Bachelor Degree Project

Comparative Analysis of Programming Approaches in Software Development
An Empirical Study of Solo, Pair, and Mob Programming

Author: Yibo Wang, Andreas Manos
Supervisor: Diego Perez
Semester: VT/HT 23XY
Subject: Computer Science

Abstract

This research study makes a comparison of solo programming, pair programming, and mob programming on collaboration, knowledge sharing, stress levels, productivity, and efficiency. The study draws insights from the analysis of data from other research papers and articles, and an experiment, which was conducted simulating a real life developing environment for each programming approach. The findings reveal that pair and mob programming are more effective in promoting collaboration and knowledge sharing than solo programming, with the latter having an edge over the former. Mob programming stands out in terms of teamwork, problem-solving, and celebrating team achievements. In contrast, solo programming is characterized by low levels of active participation and collaborative problem-solving. While solo programmers may also exchange knowledge, pair and mob programming are better suited for fostering knowledge sharing. Regarding stress, the experiment shows that solo programmers feel more stressed by accumulating difficulties. Mob programmers experience stress in task management, while pair programmers report lower stress levels. Productivity and efficiency vary across programming practices, with mob programming displaying high quality and efficiency and solo programming achieving higher scores but with lower efficiency. These findings underscore the significance of taking into account task nature, desired outcomes, and team dynamics in selecting programming practices. Additional research is imperative to explore the lasting implications, effectiveness in diverse environments, and impact on productivity and wellbeing in the technology domain.

Keywords: collaboration, knowledge sharing, stress levels, productivity, efficiency, solo programming, pair programming, mob programming, teamwork, active participation, software development industry.

Contents

1 Introduction

As software development evolves, so does the programming approach. Programming is a complex discipline that requires accuracy, focus, and collaboration. While single-person programming has long been a popular choice for developers, methods such as pair programming and mob programming have begun to gain ground in the industry among developing teams.

A few technical terms are explained here:

- Productivity: In the present thesis, the term productivity pertains to the quantum of work accomplished by a programmer or a group of programmers within a specific time period. This metric could be assessed in terms of various factors such as lines of code, completed tasks, and other relevant indicators.
- Effectiveness: Effectiveness pertains to the caliber of output generated by an individual programmer or a collective of programmers. This can be evaluated with regard to the quantity of software faults, the functionality of the software, or the contentment of the ultimate users.
- Team Cooperation: Team collaboration alludes to the aptitude of a group of programmers to engage in effective cooperation. This can be appraised in relation to the team's communication, allocation of responsibilities, or the resolution of disputes.
- Knowledge Distribution: Knowledge distribution refers to the extent to which knowledge about the software and the programming tasks is shared among the team members. This could be measured by assessing the understanding of each team member about the overall project and their specific tasks.
- Work Stress Levels: The concept of work stress levels pertains to the degree of stress encountered by a programmer or a group of programmers during their software development activities. The evaluation of such levels may be carried out by means of self-reporting methodologies, behavioral observations, or physiological markers.

This report compares the three programming approaches while delving into five key areas: productivity, efficiency, teamwork, knowledge distribution, and stress levels, to provide a detailed analysis of each programming practice. The data for this study will be collected from a simulated experiment.

This study aims to fill a gap in the current literature by comparing single, pair, and mob programming methods under a realistic working environment. As most of the existing literature either is focused on examining a single programming approach [1][2][3][4][5][13] or on comparing at most two of the approaches [6][7][11][12]. Furthermore, a single research paper that dueled on comparing the three programming approaches by Roque Hernández et al. [8], was focused on the learning outcome of each approach and was conducted with a sample of students as participants.

The objective of this study is to explicate the relative merits and demerits of various methodologies concerning productivity, efficacy, collaborative efforts, knowledge dissemination, and stress levels.

In carrying out our research for this thesis, we had a very clear and structured division of labor. Firstly, I was responsible for conducting literature research to collect and collate relevant information on solo programming, pair programming and mob programming, which included their respective strengths, as well as exploring the research successes that already exist. My partner Andreas, who is a regular employee of Fortnox, was responsible for designing the experiments and conducting them as well as analyzing the data. During this process, we regularly communicate and discuss, sharing our findings and understandings to help us better understand the research questions and reduce the conflicts that arise from working in pairs. Throughout the writing process, we both reviewed and edited all chapters together to ensure that our research objectives were met. In this way, we collaborated on this thesis research, each contributing their expertise.

1.1 Background

The software industry has undergone evolutionary changes that have brought about a transformation in programming practices. Although traditional solo programming remains prevalent, it has been augmented by collaborative approaches such as pair programming and mob programming [8]. These methods involve two people or a larger group working together on a development platform to complete a task. The rise of these collaborative programming techniques is a response to the industry's increasing demand for higher quality software and more efficient development processes. However, there is a lack of comprehensive studies comparing these different programming approaches, particularly in terms of their impact on productivity, efficiency, teamwork, knowledge distribution, and stress levels. This study was motivated by this gap in the existing research [6][7]. It aims to provide a detailed analysis of solo, pair, and mob programming methods, contributing to both academic research and industrial applications.

1.2 Related work

Numerous studies have explored the benefits and drawbacks of pair programming [1][2] and mob programming [3][4][5]. Most of these studies have focused on specific aspects of each approach, such as the effects on code quality and the extent of participant involvement. However, these studies often examine these programming methods in isolation, without comparing them directly with each other or with solo programming. Some research papers have compared pair programming and mob programming [6][7], but these comparisons often do not include solo programming, and they may not cover all the key areas of interest, such as productivity, efficiency, teamwork, knowledge distribution, and stress levels. Furthermore, some studies are limited to academic settings and may not reflect the realities of industrial software development [8]. This study aims to address these limitations by conducting a comprehensive comparison of solo, pair, and mob programming in a real-life corporate development environment.

1.3 Problem formulation

Although concepts such as single, pair, or mob programming have been proposed for a long time, there remains a lack of clarity regarding the advantages and disadvantages associated with each method. The aim of this study is to untangle these intricate matters and offer an alternative means of selecting an appropriate programming approach.
Some of the questions that will be answered by this thesis are:

1 What are the primary differences in productivity and effectiveness between solo, pair, and mob programming practices?

2 How do different programming practices affect individual developers' overall experience regarding team cooperation, knowledge distribution and working stress level?

1.4 Motivation

The motivation for this research is threefold: scientific interest, industrial relevance, and personal curiosity. Current software engineering literature still exhibits a significant gap in comprehensive, comparative analyses of solo programming, pair programming, and mob programming. The studies by Hannay et al. [1], Williams et al. [2], and others [3][4][5][6][7][8] offer individual pieces of the puzzle but a holistic picture is yet to be developed. This study, therefore, attempts to bridge this gap by providing an in-depth understanding of the strengths and limitations of each programming approach and offering evidence-based recommendations for future research and industry practices.

From an industry perspective, understanding programming methodologies is critical due to their influence on pivotal factors such as productivity, cost, and quality in software development. The choice of programming method is not just a technical decision; it can significantly impact productivity, staff motivation, and overall project costs [11][16]. Hence, understanding the trade-offs between different programming methods can help companies make informed decisions to optimize these factors.

Finally, this research is also driven by personal interest. Through this research, we hope to support software development teams in obtaining insights to improve productivity and job satisfaction.

1.5 Results

Based on the literature review and the objectives of this study, we anticipate the following outcomes:

- Productivity: We expect that collaborative programming methods (pair and mob programming) may lead to higher productivity compared to solo programming. This is due to the potential for increased idea generation and problem-solving capabilities when multiple individuals work together.
- Effectiveness: We anticipate that pair and mob programming might result in more effective code, as multiple individuals can catch and correct errors more efficiently than a single individual.
- Team Cooperation: We predict that pair and mob programming will score higher in team cooperation. The collaborative nature of these methods inherently

requires and promotes better communication and cooperation among team members.

- Knowledge Distribution: We expect that knowledge distribution will be more evenly spread in pair and mob programming scenarios. These methods allow for continuous exchange of information and ideas, leading to a more balanced distribution of knowledge among team members.
- Work Stress Levels: The impact on work stress levels is less clear. While collaborative programming methods could potentially reduce stress by sharing the workload, they could also increase stress due to potential conflicts or pressure to keep up with the team.

Our research will be guided by these expectations, which will assist us in interpreting our findings. Nonetheless, it is important to note that these expectations are hypotheses, and the actual results may vary depending on the specific circumstances of our study.

1.6 Scope/Limitation

This study's major objective was to offer insightful comparisons of single, pair, and mobbing programming techniques. Nevertheless, it is important to remember that this study's approach has some flaws. First off, the experiment only included four people, which could have boosted the findings. Second, each programming technique was examined on a specific day of the week, making the experiment's duration relatively brief. Due to the short amount of time available, each programming technique's long-term repercussions and advantages may not be completely realized. Finally, the results could have been impacted by the individuals' circumstances on the testing day. Future research should aim for larger sample sizes and longer experimental times to further validate these findings.

1.7 Target group

This investigation presents significant worth to a broad spectrum of interested parties. Researchers and academics can use our findings to further their understanding of different programming methods. For software engineers and project managers, our research provides practical insights that can help them optimize their programming practices to improve productivity and collaboration. The findings of this study could benefit software development organizations seeking to improve productivity, promote effective collaboration within teams, and reduce developer stress and anxiety [14].

1.8 Outline

The remainder of the document is structured as follows:

- Chapter 2: Method - This chapter outlines the research methodology, including the chosen research method and data collection procedures.
- Chapter 3: Theoretical Background - This chapter explores the theoretical foundations of solo programming, pair programming, and mob programming.

- Chapter 4: Research project - Implementation - This chapter describes the implementation of the research project, including data collection and experimental design.
- Chapter 5: Result - This chapter presents the obtained results from the literature review and experiment.
- Chapter 6: Data Analysis - This chapter analyzes the collected data.
- Chapter 7: Discussion - This chapter discusses the findings and their implications.
- Chapter 8: The present chapter proffers an exposition of the conclusions gleaned from the study and posits avenues for prospective investigations.

2 Method

This chapter is used to show the methods of this study and how to explore efficiency, which aims to solve the problem proposed in Section 1.3.

2.1 Research Project

This research adopts a mixed method, combining both quantitative and qualitative data collection and analysis methods to investigate this topic. Firstly, we conducted a literature review of existing results to understand the technique used in the previous study.

After the literature review, we conducted a study that assigned participants to use three different programming methods to solve problems in a real programming context. This experimental design allows a direct comparison of different programming approaches.

Upon completing each task, participants were interviewed to collect qualitative data on team cooperation, knowledge distribution, and work stress levels. The combination of a literature review, experimental design, and qualitative interviews facilitated a thorough exploration of the research questions raised in this study. It is important to emphasize that the literature review did not act as a post hoc comparison but rather guided the design and interpretation of the experimental component.

2.2 Research methods

A two-step research method was adopted to provide a comprehensive understanding of the phenomena under study.

2.2.1 Literature review

A literature review was conducted in this study to understand and summarize existing research on different programming methods. This review sought to investigate extant literature, comprising research papers, books, and other relevant sources, and to draw conclusive findings concerning the efficacy of diverse programming methodologies adopted in software development initiatives. The literature review process involved searching relevant databases (e.g., IEEE Xplore, ACM Digital Library, Google Scholar) using specific keywords (e.g., "pair programming," "mob programming," "programming practices," "software development"). Studies were chosen for inclusion in the reading if they were featured in peer-reviewed journals or conference proceedings, composed in the English language, and concentrated on contrasting diverse programming methodologies. Key information and data were extracted by reading and researching the literature of the selected articles. Common themes and research gaps were identified by comparing the extracted data.

2.2.2 Experimental Design and Quantitative Interview

The mixed-methods study was designed to investigate the real-world application of different programming approaches: regular (solo), pair, or mob programming. This mixed-methods approach, combining an experimental design with qualitative interviews, allowed for a comprehensive exploration of the research questions raised in this study. Each of these three approaches was implemented by a team of four developers, each being conducted for a single working day. The mixed-methods study focused on the outcomes we observed in response to the different programming approaches, including productivity, efficiency, knowledge sharing, collaboration, and work stress levels. Productivity was measured by the number of programming tasks solved during the day under each approach. Tasks were of similar complexity to ensure a fair comparison. In the context of pair and solo programming, different tasks were assigned to each pair or individual developer, while in mob programming, the entire team worked collectively on the same task. Efficiency was gauged based on the code quality, which was evaluated considering factors like readability, adherence to coding standards, and the absence of bugs or errors [15]. The time required to correctly complete the task was another objective measure of productivity and efficiency. As for the subjective measures, knowledge sharing and collaboration were assessed through post-task surveys. The questionnaires consisted of inquiries fashioned to gather the participants' apprehensions pertaining to their team's collaboration and the apportionment of expertise amidst the members of their team whilst undertaking the task. Lastly, work stress levels were quantified using a validated, standardized questionnaire [10]. This questionnaire was administered after each programming approach was implemented, providing insight into the potential stress or anxiety experienced by the developers under the different programming conditions.

2.3 Reliability and Validity

To enhance the reliability and validity of our research, we thoroughly reviewed and analyzed documents about solo, pair, and mob programming. This helped our research avoid duplication of effort and ensured that our findings were grounded in existing knowledge.

Furthermore, we subjected the same development team to three distinct programming tasks to mitigate the impact of individual testers' skill levels on the experiments. We adhered to identical metrics to evaluate productivity and performance, such as maintaining identical criteria for code quality, and allocating the same amount of time to the programmers taking part in each experiment.

In order to attain more exact results, our project contributors carried out solitary consultations. To guarantee heightened precision, we ensured that the contributors were ignorant of the interrogatories and responses beforehand and proctored the survey solely after they had fulfilled the programming trials.

The survey on work stress was conducted using stress level scales that have already been employed by professional psychologists [10]. This approach has enhanced the reliability of the results. Nonetheless, it is crucial to acknowledge that the scales used in the survey are reliant on self-reported data. This means that the outcomes could be swayed by various factors such as the current mood of the participant and the level of comprehension of the questions.

The utilization of authentic programming tasks in the investigation and the real-life milieu in which it was executed have augmented the ecological authenticity of the discoveries. However, the small size of the sample and the particular characteristics of the participants, including their level of expertise, familiarity with each other, and other similar factors, could potentially limit the applicability of the results. Thus, it is imperative to carry out prospective studies with more sizable and varied samples to substantiate and broaden the extant findings. In doing so, we will be better equipped to assess the validity and dependability of the outcomes. Furthermore, forthcoming studies could also contemplate the employment of diverse research techniques to complement and amplify the current findings.

2.4 Ethical Considerations

Various ethical considerations were made in this study to ensure that the research was conducted responsibly and respectfully. Confidentiality, bias in the sampling process, and obtaining the consent of the participants to experiment are some of these ethical considerations.

The experiment prioritizes the privacy of its participants, ensuring that all data is stored and processed anonymously and securely and following General Data Protection Regulation (GDPR) guidelines [9]. Access to the data, related to the programming tasks, is restricted to the research team. Furthermore, to protect proprietary technology, the resulting code, associated tools, and processes are not publicly available. It is important to note that when publishing the results in this public document, only aggregated results are presented, thereby maintaining the confidentiality of individual participant data while providing insight into the experimental A comprehensive overview of the results.

The design of the study aimed to present an equitable portrayal of the gender, age, and programming proficiency of the participants. It is imperative to note, however, that the sample size is relatively limited, and thus, the outcomes may not be applicable in all circumstances.

Prior to the commencement of the study, all participants were duly notified that their participation was voluntary and that they had the liberty to withdraw from the study at any point without any negative impact. In order to guarantee that the subjects fully grasped the underlying objective of the investigation and provided their consent to partake, informed consent was procured from each individual prior to the initiation of the study.

3 Theoretical Background

In the realm of software production and development, a proficient programming approach can engender a distinct encounter for the team, not solely in terms of mitigating the workload of developers, but also in augmenting the production efficiency of enterprises. Traditional independent programming occurs when developers complete programming tasks alone. This method of programming requires developers to have a clear understanding of what they are doing. It also gives developers more freedom, which helps with creativity [12]. When more than two developers engage in programming tasks, there is a risk that it may undermine a good idea rather than enhance it. In contrast, pair programming entails two developers collaborating in pairs on a project. This approach allows for mutual supervision, leading to improved code quality. Additionally, it facilitates knowledge sharing among developers, which further contributes to its benefits. Finally, mob programming is a team programming method. More than two people can complete a task to improve productivity, but the participants in this method often lack decision-making power and may also reduce efficiency [3].

In this section, we will explain each of these programming approaches in more depth according to the related work done so far. The theoretical background of this study was developed based on a literature review. Key information and data were extracted by reading and researching the literature of the selected articles. By comparing the extracted data, common themes and research gaps were identified. The theoretical background provides a comprehensive understanding of the phenomenon under study, including the efficacy of the various programming methods employed in the software development programme. The existing literature was scrutinized in order to identify recurring motifs and areas of research that have yet to be explored. The results of this analysis informed the methodology and interpretation of the experimental component of this study.

3.1 Solo programming

Solo programming, as the name suggests, is when a developer completes a task alone. This programming method is widely used and is the most familiar. One of the advantages of programming alone is that it allows the user the freedom to change the project at their whim, and they have absolute control over the development. They can complete some tasks first at their own pace and according to their own preferences, and they can also solve difficult tasks first. They can even stop and take a break at any time without consulting other people. This approach may benefit programmers who like to work alone and have unique insights and special designs for projects [11].

However, this method also has some disadvantages. First, developers complete the task alone, and the quality of the code cannot be guaranteed. After all, a code review by a person who completes the code himself may not find the problem. Therefore, many large technology companies have regular internal peer reviews to check the tasks completed by others yesterday or last week and give some feedback. So solo programming needs to spend more time on code checking and debugging to improve code quality and reduce potential problems.

Overall, solo programming can be useful in some situations, especially for experienced developers with smaller projects. However, one needs to weigh its advantages and disadvantages before deciding on one-person programming. In many cases, pair programming or teamwork may provide a more efficient way of developing software [11].

3.2 Pair programming

Pair programming is the term used to describe the practice of two programmers collaborating on the same programming task. One member of the pair is the driver, who actively types at the computer, or records a design or architecture. The other plays the role of navigator [16]. This tactic creates a space for constant learning and information sharing among programmers, enhancing software quality, and reducing time to market [2].

Hannay and colleagues (2009) discovered that various factors, comprising the proficiency level of the programmer, the complexity of the assigned tasks, and the degree of reliance between the parties, could potentially elicit minor to moderate consequences on the results [1]. The performance of less experienced developers can approach that of an experienced duo, which is an intriguing conclusion. This is because the two beginners aid and support one another in terms of concepts, allowing them to comprehend the project more quickly and thoroughly [1].

In addition to improving software quality and efficiency, pair programming can also serve as a way for programmers to share knowledge and learn from each other [2]. Pair programming can aid in bridging knowledge gaps and fostering teamwork abilities, per research by Williams et al. (2000) [2].

Pair programming may or may not be effective, depending on several circumstances. Müller (2006) [16] claims that adding a preliminary design process can make programming more effective when done alone or in pairs [11].

Overall, pair programming will produce positive outcomes in some situations, but there are several determining factors. Some factors mentioned above should be considered when choosing pair programming.

3.3 Mob programming

Mob programming is an approach that places more emphasis on teamwork than pair programming and involves the entire team working together on a task. In this method, one team member develops code while the others watch, chat, and offer suggestions. After some time, the roles switch [4].

One of the key advantages of mob programming is that it promotes more positive cooperation, information sharing, and higher-quality code. Because mob programming enables developers to communicate more frequently throughout the development process, Mob programming will speed up problem identification and resolution for the team, according to Zuill and Meadows [3], increasing development effectiveness.

In Ståhl, Daniel, and Torvald Mårtensson's study, they highlighted multiple advantages of mob programming (mob programming), such as improved decision-making and enhanced knowledge sharing, but they also pointed out that applying it in practice may encounter some challenges [4]. According to their research, mob programming promotes workplace diversity by enhancing the participation and contribution of team members of various skill levels. With this strategy, teams can leverage their combined expertise to decide more effectively, improving the effectiveness of the entire decision-making process. However, Sthl and Mrtensson also mention that putting mob programming into practice might not be simple. Some challenges may come up, particularly when attempting to strike a balance between individual autonomy and the team's overarching objectives. This indicates that, despite mob programming's advantages, there may still be some difficulties to overcome to maintain each team member's autonomy while attaining the team's general objectives [4].

Hence, mob programming is not without its drawbacks. One disadvantage is the potential reduction of autonomy and creativity among team members, as all decisions are made collectively [5]. Creative suggestions might require unanimous approval before they can be implemented. Another issue that can arise is when numerous developers with varying levels of experience work simultaneously on the same project. This situation may cause work to progress slower than anticipated, as the team is influenced by the less experienced members.

3.4 Research gap

Current studies have examined the advantages and disadvantages of solo programming, pair programming, and mob programming, frequently contrasting two of these strategies [6, 7, 1]. Comparing all three programming approaches in the context of the software industry, however, is a gap in the literature. Furthermore, it should be noted that a considerable number of investigations conducted thus far have taken place within academic environments, which may not necessarily provide an accurate representation of the intricacies and challenges encountered by software development teams in real-world settings [8, 10, 11]. To address this issue, this study aims to conduct an experiment that compares the effectiveness of solo programming, pair programming, and mob programming under varying conditions in the software industry. This will serve to bridge the gap and further augment the conclusions drawn by Hernández et al. [8], who made a comparison of the three methodologies in an academic context.

Overall, more research is necessary to fully comprehend the specific scenarios in which each programming approach performs optimally, as well as the interplay between them.

4 Research project – Implementation

This particular chapter elucidates the intricate and multifaceted process of data collection, as well as the comprehensive design methodology employed in the experiment.

4.1 Data Collection

This study employs two distinct methodologies in data collection, namely a literature review and an experiment. The literature review serves to establish a foundational comprehension of extant literature within this domain. The principal data for this particular study was obtained via a review of academic articles germane to the research questions. The selection process underwent multiple stages to ensure the inclusion of the most pertinent studies and to promote reproducibility.

Initially, a thorough exploration was carried out across a multitude of scholarly databases, notably but not limited to IEEE Xplore, ACM Digital Library, Springer, JSTOR, and Google Scholar, by means of entering relevant keywords to retrieve pertinent articles. Our objective was to identify articles encompassing single, pair, and mob programming practices and their effects on productivity, efficiency, teamwork, knowledge distribution, and work stress levels.

Upon carrying out a thorough investigation, a straightforward filtration technique predicated on the titles and abstracts of scholarly publications was employed to exclude articles that lay beyond the purview of our research, despite potentially being focused on the three programming methodologies. Such articles, however, lacked direct relevance to the subject matter investigated in this study.

The residual articles were subsequently subjected to a comprehensive textual scrutiny. During this phase, the articles were evaluated on the basis of their direct applicability to our research inquiries. An article was deemed relevant if it encompassed empirical evidence or theoretical perspectives that were directly linked to fundamental facets of programming practices involving one, two, or more than two participants. These facets included but were not limited to productivity, effectiveness, collaborative efforts, knowledge dissemination, and work-related stress levels.

Due to the subjective nature of the selection of relevant articles, it is acceptable that there will be slight variations in the final selection of articles.

4.2. Experimental Design

The experiment was designed to implement mob programming, pair programming, and individual programming in a real-world setting. The participants were four employees of Fortnox, holding software developer roles. The team comprised two experienced developers with five to ten years of experience, and two junior developers with less than a year of working experience. This mix of experience levels aimed to mimic the realistic conditions of development teams in the industry.

The experiment was divided into three sessions, each lasting for a single working day and dedicated to one of the programming practices. The same team participated in all three sessions.

During the mob programming session, the entire team worked on the same task, following the rules of mob programming. Each developer had a distinct role: a driver, a navigator, and the rest functioned as the mob. Each role was rotated every forty-five minutes.

In the pair programming session, the team was divided into two subgroups, each working together on a single device. The rules for pair programming were more flexible, with the pairs deciding the typing time and iteration time frames.

During the solo programming session, the team was divided into individuals, each working independently following their normal routine.

The tasks for each session were designed to be of similar difficulty, determined by the number of different repositories involved and the task description. All sessions used Java as the programming language and the same computing devices.

At the end of each day, questionnaires were distributed among the participants to record measures of teamwork, knowledge sharing, and work stress levels. This data collection method allowed us to gather subjective measures from the participants' experiences under different programming conditions.

We then manipulated several dependent variables, including productivity, efficiency, knowledge sharing, level of collaboration, and stress level.

We monitored the number of tasks completed by each individual programmer, pair, or group in a predetermined amount of time, allowing us to compare the productivity of the different programming methods. In pair programming and mob programming groups, we treat the combined output as a unit for comparison purposes.

For the productivity measures, we asked each participant (whether in a single, pair, or mob programming setup) to complete a set of programming tasks. We defined a task as completed when that task would have passed all the phases of software development, which we defined as brainstorming, implementation, testing and entered the pull request phase. These programming tasks were set to be as similar in difficulty as possible.

To assess the quality of the code (a key indicator of efficiency), we performed a post-task analysis of the code generated by each group. The code was compared by the number of bugs in the code, and the review of code bugs was peer-reviewed by participants. Subjective measures, including knowledge sharing, collaboration, and work stress levels, were assessed through post-task surveys and a validated, standardized questionnaire. The survey and questionnaire were executed after participants experienced each approach, capturing participants' perceptions and experiences under different programming conditions.

4.2.1 Quantitative interview

As part of our data collection methods, we administered several questionnaires and surveys to the participants to gauge their subjective experiences with each programming approach. In order to maintain the integrity and accuracy of the results, the questions pertaining to work-related stress were meticulously crafted by strictly adhering to the guidelines provided by the SPP-10 (Perceived Stress Scale) [10]. No alterations were made to these questions, as any modifications could potentially compromise the validity of the findings. Furthermore, when designing the questionnaires for measuring collaboration and knowledge distribution, careful consideration was given to the existing body of literature on these specific subjects [3][4][6]. By drawing insights and inspiration from reputable sources, we aimed to ensure that the questionnaires captured the essential aspects of collaboration and knowledge sharing. The detailed questions are shown below:

Stress-related Questions:

1. Today, how often have you been upset because of something that happened unexpectedly?
2. Today, how often have you felt that you were unable to control the important things in your life?
3. Today, how often have you felt nervous and stressed?
4. Today, how often have you felt confident about your ability to handle your personal problems?
5. Today, how often have you felt that things were going your way?
6. Today, how often have you found that you could not cope with all the things that you had to do?
7. Today, how often have you been able to control irritations in your life?
8. Today, how often have you felt that you were on top of things?
9. Today, how often have you been angry because of things that happened that were outside of your control?
10. Today, how often have you felt that difficulties were piling up so high that you could not overcome them?

Collaboration-related Questions:

1. Did you collaborate effectively with your colleagues today to accomplish your shared goals?
2. Did you actively participate in discussions and brainstorming sessions with your team members today?
3. Did you offer constructive feedback and suggestions to your team members today?
4. Did you communicate openly and clearly with your team members today?
5. Did you work together with your team members to solve any problems that arose today?
6. Did you show respect for the opinions and contributions of your team members today?
7. Did you take responsibility for your actions and fulfill your commitments to the team today?
8. Did you recognize and celebrate the accomplishments of your team members today?
9. Did you feel that your team had a collaborative culture that supported effective teamwork and communication today?
10. Were you able to contribute effectively to the overall success of your team's projects today?

Knowledge Sharing-related Questions:

1. Did you share your technical knowledge or expertise with colleagues today?

2. Did you receive useful feedback or learn new technical skills from your colleagues today?
3. Did you encounter any roadblocks or challenges while working on a task today?
4. Did you seek advice or assistance from colleagues to overcome any roadblocks or challenges today?
5. Did you feel comfortable sharing your code or ideas with colleagues today?
6. Did you learn any new programming concepts or techniques from your colleagues today?
7. Did you feel that your colleagues were open and receptive to your suggestions or ideas today?
8. Did you take the time to explain any technical concepts or solutions to your colleagues today?
9. Did you feel that your team had a culture that supported knowledge sharing and learning today?
10. Did you find that knowledge sharing with colleagues helped you complete tasks more efficiently or effectively today?

These questions were designed to capture the participants' perceptions and experiences under different programming conditions and played a crucial role in evaluating the subjective measures of our study.

To measure the effects of different programming practices on stress, knowledge sharing, and collaboration, we employed three distinct questionnaires, each with a Likert scale of 1-5 [18]. These scales, used as additional dependent variables, allowed us to assess the subjective experiences of the participants involved in each programming method.

- Stress Questionnaire: We administered a stress questionnaire to all participants, wherein we gauged their subjective perception of stress levels through a graduated scale. Said scale spanned from the lowest score of 'Never' (1), to the highest score of 'Very often' (5). This allowed us to measure the frequency of stress-related feelings or experiences directly associated with the different programming methods.
- Knowledge Sharing Questionnaire: We evaluated the perception of knowledge sharing within the teams using a scale from 'No' (score of 1) to 'Yes, a lot' (score of 5). This helped us understand the extent to which participants felt they were sharing and gaining knowledge during the programming tasks.
- Collaboration Questionnaire: We determined the perceived level of collaboration amongst the team members using a scale from 'No' (score of 1) to 'Completely' (score of 5). This scale helped us measure the degree of collaborative interaction and engagement among the participants in the pair and mob programming groups.

5 Results

5.1 Results from Literature Review

The literature review included a review of fourteen scholarly articles. These articles provide insights into mob programming, pair programming, and one-person programming practices. The findings derived from the comprehensive review of existing literature are hereby enumerated in the subsequent sections.

5.1.1 Distribution of Papers

Among the fourteen scrutinized articles, a considerable number of them, namely five, provided extensive insights into the subject of pair programming [1, 2, 11, 16, 17]. Four articles, on the other hand, were intently focused on the subject of mob programming [3, 4, 5, 13], while another four articles offered a comparative analysis of both pair programming and mob programming [6, 7, 12, 14]. Furthermore, a solitary article delved into a comprehensive discussion of all three programming practices, including solo, pair, and mob programming [8].

5.1.2 Common Themes

Several themes emerged from the literature review, providing valuable insights into each programming practice.

- Effectiveness: All fourteen articles discussed the effectiveness of the programming practices to varying degrees. For example, Hannay et al. [1] found that pair programming can improve code quality and task completion rates. Similarly, Ståhl and Mårtensson [4] reported that mob programming can lead to high-quality code and efficient task completion.
- Collaboration: Fourteen articles highlighted the role of collaboration in programming practices. Williams et al. [2] noted that pair programming fosters effective teamwork, while Zuill and Meadows [3] observed that mob programming encourages a high level of collaboration among team members.
- Learning and Knowledge Sharing: Six articles explored the learning and knowledge sharing aspects of the programming practices. For instance, Aune et al. [5] found that mob programming promotes knowledge sharing among team members, while Müller [11] reported that pair programming can facilitate learning and knowledge transfer.
- Stress and Workload: Four articles delved into the stress levels and workload associated with different programming practices. For example, Dragos [6] found that mob programming can lead to lower stress levels compared to pair programming, while Roque Hernández et al. [8] reported that solo programming can result in a higher workload.

5.1.3 Data Extraction from Literature

The findings from the literature have:

- Pair Programming: The reviewed articles on pair programming [1, 2, 11] mainly highlighted its effectiveness in improving code quality and fostering collaboration. For example, Williams et al. [2] found that pair programming can lead to higher code quality compared to solo programming, while Hannay et al. [1] reported that pair programming can improve task completion rates.

- Mob Programming: The articles on mob programming [3, 4] discussed how this practice encourages a high level of knowledge sharing and collaboration. For instance, Zuill and Meadows [3] observed that mob programming promotes a high level of collaboration among team members, while Ståhl and Mårtensson [4] reported that mob programming can lead to high-quality code.
- Pair vs. Mob Programming: The comparison between mob and pair programming in articles [6, 7] suggested that the choice largely depends on the team's specific context and needs. For example, Dragos [6] found that mob programming can lead to lower stress levels compared to pair programming, while Kattan et al. [7] observed that pair programming can be more suitable for tasks that require a high level of collaboration.
- Solo, Pair, and Mob Programming: The article [8] discussing all three practices provided insights into when each practice might be the most suitable based on different factors. For instance, Roque Hernández et al. [8] suggested that solo programming might be more suitable for tasks that require a high level of focus, while pair and mob programming might be more suitable for tasks that require a high level of collaboration and knowledge sharing.

5.2 Results from Experiment

In the experiment, a total of four participants were involved. The results presented are calculated as averages based on the responses of these four participants. It's important to note that due to the small sample size, the results should be interpreted with caution. While they provide valuable insights into the impact of different programming approaches on productivity, efficiency, team cooperation, knowledge distribution, and work stress levels, they may not be representative of all programming teams.

All experiment results are shown below:

Figure 5.1 Collaboration in Different Programming Practices

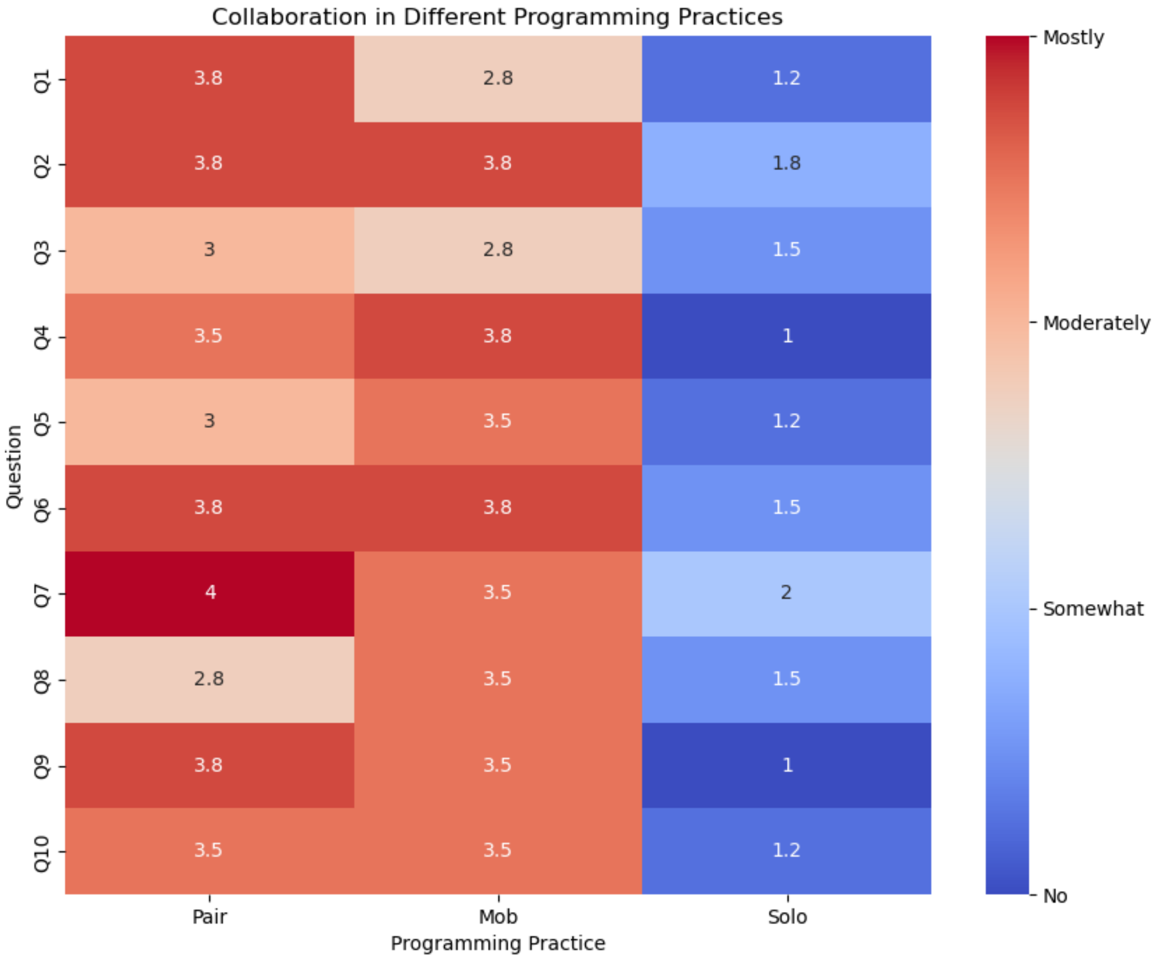

Figure 5.1 presents a heatmap illustrating the collaboration levels among different programming practices. The rows represent specific questions from Q1 to Q10 related to collaboration, while the columns represent the programming methods: Pair Programming, Mob Programming, and Solo Programming. The heatmap employs color variations to indicate the level of collaboration, with darker colors indicating higher collaboration levels and lighter colors indicating lower levels. The numerical values from 1 to 5 within each cell represent the collaboration ratings for the corresponding programming practice and question. Higher numerical values indicate a better or higher level of collaboration in addressing specific questions.

Figure 5.2 Knowledge Sharing in Different Programming Practices

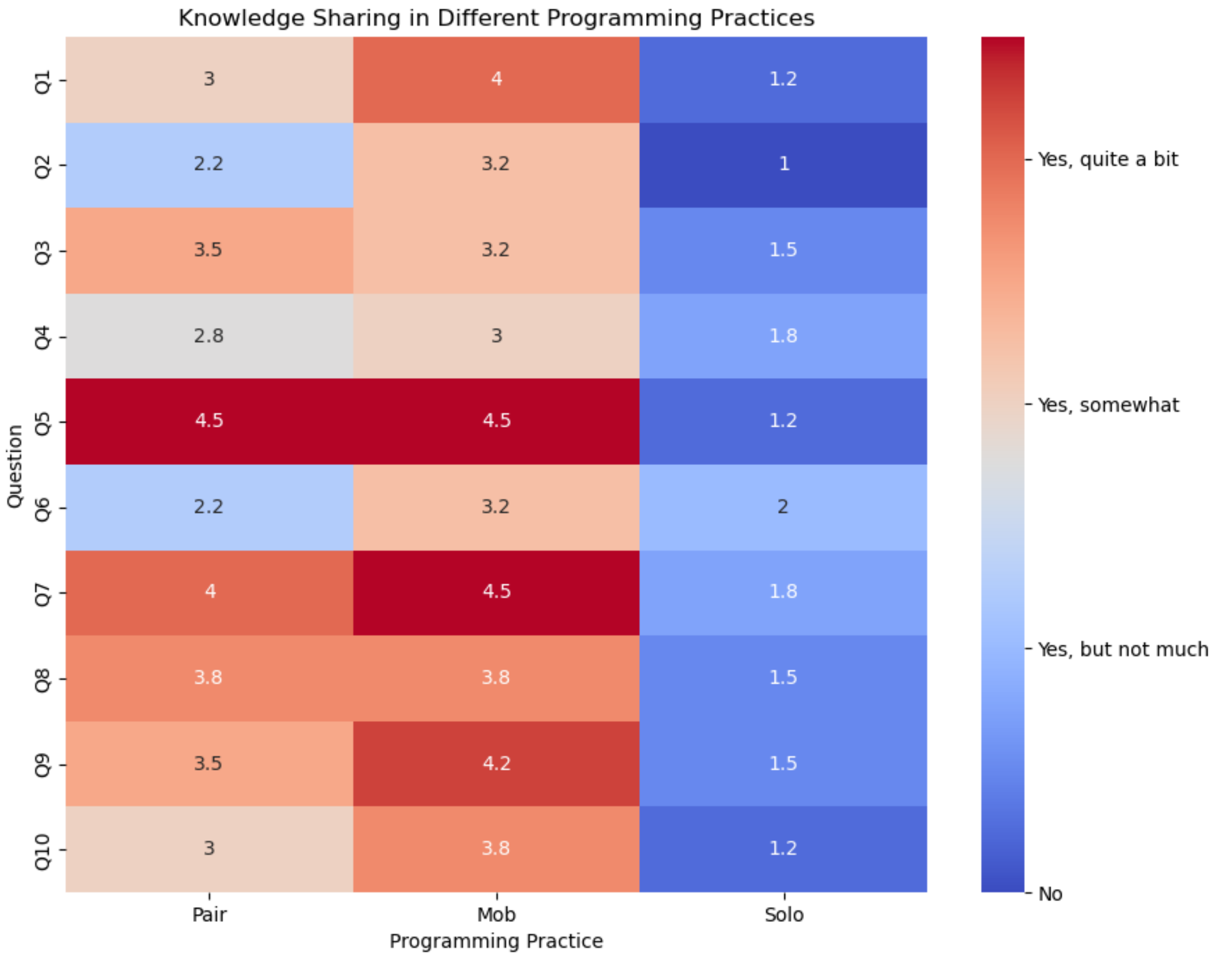

Figure 5.2 displays a heatmap illustrating the level of knowledge sharing in various programming practices. The rows correspond to specific questions from Q1 to Q10 about knowledge sharing, while the columns represent the programming methods: Pair Programming, Mob Programming, and Solo Programming. The colors in the heatmap depict the extent of knowledge sharing, with darker colors indicating a higher level and lighter colors indicating a lower level. The numeric values within each cell represent the knowledge sharing ratings for the respective programming practice and question. Higher numerical values indicate a greater degree of knowledge sharing in response to specific questions.

Figure 5.3 Stress Levels in Different Programming Practices

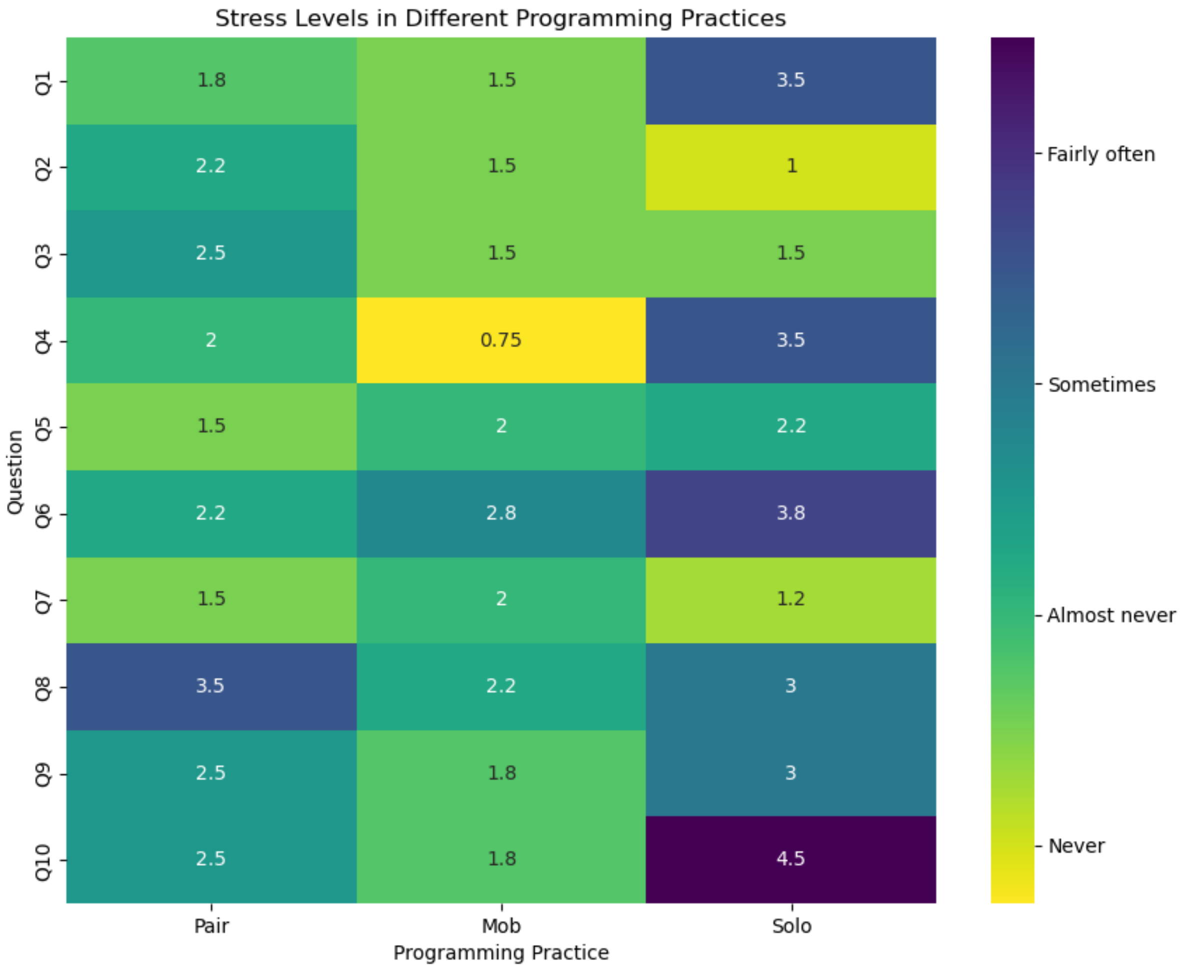

Figure 5.3 is a heatmap that visualizes the stress levels in different programming practices. The heatmap uses color variations to represent the frequency of experiencing stress, with darker colors indicating higher levels of stress and lighter colors indicating lower levels. The horizontal axis represents different programming methods, and the vertical axis represents the questions Q1 to Q10 from top to bottom. The numeric values in the cells represent the frequency of experiencing stress, ranging from 1 to 5. For example, a rating of 1 indicates that stress is "never" or "almost never" experienced, while a rating of 5 suggests that stress is "very often" experienced. By examining the colors and numerical values, we can compare and analyze the stress levels associated with different programming practices and specific stress-related questions.

Table 5.1 Productivity

Development Stage	Mob programming	Pair programming	Solo programming
Brainstorming	1	S1-1 | S2-1	S1-1 | S2-1 | S3-1 | S4-1
implementation	1	S1-1 | S2-1	S1-1 | S2-1 | S3-1 | S4-0
testing	1/2	S1-1 | S2-2	S1-0 | S2-1 | S3-0 | S4-0
total	2.5	S1 * 0.5 + S2 * 0.6 = 2.7	(S1 + S2 + S3 +S4) * 0.4 = 3.2

In Table 5.1, the productivity metrics have been ascertained via the rigorous processes of brainstorming, implementation, and testing. It is noteworthy that each task has been subjected to a meticulous evaluation process, and has been assigned a difficulty factor of either 0.6 or 0.5. With respect to each activity, i.e. brainstorming, implementation, and testing, a score of 1 has been affixed in the event of successful completion, whereas a score of 0 has been attributed in case of failure to accomplish the task at hand.

Table 5.2 Efficiency

Efficiency Metrics	Mob programming	Pair programming	Solo programming
Number of comments	5	1 | no PR for the 2nd pair	17
Number of bugs	0	1	2

In Table 5.2, the efficiency is assessed by evaluating the number comments and number of bugs found in the pull requests. Fewer comments and fewer bugs indicate higher efficiency.

6 Analysis

Using the collected data from different articles and responses to the questions related to collaboration, knowledge sharing, and stress levels, we carried out an analysis for each programming practice.

6.1 Collaboration

Figure 5.1 heatmap analysis easily reveals a clear difference in the level of collaboration between the three programming practices: pair programming, congregate programming, and solo programming. Given the questions asked in the survey, solo programming obviously scores the minimum in this category. Therefore we discuss in depth only the differences obtained between pair and mob.

Taking a deeper dive into some distinct data points, the practice of actively participating in discussions and brainstorming sessions stands out. Pair programming shows a high level of active participation with an average score of 3.75, which indicates that this practice 'mostly' encourages active participation. This is illustrated in the cell corresponding to question 2 (Q2) and 'Pair' on the heatmap. In comparison, mob programming also performs well, with an average score of 3.75. This score is found in the cell under 'Mob' and Q2 on the heatmap. However, solo programming falls short with the lowest possible score of 1.75, suggesting no active participation. This score can be seen in the cell under 'Solo' and Q2 on the heatmap.

Another interesting aspect to discuss is offering constructive feedback and suggestions. This is addressed in the third question of our survey. Here, pair programming again scores the highest with an average score of 3. On the other hand, mob programming shows a moderate score of 2.75, reflecting a somewhat lesser extent of constructive feedback and suggestions within mob programming environments. Both scores can be seen in two cells corresponding to question 3 (Q3).

In the context of working together to solve any problems that arose, mob programming scores an impressive average of 3.75, implying that problem-solving is 'mostly' carried out collaboratively. This is visible in the cell corresponding to Q5 under 'Mob'. Pair programming also fares well, with an average score of 3, indicating moderate collaborative problem-solving. This score can be seen in the cell corresponding to Q5 under 'Pair'. Once again, solo programming lags with a score of 1, which can be seen at the intersection of Q5 and 'Solo'.

On the question of recognizing and celebrating the accomplishments of team members, we see an interesting flip in the trends. While pair programming had been leading in most of the other parameters, it scores lower in this criterion with an average of 2.75, suggesting that recognition of accomplishments happens only 'somewhat' to 'moderately'. This score can be seen in the cell corresponding to Q8 under 'Pair'. On the other hand, mob programming scores 'mostly' with an average of 3.5, implying a more celebratory culture within mob programming environments. This score can be found at the intersection of Q8 and 'Mob'.

Our investigation reveals significant consistency with the existing literature [1, 2, 3, 4, 7, 8] when comparing our findings. Our data demonstrate that pair programming and mob programming foster active participation and effective collaboration among team members [1, 2, 3, 7], which aligns with prior studies. Nevertheless, we discovered that the recognition of accomplishments in pair programming occurred less frequently, which is a deviation from some reports in the literature [7, 8].

6.2 Knowledge Sharing

Figure 5.2 presents some findings, one of which pertains to the degree of comfort colleagues feel in sharing code or ideas, as indicated by Q5 on the heatmap. Notably, both pair and mob programming fosters an environment where individuals feel at ease sharing, as evidenced by average scores of 4.5 ("yes, quite a lot") and 4.5 ("yes, a lot"), respectively, as can be observed in the 'Pair' and 'Mob' columns of Q5. Conversely, solo programming, reflected in the 'Solo' column of Q5, displays a slight uptick with an average score of 1.75 ("yes, but not much"). This suggests that even when working independently, coders may engage in sharing practices, such as code reviews or constructive feedback.

In Q3 of the heatmap, a noteworthy shift is observed regarding overcoming obstacles or challenges. While the practice of pair programming remains prominent with an average score of 3.5 ("yes, somewhat"), as evidenced in the 'Pair' column of Q3, solo programming, depicted in the 'Solo' column of Q3, exhibits a discernible increase to 1.75 ("yes, but not much"). This may indicate individual problem-solving capabilities and self-reliance in surmounting impediments during solitary work.

The question concerning the acquisition of new programming concepts or techniques from colleagues, denoted as Q6 in the heatmap, presents a captivating contrast. While mob programming, seen in the 'Mob' column of Q6, leads with an average score of 3.25 ("yes, somewhat"), pair programming, as reflected in the 'Pair' column of Q6, demonstrates a slightly lower score of 2.25 ("yes, but not much"). Solo programming, in the 'Solo' column of Q6, evidences a modest improvement, averaging a score of 2 ("Yes, but not much"). This suggests that solo programmers may be resorting to external resources such as online forums, blogs, or documentation, to acquire new techniques and concepts.

Regarding conveying technical concepts or solutions to colleagues, indicated as Q8 on the heatmap, the scores for mob programming (4.25, 'Yes, a lot') and pair programming (3.5, 'Yes, somewhat') imply that these practices inherently foster peer instruction. Solo programming, with a slightly enhanced score of 1.5 ("yes, but not much"), located in the 'Solo' column of Q8, could suggest occasional instances of knowledge-sharing within a broader team context.

Our research not only reinforces the viewpoints proposed by Beck [2] and Hannay et al. [1] concerning knowledge sharing within programming teams, with specific emphasis on pair and mob programming developing techniques, but also sheds light on the potential for knowledge sharing in solo programming. This aspect, which has been largely overlooked in the current literature [8, 11], presents a compelling opportunity for further exploration and understanding.

6.3 Stress

In Figure 5.3, a depiction of stress can be observed. Notably, Q2 on the heatmap highlights the feeling of being unable to control crucial aspects of life. The score for pair programming remains at 2, indicating that pair programmers experience this feeling 'almost never,' as evidenced in the 'Pair' column of Q2. Similarly, mob programming is reflected in the 'Mob' column of Q2 scores 1, suggesting that individuals in these environments 'never' feel out of control. However, solo programmers now score an average of 1, implying that they too 'almost never' feel unable to control important aspects of their lives. This significant change from the previous data suggests a decrease in perceived stress in the solo programming environment for this particular aspect, as evident in the 'Solo' column of Q2.

Upon investigating Q1, which pertains to being upset due to unexpected events, the scores for all programming practices have undergone changes. Solo programmers now score an average of 2 ('almost never') as seen in the 'Solo' column of Q1, which is significantly lower than before. This suggests a decrease in unpredictability or an improvement in handling such situations. Pair programmers maintain a score of 2 ('almost never'), as demonstrated in the 'Pair' column of Q1. The mob programmers continue to score 1 ('never'), located in the 'Mob' column of Q1, indicating very low stress levels related to unexpected events in these environments.

The responses to question six, which concerns coping with tasks at hand, have exhibited significant changes. Specifically, solo programmers now report a score of 3 ('sometimes') in the 'Solo' column of Q6, indicating that they experience a certain degree of overwhelm in relation to their workload. This score represents a decrease in perceived stress levels as compared to previous data. Conversely, mob programmers, as reflected in the 'Mob' column of Q6, have reported a score of 2.8, which suggests that they 'fairly often' feel unable to cope with the tasks at hand. This finding suggests an increase in stress levels as compared to the previous data. Pair programmers maintain their previous score of 2 ('almost never'), as indicated in the 'Pair' column of Q6.

Regarding Q10 on the heatmap, which explores the notion of difficulties piling up to such an extent that they cannot be overcome, solo programmers now report a score of 5 ('very often'), which is consistent with the previous data. This finding suggests that solo programmers, as evidenced in the 'Solo' column of Q10, often feel overwhelmed by the accumulation of difficulties. On the other hand, pair and mob programmers' scores remain low, averaging 2 ('almost never') and 1 ('never'), respectively, as shown in the 'Pair' and 'Mob' columns of Q10. This indicates that these environments continue to offer superior stress management in this respect.

Our observations concur with Rostaher and Hericko's [10] assertion that collaborative programming environments such as pair and mob programming may lead to reduced stress levels. Our study, however, also reveals a substantial reduction in perceived stress in solo programming environments, which is not frequently highlighted in the literature [8, 10].

6.4 Productivity and Efficiency

Tables 5.1 and 5.2 are a description of productivity and efficiency. For mobbing, brainstorming, implementation, and testing tasks are accomplished successfully, leading to a total score of 2.5 (considering task difficulties). The efficiency is measured as five comments and no bugs. It means the quality of work done in the mob programming setting is quite good, as no bugs were introduced, and the comments were related to improvements rather than issues.

In terms of pair programming, there are two groups (Group 1 and Group 2). In the first group, all tasks are accomplished, with a difficulty-adjusted score of 1.5. The efficiency is lower than mob programming, with one bug found and one comment. In the second group, the testing task was not accomplished, yielding a difficulty-adjusted score of 1.2. However, there were no comments or bugs, which indicates a better performance than the first group in terms of the quality of work.

The last one is solo programming. There are four groups. All groups accomplished the brainstorming task. The implementation task was not accomplished in the fourth group, and testing was only successful in the second group. The difficulty-adjusted total score is 3.2, which is higher than both mob and pair programming. However, the efficiency is considerably lower, with 17 comments and 2 bugs.

Our findings align with Nosek's [2] and Müller's [11] studies, which suggest that collaborative programming methods, such as pair and mob programming, often result in higher productivity and fewer bugs. Interestingly, our investigation also reveals that solo programming can yield higher task completion rates, contradicting common perceptions in the literature, although it compromises efficiency with a higher rate of comments and bugs [11].

7 Discussion

The research objectives of this study were to comprehensively understand the key disparities between solo, pair, and mob programming practices, with respect to productivity, effectiveness, teamwork, knowledge distribution, and work stress levels. In this case, our experiments provide practical help in solving these problems.

In terms of collaborative efforts, both pair and mob programming act as catalysts for an enhanced degree of teamwork as compared to individual programming, as per the established understanding and previous research. The results of our investigation stand as a testament to previous research [1] [2] [4], with our current observations mirroring the established understanding that these programming methodologies are inherently intended to promote cooperative work. It is worth noting that our research has also illuminated the subtleties in the recognition of achievements within these practices. Notably, mob programming distinguishes itself with its more appreciative culture as opposed to pair programming. As such, our study contributes to the wider academic discourse by reaffirming that each programming approach can cultivate a unique team dynamic and working environment.

Knowledge sharing is another crucial aspect that our study explored. Our data demonstrated that pair and mob programming significantly encourage sharing ideas and code, consistent with studies by Hannay et al. [1] and Williams et al. [2]. Unexpectedly, we also found that solo programmers also showed instances of knowledge sharing, possibly driven by mechanisms like code reviews or asynchronous feedback or by using external resources for learning.

The intricacies of work-related stress across programming practices were analyzed, unveiling unique insights. Solo programmers demonstrated less stress when managing significant aspects of their work, implying that solo programming can offer greater autonomy and less imposed pressure. On the flip side, mob programmers displayed a heightened stress level tied to task processing, even with shared responsibilities. While solo programmers showcased an enhanced ability to handle unexpected events and task-related stress, they continued to grapple with stress when confronted with increased task difficulty. Pair and mob programming environments, however, consistently excelled in overall stress management, which was substantiated by the consistently lower stress scores.

The productivity and efficiency of teams differed across the practices. Mob programming demonstrated excellent efficiency and quality; pair programming results varied based on group dynamics; and solo programming, despite a higher total score, exhibited lower efficiency. This underscores the complex interplay between task nature, team dynamics, and programming practices in influencing productivity and efficiency outcomes.

Based on these findings, the choice of programming practice solo, pair, or mob depends on the task's nature, the desired outcomes (like collaboration, knowledge sharing, stress management, and productivity), and the dynamics of the team. There isn't a universally optimal programming practice; it's crucial to understand the strengths and limitations of each and select the one that best aligns with the team's needs and project requirements.

8 Conclusions and Future Work

Our study shows that collaboration is more evident in pair and mob programming, whereas solo programming lacks in this aspect. Interestingly, mob programming showed a more appreciative culture, particularly in recognizing and celebrating accomplishments. In terms of knowledge sharing, both pair and mob programming environments are encouraged with sharing of ideas and code, yet, surprisingly, knowledge sharing also occurs in solo programming, underscoring the value of consulting coworkers and interacting with them in the workplace. When managing work stress, a trade-off between solo and mob programming became apparent. Solo programmers experienced less stress in being in control of important aspects of their lives, suggesting the benefits of independence afforded them greater control over their ideas. However, mob programmers, despite the shared responsibilities inherent in the approach, reported an increase in stress related to task management. In terms of productivity, mob programming exhibited a superior quality of work even though only one person was operating, while pair programming exhibited variability based on group dynamics. Single-person programming was inefficient despite scoring high on adjusted productivity, indicating poor code quality.

The objective of this study was to address the following research questions:

- What are the primary differences in productivity and effectiveness between solo, pair, and mob programming practices?

According to the productivity scores, solo programming achieved the highest score, followed by pair programming and mob programming. This finding contradicts the conclusions drawn from the literature review, which suggests that mob programming is the most productive approach. It is possible that the varying levels of experience among the developers involved contributed to this discrepancy. In terms of efficiency, mob programming obtained the highest score, as expected, due to the increased number of participants observing the implementation process. On the other hand, solo programming obtained the lowest score in terms of efficiency.

- How do different programming practices affect individual developers' overall experience regarding team cooperation, knowledge distribution and working stress level?

The most satisfying developer's experience can be attributed to mob programming, particularly in terms of knowledge sharing and low stress levels. However, it is worth noting that pair programming achieved satisfying results as well, even surpassing mob programming by a small margin in terms of collaboration. This can be attributed to the fact that participants had more opportunities to actively engage with the code in pair programming compared to group programming. Conversely, solo programming performed poorly in these metrics.

The limitations of this study necessitate further research, despite the fact that it provides some initial answers and paves the way for future exploration. One such limitation is that our literature review, while comprehensive and detailed, does not qualify as a systematic literature review due to the absence of predefined inclusion and exclusion criteria and the lack of systematic coding and analysis. In light of these limitations, a more in-depth investigation into the long-term consequences of various programming techniques on developers could be beneficial. Additionally, broadening the scope to encompass more diverse teams, larger projects, and a variety of organizational cultures may help provide a more comprehensive understanding of how certain approaches function in different settings. Future studies could also explore the role of various programming techniques in relation to their efficacy and applicability to participants of different programming skill levels. It's important to consider that this study relied on self-reported data, such as job stress levels and perceptions of teamwork and knowledge distribution, which are inherently subjective and may be affected by factors such as participant bias and interpretation. Moreover, despite efforts to ensure that tasks were of similar complexity, there may still be variability in the tasks assigned under each programming method, potentially influencing the results. Finally, according to Thomas et al. [17] from the University of Wales, students who had the least confidence in themselves preferred pair programming the most, and the majority of students with higher ability levels opted not to pair up with peers who had lower competence levels, a finding that could have implications for the selection of programming methods in educational and professional settings.

Finally, the study revealed several interesting outcomes that demand more research. For instance, despite the possibility of unpredictability and their significant effort, why do solitary programmers appear to be less stressed while managing essential elements of their lives? Why do mob programmers experience higher levels of stress due to juggling tasks while having shared responsibilities? Future studies might explore these paradoxes in more depth to better understand how work habits affect productivity and well-being in the technology industry.

References

[1] Hannay, Jo E., et al. "The effectiveness of pair programming: A meta-analysis." Information and software technology 51.7 (2009): 1110-1122.
https://doi-org.proxy.lnu.se/10.1016/j.infsof.2009.02.001

[2] Williams, Laurie, et al. "Strengthening the case for pair programming." IEEE software 17.4 (2000): 19-25.
https://ieeexplore-ieee-org.proxy.lnu.se/abstract/document/854064

[3] Zuill, Woody, and Kevin Meadows. "Mob programming: A whole team approach." Agile 2014 Conference, Orlando, Florida. Vol. 3. 2016.
https://www.agilealliance.org/wp-content/uploads/2015/12/ExperienceReport.2014.Zuill_.pdf

[4] Ståhl, Daniel, and Torvald Mårtensson. "Mob programming: From avant-garde experimentation to established practice." Journal of Systems and Software 180 (2021): 111017.
https://doi-org.proxy.lnu.se/10.1016/j.jss.2021.111017

[5] Aune, Ole Kristian, Christian Echtermeyer, and Elias Sørensen. "Mob programming: A qualitative study from the perspective of a development team." Research Gate (2018).
https://www.researchgate.net/publication/328150167_Mob_Programming_A_Qualitative_Study_from_the_Perspective_of_a_Development_Team

[6] Dragos, Lucian. "Mob vs Pair: Comparing the two programming practices-a case study." (2021).
https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1578097&dswid=1991

[7] Kattan, Herez Moise, et al. "Swarm or pair? strengths and weaknesses of pair programming and mob programming." Proceedings of the 19th International Conference on Agile Software Development: Companion. 2018.
https://dl-acm-org.proxy.lnu.se/doi/abs/10.1145/3234152.3234169

[8] Roque Hernández, Ramón Ventura, Sergio Armando Guerra Moya, and Adán López Mendoza. "Solo, pair or Mob programming: Which should be used in university?." Apertura (Guadalajara, Jal.) 12.1 (2020): 39-55.
https://doi.org/10.32870/ap.v12n1.1791

[9] European Union. "Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)." Official Journal of the European Union, L119 (2016): 4-88.

[10] Lee, Eun-Hyun. "Review of the psychometric evidence of the perceived stress scale." Asian nursing research 6.4 (2012): 121-127.
https://doi-org.proxy.lnu.se/10.1016/j.anr.2012.08.004
https://fbanken.se/files/220/PSS-10-Pre-final_Admin_Form-Swedish.pdf

[11] Müller, Matthias M. "A preliminary study on the impact of a pair design phase on pair programming and solo programming." Information and software Technology 48.5 (2006): 335-344.
https://doi-org.proxy.lnu.se/10.1016/j.infsof.2005.09.008

[12] C. Lilienthal, "From Pair Programming to Mob Programming to Mob Architecting," in D. Winkler, S. Biffl, and J. Bergsmann (eds.), Software Quality: Complexity and Challenges of Software Engineering in Emerging Technologies, SWQD 2017, Lecture Notes in Business Information Processing, vol. 269, Springer, Cham, 2017, pp. 1-10. [Online]. Available:
https://doi-org.proxy.lnu.se/10.1007/978-3-319-49421-0_1

[13] M. Shiraishi, H. Washizaki, Y. Fukazawa, and J. Yoder, "Mob Programming: A Systematic Literature Review," in 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), Milwaukee, WI, USA, 2019, pp. 616-621.
https://ieeexplore-ieee-org.proxy.lnu.se/abstract/document/8753993

[14] H. Kattan. "Software Development Practices Patterns: from Pair to Mob Programming", in Proceedings of the 3rd Regional School of Software Engineering, Rio do Sul, 2019, pp. 147-156.
https://sol.sbc.org.br/index.php/eres/article/view/8507

[15] G. Saake, S. Apel, and I. Schaefer, “Measuring and Improving Code Quality in Highly Configurable Software Systems DISSERTATION zur Erlangung des akademischen Grades,” Uni-halle.de. [Online]. Available:
https://opendata.uni-halle.de/bitstream/1981185920/34942/1/Fenske_Wolfram_Dissertation_2020.pdf.
[Accessed: 19-May-2023]

[16] A. Begel and N. Nagappan, “Pair programming: What’s in it for me?,” in Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, 2008.
https://dl.acm.org/doi/abs/10.1145/1414004.1414026?casa_token=K7pjqWqmKLAAAAAA:tD51LDIS7JpTzczqrm75YvHlj7yWzQRtgF-Dq9uFY5epUQm40Iy3OxrGf3Gw9JQfaJIsd-RfaqY

[17] N. Nagappan, L. Williams, E. Wiebe, C. Miller, S. Balik, M. Ferzli, and J. Petlick, "Pair Learning: With an Eye Toward Future Success," in Proceedings of the 2003 International Conference on Software Engineering (ICSE '03), Portland, OR, USA, 2003, pp. 185-198. doi: 10.1109/ICSE.2003.1201248.
https://www.researchgate.net/publication/221592681_Pair_Learning_With_an_Eye_Toward_Future_Success

[18] K. L. Wuensch, What is a Likert Scale? and How Do You Pronounce “Likert?” East Carolina University, 2005.