS 5.2. Establish a Branch Naming Strategy	0.10005	0.26685	3.13954
DPS 4.4. Refactor Existing Code During Feature Development	0.09316	0.27377	4.00000
DPS 4.7. Use Code Comments to Explain Critical Code	0.07501	0.31421	3.95455
DPS 5.1. Establish a Branching Strategy	0.02033	0.44981	4.13954
DPS 2.6. Require at Least One Reviewer For Documents	-0.04849	0.38019	3.81818
DPS 4.5. Use POSIX Timestamps for Point-In-Time Variables	-0.19422	0.11490	2.70588
DPS 4.10. Review Code Generated By AI Tools Exceptionally Carefully	-0.20475	0.15280	3.90000

Table 6.2: The DPS analysis of the survey responses sorted in descending order by ρ, where `P-value > 0.05`.

The following table presents the DPSs for which the impactfulness on quality was not determined because they were excluded from the survey for survey brevity, while DPS 6.1. was excluded by mistake. The likely level of impact that each of these practices has was included in the column "Impactfulness", based on literature (where mentioned) or the opinion of the author of this thesis.

DPS	Impactfulness
DPS 1.1 Establish a Quality Standard	Likely not directly very impactful to quality
DPS 1.2. Assign Responsibility for Quality	Often quality is implicitly the responsibility of the project manager
DPS 3.1. Utilise Conventions From Other Paradigms	Impactfulness of DPS 3.2. is likely comparable to the impactfulness of this DPS (however, P-value too low to determined impactfulness)
DPS 6.1. Establish a Code Review Process	Excluded by mistake – likely impactful on quality
DPS 9.1. Conduct Pair Programming	According to literature, probably effective for learning but not very impactful on quality
DPS 9.2. Conduct Retrospective Code Reviews	According to literature, likely effective for learning and impactful on quality.

Table 6.3: The DPSs that were excluded from the analysis.

6.1 Conclusions

As a result of research conducted in this thesis, in total 14 software DPSs were found to positively correlate with software project quality with sufficient statistical significance, with Spearman's correlation coefficient ρ values ranging between 0.49965 and 0.26259 (**RQ**). The correlation to quality could not be determined for 18 further DPSs due to insufficient statistical significance, however the survey yielded mean impactfulness ratings for all the included DPSs based on the opinions of the respondents. The survey excluded 6 DPSs, however one was estimated to be impactful in the author's opinion, and another was estimated to be impactful based on the literature review.

The resulting list of DPSs sorted by their ρ values reflects the expectations of the author of this thesis well. Curiously, the ρ values tend to match their respective mean impact ratings with a few exceptions – most notably, the ρ value of DPS 6.3.

is relatively low while its mean impact rating is the highest in the survey, and the mean impact rating of DPS 4.9. is the lowest in the survey while having a relatively high ρ value. These contradictions could be caused by the fact that this thesis does not consider the causality of the DPSs, or that the respondents' perceptions do not match reality with respect to these DPSs.

There were 6 DPSs whose impactfulness was rated as 5 or 4 by at least 75% of respondents (DPSs 4.3, 5.1, 6.2, 6.3, 8.1, 8.2). Furthermore, the impactfulness of DPS 5.1. and DPS 8.2. were rated as 5 by at least 50% of respondents. The correlation with quality was found to be statistically significant for all of these DPSs except one (DPS 5.1) – notably, DPS 4.3. and DPS 6.2. received the two highest correlation coefficient ρ values out of all the DPSs analysed in this thesis. These results add further credibility to the aforementioned conclusions.

In analysing the response data directly, it was found that in an alarmingly high percentage of projects (21%), reviews for code changes were not required at all (Figure 4.20). This poses the obvious question of how well these projects are protected against vulnerable code that was introduced by a malicious actor or through legitimate mistakes, which could be an interesting research topic in of itself. Similarly, code generated by generative AI was not reviewed carefully in 9% of the projects (Figure 4.16), which poses similar questions with respect to the cyber security and code quality of those projects.

6.2 Reflecting on the Conclusions

This thesis was written while the author was a member of the SDLC development work group at Evitec Solutions. The DPSs that were identified in Chapter 3 have already been selectively incorporated into the SDLC. The final conclusions of this thesis will be further utilised to add new requirements to the SDLC, and to adjust its current requirements. When the research for this thesis began, the SDLC fo-

cused primarily on information security aspects. The author of this thesis focused on developing the software quality side of the SDLC requirements by utilising the information gained through the research for this thesis, and its conclusions.

The DPSs that were identified in this thesis and the conclusions of this thesis could be utilised by individual software engineers to improve the quality of the code they produce, or by organisations to develop their SDLC processes. For example, these results may be used to select software development practices to utilise, or to develop quality assurance techniques related to the impactful practices. Additionally, the DPSs could serve as a basis for future research on the subject matter. For example, further research could be conducted to define the implementations for practices identified in this thesis further, as implementation details were not addressed in this thesis. This thesis could also be supplemented with research into other aspects of quality that were excluded from this thesis, such as the impact of software development methodologies, and software development practices related to methodologies.

The raw data of the survey responses could be further analysed by, for example, normalising the analysis with respect to respondent work experience, and/or project duration. The results of the analysis could also be further analysed to find out whether the analysis results truly match the respondents' opinions on the impactfulness of the DPSs, as they appeared to at a glance.

References

[1] R. Troy, A. van Lamsweerde, and A. Fugetta, "Impact of methods on productivity & quality: Panel: Impact of software engineering research on industrial practices", eng, in *ESEC '91*, ser. Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2005, pp. 480–484, ISBN: 9783540547426.

[2] Ghanbari, Hadi and Vartiainen, Tero and Siponen, Mikko, "Omission of quality software development practices: A systematic literature review", *ACM computing surveys*, vol. 51, no. 2, pp. 1–27, 2019, ISSN: 0360-0300.

[3] Farhan M Al Obisat et al., "Review of literature on software quality", *World of Computer Science and Information Technology Journal*, vol. 8, no. 5, 2018, ISSN: 2221-0741.

[4] Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson, 2008.

[5] O. Lemos, F. Silveira, F. Ferrari, T. Silva, E. Guerra, and A. Garcia, "Unraveling the code: An in-depth empirical study on the impact of development practices in auxiliary functions implementation", *Software quality journal*, vol. 32, no. 3, pp. 1137–1174, 2024, ISSN: 0963-9314.

[6] Regine Kadgien, Andreas Jedlitschka, Andrea Janes, Valentina Lenarduzzi, Xiaozhou Li, Ed., *Product-Focused Software Process Improvement*, PROFES, Springer Nature Switzerland, 2023.

[7] Yılmaz, Nebi and Tarhan, Ayça Kolukısa, "Matching terms of quality models and meta-models: Toward a unified meta-model of oss quality", *Software quality journal*, vol. 31, no. 3, 2023, ISSN: 0963-9314.

[8] F. N. Colakoglu, A. Yazici, and A. Mishra, "Software product quality metrics: A systematic mapping study", *IEEE access*, vol. 9, pp. 44 647–44 670, 2021, ISSN: 2169-3536.

[9] Linda Westfall, *The certified software quality engineer handbook*. ASQ Quality Press, 2009, ISBN: 9781951058777. [Online]. Available: https://ebookcentral.proquest.com/lib/kutu/detail.action?pq-origsite=primo&docID=3002591.

[10] Reiner R. Dumke, René Braungarten, Günter Büren, Alain Abran, Juan J. Cuadrado-Gallego, Ed., *Software Process and Product Measurement*, IWSM, Metrikon, Mensura, Springer Berlin, Heidelberg, 2008.

[11] J.E. Bardram et al., "Exploring quality attributes using architectural prototyping", in *Quality of Software Architectures and Software Quality*, R. R. et al., Ed., QoSA, Springer Berlin, Heidelberg, 2005, pp. 143–157. [Online]. Available: https://doi.org/10.1007/11558569.

[12] Jyrki Kontio, Reidar Conradi, Ed., *Software Quality - ECSQ 2002*, ECSQ, Springer Berlin, Heidelberg, 2002.

[13] D. Galin, *Software Quality - Concepts and Practice*. John Wiley & Sons, 2018, ISBN: 978-1-119-13449-7. [Online]. Available: https://app.knovel.com/hotlink/toc/id:kpSQCP0004/software-quality-concepts/software-quality-concepts.

[14] "Iso/iec 25010", International Organization for Standardization, Geneva, CH, Standard, 2023. [Online]. Available: https://www.iso.org/standard/78176.html.

[15] Sagar Bhatia. "Procedural programming [definition]". (2022), [Online]. Available: https://hackr.io/blog/procedural-programming.

[16] Gamma Erich et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[17] Stack Overflow. "2023 developer survey". (2023), [Online]. Available: https://survey.stackoverflow.com/2023/.

[18] Alexander Shvets. "Design patterns". (2024), [Online]. Available: https://refactoring.guru/design-patterns.

[19] A. Khanfor and Y. Yang, "2017 24th asia-pacific software engineering conference workshops (apsecw)", in *An Overview of Practical Impacts of Functional Programming*, 2017.

[20] Ljubomir Jerinic, "A perspective on combining different programming paradigms", *University of Novi Sad*, 1995.

[21] Eric Lippert. "Monads, part two". (Feb. 2013), [Online]. Available: https://ericlippert.com/2013/02/25/monads-part-two/.

[22] Git. "Git - fast, scalable, distributed revision control system". (2024), [Online]. Available: https://github.com/git/git.

[23] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality", *Empirical software engineering : an international journal*, vol. 21, no. 5, pp. 2146–2189, 2016, ISSN: 1382-3256.

[24] N. Ashrafi, "The impact of software process improvement on quality: In theory and practice", *Information & management*, vol. 40, no. 7, pp. 677–690, 2003, ISSN: 0378-7206.

[25] R. Morales, S. McIntosh, and F. Khomh, "Do code review practices impact design quality? a case study of the qt, vtk, and itk projects", in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, 2015, pp. 171–180, ISBN: 9781479984695.

[26] D. Port, B. Taber, and P. Emkani, "Investigating effectiveness and compliance to devops policies and practices for managing productivity and quality variability", *The Journal of systems and software*, vol. 213, pp. 112 030–, 2024, ISSN: 0164-1212.

[27] T. Hoare. "Null references: The billion dollar mistake". (2009), [Online]. Available: `https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/`.

[28] GitLab. "Branches". (2024), [Online]. Available: `https://docs.gitlab.com/ee/user/project/repository/branches/`.

[29] Lauri Oikarinen, "Koodinkatselmointikäytännöt ketterässä ohjelmistokehityksessä", *Tampereen Teknillinen Yliopisto*, 2014.

[30] Lauri Oikarinen, "Unpublished interview with the author of "koodinkatselmointikäytännöt ketterässä ohjelmistokehityksessä"", 2024.

[31] Wikipedia. "Pearson correlation coefficient". (2025), [Online]. Available: `https://en.wikipedia.org/wiki/Pearson_correlation_coefficient#For_a_sample`.

[32] Wikipedia. "Student's t-distribution". (2025), [Online]. Available: `https://en.wikipedia.org/wiki/Student%5C%27s_t-distribution`.

[33] Wikipedia. "Arithmetic mean". (2025), [Online]. Available: `https://en.wikipedia.org/wiki/Arithmetic_mean`.

Appendix A Survey Figures

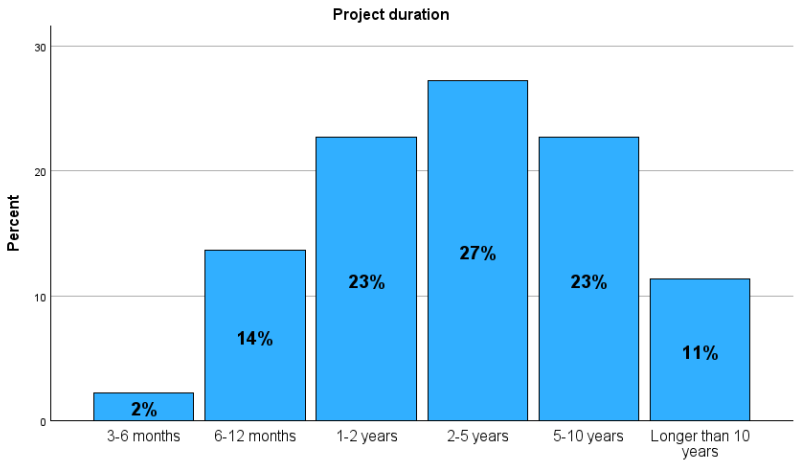

Figure A.1: Project Duration

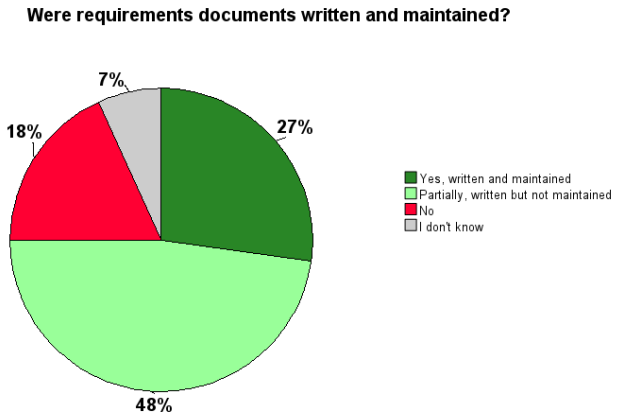

Figure A.2: Requirements Documentation (DPS 2.1.)

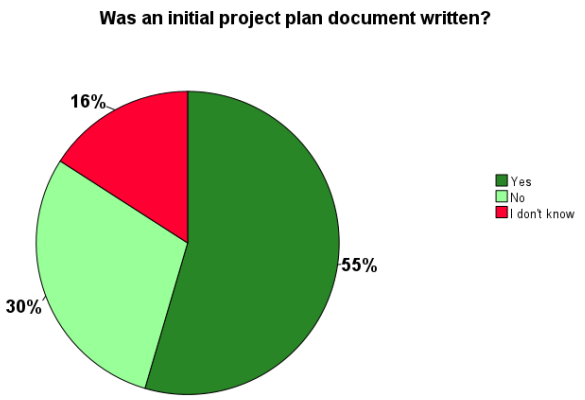

Figure A.3: Initial Project Plan Documentation (DPS 2.2.)

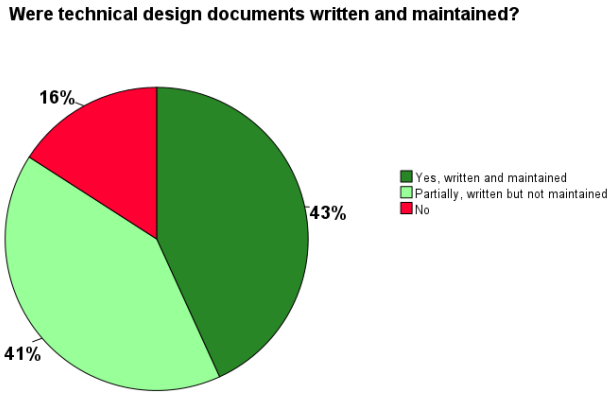

Figure A.4: Technical Design Documentation (DPS 2.4.)

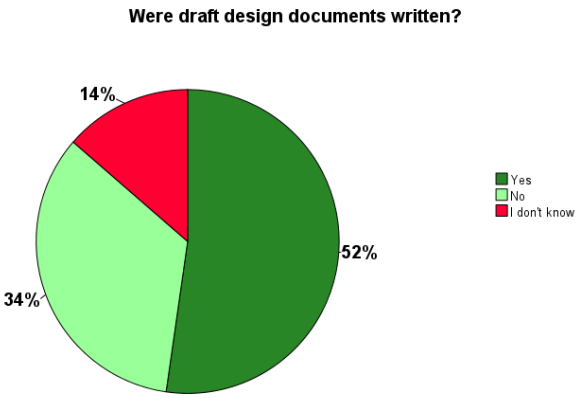

Figure A.5: Draft Design Documentation (DPS 2.5.)

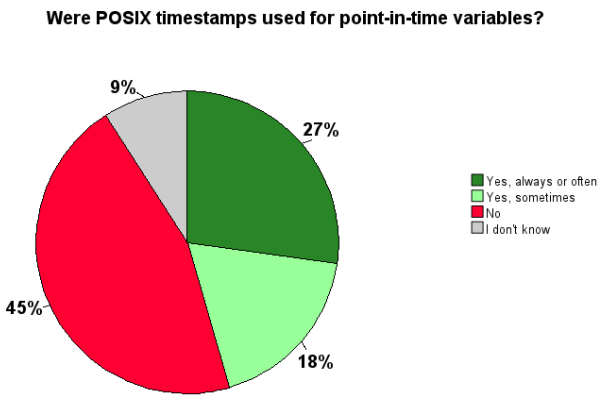

Figure A.6: Using POSIX Timestamps for Point-In-Time Variables (DPS 4.5.)

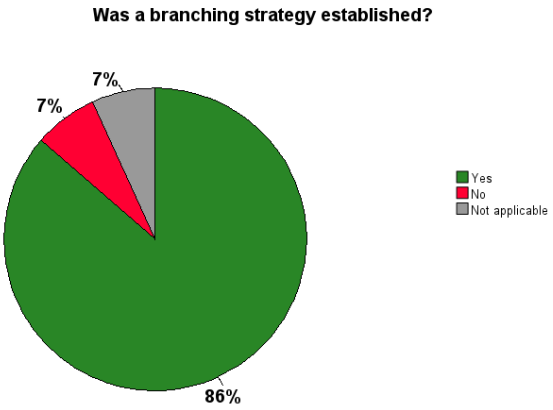

Figure A.7: Establishing Branching Strategy (DPS 5.1.)

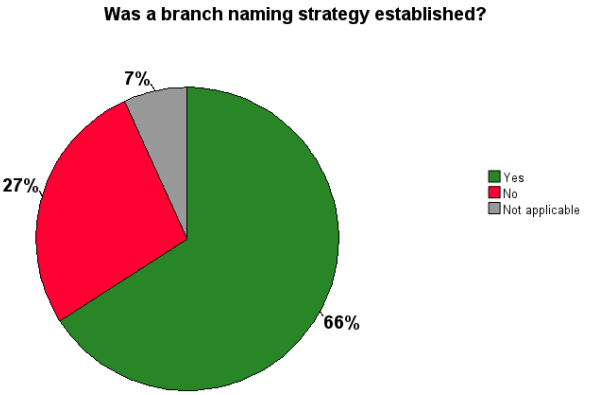

Figure A.8: Establishing Branch Naming Strategy (DPS 5.2.)

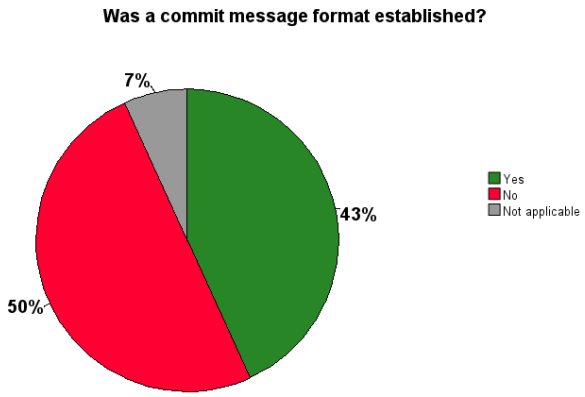

Figure A.9: Establishing Commit Message Format (DPS 5.3.)

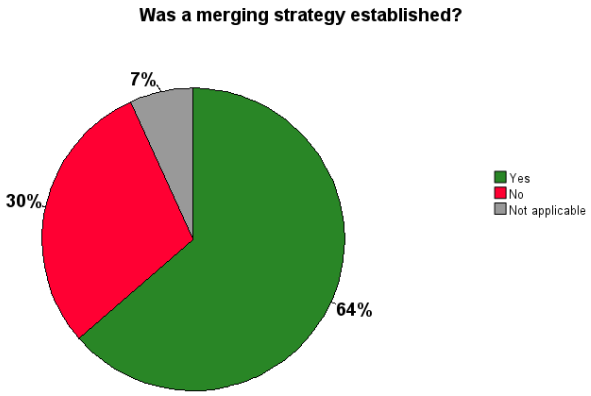

Figure A.10: Establishing Merging Strategy (DPS 5.4.)

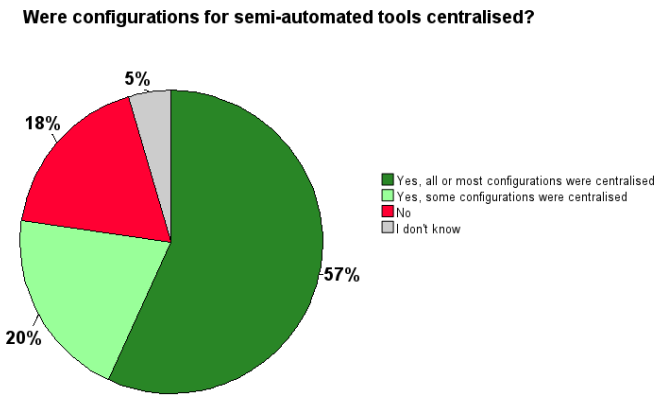

Figure A.11: Centralising Tool Configurations (DPS 7.3.)

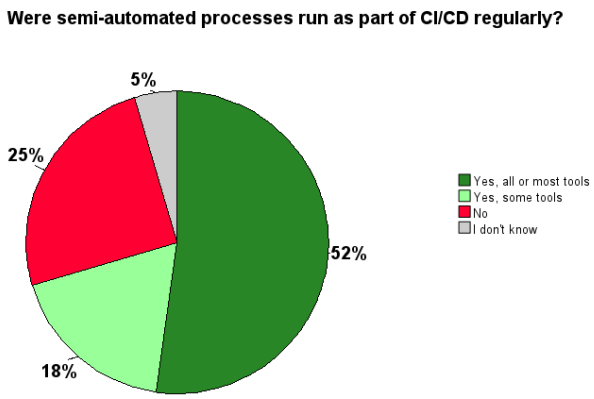

Figure A.12: Running Semi-Automated Processes In CI/CD (DPS 8.1.)

Appendix B Survey Form

The form that was used in the survey. The survey begins on the next page.

The Impact of Software Development Practices on Quality

This survey is part of the Master of Science (Technology) thesis "The Impact of Software Development Practices on Quality" by Lauri-Mikael Pakkanen at the University of Turku.

You belong to the target audience of this survey if:

- you are a professional software engineer
- you are, or have been, involved in the development of a software project of moderate (or higher) complexity as part of a software engineering team

* Indicates required question

1. **Are you a professional software engineer?** *

   If you are **not** a professional software engineer, you don't belong to the target audience of this survey. If you answer `No`, this survey will end.

   Mark only one oval.

   - ◯ Yes
   - ◯ No

Choose a Project

Choose one software development project where you have been a software developer as the basis of your answers in this survey.

The **same** project should be used in answering all project-related questions. If you have worked on multiple projects of moderate (or higher) complexity, please prioritise higher complexity projects.

If you wish to, free form questions may also be answered in Finnish.

About this survey

Answering this survey is expected to take around 10-15 minutes.
Most questions in this survey are multiple choice questions.

The responses from this survey are used to analyse the
impact on code quality of certain software development practices.

No personally identifying information is asked during this survey. All responses are completely anonymous.

Thank you for responding to this survey, your participation is appreciated!

Your Role and Experience

2. Technical work experience *

*Estimate your **total** professional, technical work experience related to software development.*

Part-time work is counted the same as full-time work. Hobbyist experience is not counted.
*If two categories apply simultaneously, pick the **higher** experience category.*

Mark only one oval.

- ◯ Less than 2 years
- ◯ 2-4 years
- ◯ 4-7 years
- ◯ 7-10 years
- ◯ More than 10 years

3. Project duration *

How long did the project last overall (including before you joined the project)?
*If active development is still ongoing, only consider the **elapsed** time up to now.*

*If two categories apply simultaneously, pick the **longer** duration category.*

Mark only one oval.

- ◯ Shorter than 3 months
- ◯ 3-6 months
- ◯ 6-12 months
- ◯ 1-2 years
- ◯ 2-5 years
- ◯ 5-10 years
- ◯ Longer than 10 years

Software Development Practices

The following sections will ask you about the usage of particular development practices in the software development project you chose.

Practices Related to Documentation

4. Were requirements documents written and maintained? *

*Requirements documents refer to the functional and non-functional requirements of the project that are specified **by the customer**.*

Mark only one oval.

- ◯ Yes, they were written and maintained
- ◯ Partially; they were written but not maintained
- ◯ No
- ◯ I don't know

5. Was an initial project plan document written? *

The initial project plan document refers to a document which contains information about e.g. the estimated schedule and personnel chosen for the project.

Mark only one oval.

- ◯ Yes
- ◯ No
- ◯ I don't know

6. Were specification documents written and maintained? *

Specification documents refer to documents that software engineers use to write code. Generally these are created by e.g. solution architects through requirements engineering.

Mark only one oval.

- ◯ Yes, they were written and maintained
- ◯ Partially; they were written but not maintained
- ◯ No
- ◯ I don't know

7. Were technical design documents written and maintained? *

Technical design documents refer to documents which describe the technical design and/or implementation.

Mark only one oval.

- ◯ Yes, they were written and maintained
- ◯ Partially; they were written but not maintained
- ◯ No
- ◯ I don't know

8. Was a software bill of materials (SBOM) document written and maintained? *

An SBOM refers to a document that lists the direct (and in some cases, indirect) dependencies of the project.

Mark only one oval.

- ◯ Yes, it was written and maintained
- ◯ Partially; it was written but not maintained
- ◯ No
- ◯ I don't know

9. Were draft design documents written? *

Draft design documents refer to documents which are created before implementation by software engineers to design the high-level functionality of a given component. These differ from technical design documents in that these are not final and should therefore usually not be maintained.

Draft design documents may be made with e.g. whiteboards, graphs, bullet points etc.

Mark only one oval.

- ◯ Yes
- ◯ No
- ◯ I don't know

10. Were project documents reviewed by someone, not including the author? *

Did someone other than the original author(s) of the documents review them?

Mark only one oval.

- ◯ Yes, all or most documents were reviewed
- ◯ Partially; some documents were reviewed
- ◯ No
- ◯ I don't know

11. In general, how impactful on quality are the following development practices in your opinion? *

1 means not impactful at all and 5 means very impactful.

Mark only one oval per row.

	I don't know	1	2	3	4	5
Writing and maintaining requirements documents						
Writing an initial project plan document						
Writing and maintaining specification documents						
Writing and maintaining technical design documents						
Writing and maintaining an SBOM document						
Writing draft design documents						
Requiring at least one reviewer for documents						

Practices Related to Programming Paradigms

12. Were mutability and side effects avoided? *

Mutability means changing a variable's value after it has been assigned an initial value. Side effects refer to e.g. functions mutating some variable outside of its own scope (global variables).

Mark only one oval.

- Yes, always or often
- Yes, sometimes
- No, they were purposefully not avoided or this practice was not considered
- I don't know

13. Were mutable variable and method names lexically encoded? *

Was lexical encoding used to distinguish mutable variables from non-mutable variables, and functions which mutate its inputs from ones that do not? An example of such encoding could be e.g. adding the word `mutable` as a prefix to name of mutable variables.

Mark only one oval.

- Yes, always or often
- Yes, sometimes
- No
- I don't know

14. In general, how impactful on quality are the following development practices in your opinion? *

1 means not impactful at all and 5 means very impactful.

Mark only one oval per row.

	I don't know	1	2	3	4	5
Avoiding mutability and side effects	◯	◯	◯	◯	◯	◯
Lexically encoding mutable variable and method names	◯	◯	◯	◯	◯	◯

Practices Related to Coding Conventions

15. Was the use of `null` values avoided? *

If the programming language(s) used in the project support `null` values, were the use of them avoided? The `null` values could be avoided by e.g. using the monad `Optional` in Java.

If the programming language doesn't support `null` values, select `not applicable`.

Mark only one oval.

- ◯ Yes, always or often
- ◯ Yes, sometimes
- ◯ No, they were purposefully not avoided or this practice was not considered
- ◯ Not applicable
- ◯ I don't know

16. Were languages other than English used in written technical disciplines? *

Disciplines refer to e.g. code and infrastructure naming.

Mark only one oval.

- ◯ Yes, another language was used a lot
- ◯ Yes, another language was used to some degree
- ◯ No, mostly English was used
- ◯ I don't know

17. Was a code style established and enforced? *

Code style could be an established code style or e.g. an in-house code style. For this question, the source of the style is not important.

Mark only one oval.

- ◯ Yes, established and enforced
- ◯ Partially; established but not enforced
- ◯ No
- ◯ I don't know

18. Was existing code refactored during feature development? *

Minor refactoring that is necessary for new feature development is not considered in this question.

Mark only one oval.

- ◯ Yes, with major refactoring done in separate branches
- ◯ Yes, without separate branches
- ◯ No
- ◯ I don't know

19. Were POSIX timestamps used for point-in-time variables? *

This questions considers only point-in-time variables where the timestamp data types make sense. POSIX is also known as e.g. Epoch and Unix timestamps.

The opposite to using POSIX timestamps would be e.g. using the `ZonedDateTime` *class in Java.*

Mark only one oval.

- ◯ Yes, always or often
- ◯ Yes, sometimes
- ◯ No
- ◯ I don't know

20. Were variables created for "magic numbers" ? *

Magic numbers refer to numbers that by themselves provide no clear meaning, e.g. the timeout duration of a request in seconds. This question considers whether these magic numbers were used as ***literal values*** *in e.g. function arguments or if variables with* ***meaningful names*** *were created for them.*

Mark only one oval.

- ◯ Yes, variables with meaningful names were created
- ◯ No, "magic numbers" were used as literal values
- ◯ I don't know

21. Were code comments used to explain critical code? *

Were code comments used to explain e.g. design choices or other critical aspects of code that are not self-evident from the code itself?

Mark only one oval.

- ◯ Yes, always or often
- ◯ Yes, sometimes
- ◯ No
- ◯ I don't know

22. Were Regex patterns explained with code comments? *

Regex is often difficult to read -- were code comments used to explain what each pattern does?

If Regex patterns were not used, select `not applicable`.

Mark only one oval.

- ◯ Yes, always or often
- ◯ Yes, sometimes
- ◯ No
- ◯ Not applicable
- ◯ I don't know

23. Was leaving `TODO` code comments avoided? *

`TODO` code comments refer to comments which denote e.g. that a piece of code should be refactored or fixed later.

Mark only one oval.

- ◯ Yes, always or often
- ◯ Yes, sometimes
- ◯ No
- ◯ I don't know

24. Was code generated by AI tools reviewed exceptionally carefully? *

Was code generated by generative AI tools, such as Github Copilot, reviewed more thoroughly than other code?

If generative AI tools were not used, select `not applicable`.

Mark only one oval.

- ◯ Yes, always or often
- ◯ Yes, sometimes
- ◯ No
- ◯ Not applicable
- ◯ I don't know

25. In general, how impactful on quality are the following development practices in your opinion? *

1 means not impactful at all and 5 means very impactful.

Mark only one oval per row.

	I don't know	1	2	3	4	5
Avoiding `null` values						
Primarily using English in written technical disciplines						
Establishing and enforcing code style						
Refactoring existing code during feature development						
Using POSIX timestamps for point-in-time variables						
Creating variables for magic numbers						
Using code comments to						

explain critical code						
Explaining Regex patterns with code comments						
Avoiding leaving `TODO` code comments						
Reviewing code generated by AI tools exceptionally carefully						

Practices Related to Version Control

*These questions are only applicable if **Git** was used as a Version Control System (VCS). **If some other kind of VCS was used, answer `Not applicable` to all questions in this section.***

26. Was a branching strategy established? *

A branching strategy dictates what branches are to be used and how they are to be used. Often such a strategy involves e.g. feature branches and feature trunks that are eventually merged into a main branch.

Mark only one oval.

- Yes
- No
- I don't know
- Not applicable

27. Was a branch naming strategy established? *

A branch naming strategy dictates how branches should be named, e.g. requiring an issue ID to be part of the branch name.

Mark only one oval.

- Yes
- No
- I don't know
- Not applicable

28. Was a commit message format established? *

A commit message format could e.g. require an issue ID to be part of the commit message.

Mark only one oval.

- Yes
- No
- I don't know
- Not applicable

29. Was a merging strategy established? *

A merging strategy dictates how changes to code should be merged, e.g. requiring commit squashing before merging.

Mark only one oval.

- ◯ Yes
- ◯ No
- ◯ I don't know
- ◯ Not applicable

30. In general, how impactful on quality are the following development practices in your opinion? *

1 means not impactful at all and 5 means very impactful.

Mark only one oval per row.

	I don't know	1	2	3	4	5
Establishing a branching strategy	◯	◯	◯	◯	◯	◯
Establishing a branch naming strategy	◯	◯	◯	◯	◯	◯
Establishing a commit message format	◯	◯	◯	◯	◯	◯
Establishing a merging strategy	◯	◯	◯	◯	◯	◯

Practices Related to Merge Request Reviews

31. Were higher level design issues considered in code reviews? *

Mark only one oval.

- ◯ Yes, always or often
- ◯ Yes, sometimes
- ◯ No
- ◯ I don't know

32. Was at least one reviewer required for all code changes? *

Mark only one oval.

- ◯ Yes, always or often
- ◯ Yes, sometimes
- ◯ No
- ◯ I don't know

33. In general, how impactful on quality are the following development practices in your opinion? *

1 means not impactful at all and 5 means very impactful.

Mark only one oval per row.

	I don't know	1	2	3	4	5
Considering higher level design issues in code reviews	◯	◯	◯	◯	◯	◯
Requiring at least one reviewer for all code changes	◯	◯	◯	◯	◯	◯

Practices Related to Semi-Automated and Automated Processes

34. Were formatter and linter tools established? *

Mark only one oval.

- ◯ Yes
- ◯ No
- ◯ I don't know

35. Were static analysis tools established? *

*This question **excludes** formatters and linters, although they are often considered to be static analysis tools.*

Mark only one oval.

- ◯ Yes
- ◯ No
- ◯ I don't know

36. Were configurations for semi-automated tools centralised? *

This question considers the configuration of formatters, linters and static analysis tools.

Mark only one oval.

- ◯ Yes, all or most configurations were centralised
- ◯ Yes, some configurations were centralised
- ◯ No
- ◯ I don't know

37. Were semi-automated processes run as part of CI/CD regularly? *

Were the tools run regularly as part of CI/CD (continuous integration and continuous delivery), e.g. running formatters and linters during merge requests?

Mark only one oval.

- ◯ Yes, all or most tools
- ◯ Yes, some tools
- ◯ No
- ◯ I don't know

38. Were automated tests run periodically? *

Were automated tests run regularly, e.g. every night?

Mark only one oval.

- ◯ Yes, all or most tests were run periodically
- ◯ Partially; some tests were run periodically
- ◯ No
- ◯ I don't know

39. Were tests written like production code? *

If you answer no, it means code in tests was e.g. considered to be less important code quality wise.

Mark only one oval.

- ◯ Yes, always or often
- ◯ Yes, sometimes
- ◯ No
- ◯ I don't know

40. Were unit tests written with the build-operate-check pattern? *

Tests written using the build-operate-check pattern divide test functions into distinct sections -- first building any necessary variables, then operating the code, and finally checking the results (instead of e.g. mixing these steps between one another).

Mark only one oval.

- ◯ Yes, always or often
- ◯ Yes, sometimes
- ◯ No
- ◯ I don't know

41. In general, how impactful on quality are the following development practices in your opinion? *

1 means not impactful at all and 5 means very impactful.

Mark only one oval per row.

	I don't know	1	2	3	4	5
Establishing formatter and linter tools						
Establishing static analysis tools						
Centralising configurations of tools						
Running semi-automatic tools as part of CI/CD						
Running automated tests periodically						
Writing tests like production code						
Writing unit tests with the build-operate-check pattern						

Resulting Project Quality

42. Estimate the level of quality of your chosen project *

In your opinion, what is the overall level of quality of the project you chose?

1 means very poor quality and 10 means very high quality *(does not necessarily mean that the project is perfect).*

If active development is still ongoing, consider the current state of the project.

1 2 3 4 5 6 7 8 9 10

☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆

43. Free form response

If there is anything else you would like to say about this topic or you wish to clarify your answer(s), please write it here. Otherwise, leave this blank.

This content is neither created nor endorsed by Google.

Google Forms

Appendix C Survey Analysis Scripts

The source code of the scripts that were used to analyse survey results.

Open source code of the analysis scripts:
https://github.com/lm-pakkanen/di-analyser

Figure C.1: Survey analysis scripts.