Classification of human-human and human-AI pair programming effects and expansion for AI pair programming patterns

YUMA TAKAI, Waseda University
HIRONORI WASHIZAKI, Waseda University
NOBUKAZU YOSHIOKA, Waseda University
YOSHIAKI FUKAZAWA, Waseda University

Many researchers explored the contributions of both human-human and human-AI pair programming. We categorized the previous research on pair programming and reveals four primary domains of influence: Code, Coding, Coder, and Pair (AI). AI pair programming primarily enhances code readability, test coverage, and development speed, while human-human pair programming enhances knowledge sharing, problem-solving, and individual motivations. However, the programming experience and outcomes are contingent upon the proficiency level of the programmers involved. In addition, we show a pattern of pair programming using AI. This highly abstract pattern will serve as a basis for describing multiple concrete patterns in the future. This pattern reduces the risk of using Large Language Models(LLMs) and helps developers to develop more efficiently and with higher quality.

Categories and Subject Descriptors: D.2.3 [**Software**]: Software Engineering—*Coding Tools and Techniques*

General Terms: AI Pair Programming

Additional Key Words and Phrases: Pair programming, Pattern

1. INTRODUCTION

The field of software development is rapidly evolving. A notable change is the emergence of AI pair programming. This offers a collaborative programming experience in which human developers and AI work together to write code that goes beyond the boundaries of conventional programming. Introducing this technology may realize a more sophisticated approach to code generation and even revolutionize the development process.

Typical services for AI pair programming include Copilot and ChatGPT, which are based on GPT-3 and GPT-4 developed by OpenAI, as well as CodeWhisperer, provided by Amazon. Each service leverages AI capabilities to assist developers with more efficient and higher-quality code generation.

Another popular approach is "pair programming," in which humans are paired with other humans to code. In this approach, the roles of the "driver" (the person who writes the code) and the "navigator" (the person who supervises the code and provides feedback) are alternated between the pair. Pair programming improves code quality and promotes knowledge sharing and elicitation of new ideas and solutions [Beck 2000] [Williams and Kessler 2003].

Author's address: Yuma Takai, 3-4-1 Okubo, Shinjuku Ward, Tokyo, Japan; email: marufura@fuji.waseda.jp;
H. Washizaki, 3-4-1 Okubo, Shinjuku-ku, Tokyo, Japan; email: washizaki@waseda.jp;

However, pair programming has some challenges, including communication issues, time and effort costs, and differences in technological levels.

Although human and AI pair programming share the fundamental aspect of writing code, their programming experiences differ. AI pair programming provides advanced code recommendations using machine learning techniques. In contrast, human pair programming enables complex problem-solving by exchanging ideas, experiences, and perspectives.

This study analyzes the impact of different programming approaches on the overall programming experience. First, we summarize past research on pair programming and categorize critical elements. We then propose abstract patterns that can be used to describe specific patterns using AI pair programming. From these results, we can present various patterns related to AI pair programming and proficiency.

2. BACKGROUND

This section presents recent and relevant research surveys on large language models (LLM) in software engineering. Hou et al. [2023] collected and analyzed 229 research papers from 2017 to 2023 on the current status of how LLM are being used and optimized in tasks in software engineering. According to the survey, LLM is most widely used in software development, while the least applied area is software management. Code generation and program repair are the most common tasks that employ LLM in software development and maintenance activities. In addition, Zheng et al. [2023] has collected the widest possible range of relevant literature from seven major databases and selected 123 articles for analysis and classification to analyze the current state of LLM research regarding seven major software engineering tasks.

These are the definitions of the seven major software engineering tasks and the role of LLM:

Code Generation: Automatic generation of source code based on user requirements and specified constraints; LLM is used to generate code, provide developers with a "starting point" for ideas and programming, and for other auxiliary purposes.

Code Summarization: Automatic generation of clear, accurate, and useful code comments to help developers understand and maintain code; LLM summarizes code to support different granularities (e.g., functions) or to explain the intent of the code.

Code Translation: The translation of code between different programming languages without changing its functionality or logic; in LLM, it is studied in auxiliary code translation and reverse engineering.

Vulnerability Detection: Identifying and correcting code errors that cause program crashes, performance degradation, and security problems; LLM checks code for potential flaws.

Code Evaluation: Static analysis of code to identify potential problems and areas for improvement; LLM creates test cases, and tests code for performance, usability, and other quality characteristics.

Code Management: Managing code versions and developer information; LLM involves team-based collaborative development and version control.

Q&A Interaction: Interaction between software developers and LLM, which in LLM is studied in the form of assistant and prompt engineering.

In this research, we assume a pattern in which an individual performs AI pair programming in accordance with some requirements at the development stage to complete the desired code. In doing so, Code Generation, Code Summarization, Vulnerability Detection, and Code evaluation were treated.

3. PAIR PROGRAMMING CATEGORIZATION

Pair programming encompasses multifaceted effects. These effects require systematic classification and organization. This is challenging as it difficult to assess the benefits and to effectively identify best practices.

Our study classifies the impacts of pair programming into four interrelated domains, representing various facets of software development. These domains encompass tangible outputs such as code quality and productivity as well as intangible aspects such as skill enhancement and collaborative benefits. Table I provides a multi-dimensional perspective on the implications of pair programming, capturing its diverse influences across the codebase, development process, individual developers, team dynamics, and even AI-driven pair programming scenarios.

These classifications were formed by manually reviewing the literature and extracting information relevant to each domain. Since AI pair programming is an emerging field, we prioritized more recent literature to reflect the latest research findings. Consequently, this study comprehensively captures the effects of pair programming and provides a foundation for understanding its utility and potential in a more profound sense.

Table I. Impact Domains of Pair Programming

Domain	**Description**
Code	Influence of pair programming on code readability, test coverage, and security.
Coding	Impact on development workflow, speed of code production, and total code output.
Coder	Contributions to coder's skill enhancement, knowledge acquisition, and overall productivity.
Pair (AI)	How pair programming benefits AI or the team itself in aspects like improved collaboration and problem-solving.

Table II summarizes the contributions made by AI pair programming and human-to- human pair programming across the four domains. Numerous studies on AI pair programming have investigated the generated code quality and the code creation efficiency. Some studies evaluated the AI pair programming related to education, although experimental reports are less prevalent.

For human-to-human pair programming, the reported impacts on the 'Code' and 'Coding' domains vary. Some works identify positive effects, while others found no notable gains. Furthermore, the 'Coder' and 'Pair' aspects show significant contributions such as boosting motivation and facilitating knowledge sharing. However, these aspects have yet to be prominently discussed in AI pair programming, revealing the potential of human-to-human pair programming.

Table II. Comparison of the current state of Human-AI and Human-Human Pair Programming in the four domains

	Human-AI (H-AI)	**Human-Human (H-H)**
Code	Useful for experts, less so for beginners.	Quality improvements such as readability and fewer reported, although there are also negative reports.
Coding	Provides a good starting point and increases code volume but may consume more time in case of deficiencies in the answers.	Opinions vary on whether code is efficiently produced.
Coder	There are many challenges such as references not being shown and inaccurate responses.	Pair programming enhances task satisfaction and reduces slacking due to a sense of responsibility when working as a pair.
Pair (AI)	Pair programming allows the AI to learn for better or worse.	Knowledge sharing increases the bus factor and enables backup.

3.1 Code (Human-Human)

The effectiveness of pair programming among humans remains controversial. Some argue that it produces fewer bugs and high-quality code [Begel and Nagappan 2008], whereas others claim that the benefit of pair programming is exaggerated in the literature [Hulkko and Abrahamsson 2005]. Furthermore, working in pairs does not necessarily outperform individuals working alone [Balijepally et al. 2009].

3.2 Code (Human-AI)

The applicability of AI in improving code quality continues to be a subject of debate. On the one hand, several studies, including [Bird et al. 2023], argue that the automated generation of tests by AI can significantly enhance both the quality of code and the reliability of downstream distributed systems. Similarly, Górecki [2023] highlights that collaboration with AI can yield substantial benefits, provided common pitfalls are skillfully avoided. Nevertheless, not all empirical evidence supports this optimistic view. For example, Imai [2022] found that using AI in pair programming resulted in lower-quality code in their experimental setup.

Incorporating the user's proficiency level adds another layer of complexity to this discussion. Expert developers are likely to benefit from AI, as the quality of AI-generated code tends to be comparable to that of human experts. The impact of LLMs on novice developers can be both positive and negative. It has been shown that novice developers were able to develop more efficiently by using LLM [Noy and Zhang 2023] [Brynjolfsson et al. 2023] [Ma et al. 2023]. On the other hand, the limitations inherent in AI's incomplete abstraction may present a hurdle [Sarkar et al. 2022]. Novice developers may need additional support in filtering out bugs or suboptimal solutions, thus making AI a potential source of technical debt [Dakhel et al. 2023]. If users lack proficiency or uncritically accept AI-generated outputs, the resulting code could suffer in quality and may even introduce significant vulnerabilities like security bugs. Ultimately, the perceived quality of the code varies depending on who is evaluating it, highlighting the multi-faceted nature of this issue.

This notion extends further when considering that code quality is not a one-dimensional attribute. As stipulated in ISO 25012, the quality of AI-generated code can be evaluated from multiple angles, such as application developer and its user perspectives [Zhang et al. 2022]. While developers may prioritize attributes like completeness and consistency, users might focus more on accessibility and availability.

3.3 Coding (Human-Human)

The assessment results vary in human-to-human scenarios. Theoretically, two individuals working together should double the efficiency. However, pair programming does not consistently yield superior productivity compared to solo programming [Hulkko and Abrahamsson 2005]. One challenge in pairs programming is scheduling working hours efficiently [Begel and Nagappan 2008]. Nevertheless, in some instances, problem-solving is expedited due to the high level of verbalization when pair programmers are compelled to rationally explain their decisions compared to coding alone [Alves De Lima Salge and Berente 2016].

3.4 Coding (Human-AI)

The efficiency of coding processes is deeply tied to two key factors: the completeness of the generated code and the quality of the prompts used. Research suggests that an increase in the number of added lines of code is indicative of heightened productivity [Imai 2022] [Noda et al. 2023] [Humble and Kim 2018]. However, this boost in productivity can be compromised if the generated code is fundamentally flawed or inefficient. In such cases, programmers may find themselves engaged in what can be termed a 'fruitless chase.' They may end up spending valuable time repairing or debugging the code, which often proves to be even more time-consuming than writing original code [Sarkar et al. 2022].

On the other hand, AI pair programming offers advantages in terms of scheduling flexibility. Unlike traditional human pair programming, where coordinating schedules can often be a hindrance to efficiency, AI eliminates this bottleneck. Nonetheless, it's worth noting that code quality and coding efficiency are interdependent. Specifically, if the AI-generated code lacks quality, programmers may need to invest additional time in revisions, thereby reducing overall productivity.

3.5 Coder (Human-Human)

While pair programming may not necessarily improve the quality of code produced, it does contribute to increased levels of psychological satisfaction [Balijepally et al. 2009]. Specifically, pair programming enhances focus during shared working sessions, as individuals are less likely to engage in distracting activities such as reading emails, web browsing, or making prolonged phone calls [Hannay et al. 2009].

Nevertheless, there are caveats. For example, a mismatch in communication skills or personalities could diminish the benefits, potentially leading to less positive outcomes [Begel and Nagappan 2008]. Moreover, suggestions made by navigators are not invariably of high quality; incorrect guidance can indeed hamper productivity.

In terms of task complexity, it seems that pair programming is more advantageous for complicated tasks. Conversely, simpler tasks may actually be completed more efficiently when undertaken individually.

The practice of pair programming is not just an isolated technique but is often integrated into pattern languages to foster knowledge sharing and code comprehension among team members [Weiss 2022]. This mutual exchange enables team members to substitute for one another as necessary. Therefore, pair programming serves dual purposes: it harmonizes technical knowledge across different teams within an organization [Santos et al. 2013] and fortifies effective communication. Ultimately, when there's a common understanding among all participants, risks of misunderstandings and subsequent declines in productivity are significantly reduced [Pinho and Aguiar 2020].

3.6 Coder (Human-AI)

Although the contributions of AI programming to coders are primarily seen in education and skill development, these contributions should be experimentally validated. The AI-generated response depends on the prompt. There is no guarantee that the same prompt will yield the same response, leading to an absence of completeness. Currently, coders must assess the correctness and incomplete aspects of the output, which significantly challenges their growth [Sarkar et al. 2022]. The generated response relies on what the model knows and not the human knowledge, resulting in a discrepancy from the ideal answer. Furthermore, environmental factors, particularly in educational settings, pose a risk of plagiarism since generated results are not cited or referenced, leading to possible regulations [Rahman and Watanobe 2023][Tlili et al. 2023].

3.7 Pair (Human-Human)

Benefits in pair are the same as in coder.

3.8 AI (Human-AI)

Pair programming allows the AI to learn for better or worse. If there is good feedback on the responses generated, it will receive positive reinforcement; conversely, if it stays in the wrong state, it will receive negative reinforcement. This reinforcement may affect not only current developers but also later users in terms of code quality and efficiency.

4. PATTERN

In light of these characteristics, we propose a high-level abstract pattern that will serve as the foundation for describing specific patterns in the future. We plan to offer multiple patterns for further AI use in the future based on the following pattern.

4.1 Name

Test driven AI coding

4.2 Intent

AI can generate and validate code. Supplementing areas that AI cannot cover with human intervention improves the efficiency and quality of development projects. (Contributing to Code, Coding using AI)

4.3 Context

Developers who are familiar with programming and development projects can utilize AI to accomplish tasks. Additionally, it is crucial to have the ability to translate abstract requirements into concrete assignments.

4.4 Problem

AI often give the wrong answer due to the abstract nature of what developers are looking for or potential dependencies that are not given in the prompt. This leads to further misunderstandings by the developer, and the issue becomes a complicated maze. However, without AI, efficiency and quality are compromised by the need to enter similar code over and over again, and humans tend to disregard important but tedious items such as comments.

The following forces are associated with this problem:

Productivity (Speed): Easy and straightforward but time-consuming tasks reduce the time available for considering crucial matters, leading to a decline in concentration and a negative impact on productivity due to tedious tasks.

Functional Suitability (Correction): The code may reach the code review stage at worst before one can get advice from others about their code. The absence of early verification and modification steps at a detailed granularity could mask the impact range of bugs and defects.

Explainability (Maintainability, Readability): In cases where code reviews are neglected, adequate documentation may not be prepared for the code, and variables or functions that can only be understood by the person who wrote the code could potentially be created. This could lead to a decrease in the maintainability and readability of the code.

4.5 Solution

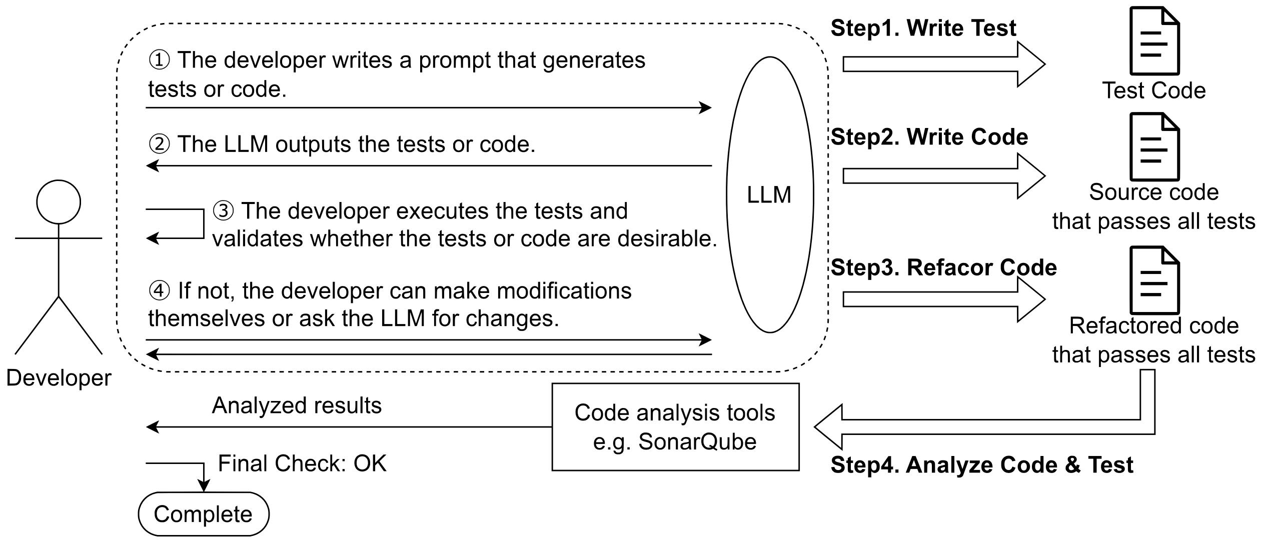
Introducing development by Test Driven Development (TDD) and divided into four steps: generating tests, generating code, refactoring code, and analyzing code. Along with these steps, prompts are given to the LLM. By using tests, it becomes possible to communicate to the AI what the correct answer is, thereby eliminating problems caused by ambiguity in the prompts. A unique feature is that both humans and AI can switch roles between being the driver and the navigator, adapting based on each other's level of understanding and strengths and weaknesses.

Step 1 (Write Test): In this step, the first task is to create tests. By presenting test cases, it becomes possible to set specific goals from abstract ones. Usually, basic test cases should be created by the AI, which shares the goal test cases with the AI, thereby reducing the time taken to create tests. However, human visual inspection and corrections become necessary for boundary conditions or ambiguous aspects that LLMs may not fully address. Specifying particular situations and constraints in test cases helps eliminate ambiguous conditions when writing actual code.

Step 2 (Write Code): At this stage, code is written, in conjunction with the AI, tailored to one's proficiency and strengths, to pass all the test cases set in step 1. If the test design is sufficient, the goals to be met in the test cases are correctly indicated, which helps maintain the reliability of the code. If tests are insufficient, it's necessary to return to step 1 for redesigning the tests. After creating the code, it is executed and modified to satisfy all the test cases. Please note that it's very risky if humans cannot understand the contents of the code. If the prompts are essentially incorrect, it may go unnoticed, potentially leading to wrong directions in discussions. LLMs have difficulties with environment-dependent issues (like file path settings, file encoding settings), and complex dependency problems. If humans do not understand the code, they may face these latent risks so it is beneficial to ask the LLM to explain it.

Step 3 (Refactor Code): Once code that passes all test cases is written, it is refactored to a form suitable for use in actual projects. This includes unifying coding conventions, redefining variable names, and dividing into multiple functions. If using an LLM integrated with a development environment like Copilot, it automatically makes these judgments, so less effort is needed for fine adjustments like naming conventions.

Step 4 (Analyze Code & Test): Finally, perform static analysis on the completed code to verify that the created tests and code are suitable for incorporation into the project, using tools like SonarQube [Marcilio et al. 2019]. If problems arise in this step, return to the relevant step for redesigning tests, re-modifying code, or refactoring code.



① The developer writes a prompt that generates tests or code.
② The LLM outputs the tests or code.
③ The developer executes the tests and validates whether the tests or code are desirable.
④ If not, the developer can make modifications themselves or ask the LLM for changes.
LLM
Developer
Step1. Write Test
Test Code
Step2. Write Code
Source code that passes all tests
Step3. Refacor Code
Refactored code that passes all tests
Step4. Analyze Code & Test
Code analysis tools e.g. SonarQube
Analyzed results
Final Check: OK
Complete

Fig. 1. Pattern solution overview

4.6 Example

Suppose that Developer is tasked with implementing a function to retrieve the five newest or oldest purchases from a list of purchase histories provided as a dictionary.

Step 1 (Write Test):

First, the Developer creates tests that define the clear goals for this occasion. It is beneficial to inform the LLM that the process is being carried out using test-driven development. If this is not communicated, the LLM often attempts to autonomously generate code that meets the requirements. This time, the developer uses ChatGPT.

```
I need Python code to select up to five newest and oldest date items from an unsorted list
    of dictionaries representing purchase history.
Developing by test driven development, please generate some test in Python firstly.
Sample purchase history data is like below.
"""
{'date': '2023-06-13', 'product': 'Apple', 'quantity': 5}
"""
```

Listing 1. The prompt to request test generation

```python
import unittest
from datetime import datetime

# Function to be implemented
def sort_purchase_history(purchases, newest=True, max_results=5):
    # This function will be implemented later
    pass

class TestSortPurchaseHistory(unittest.TestCase):

    def test_empty_list(self):
        self.assertEqual(sort_purchase_history([]), [])

    def test_single_entry(self):
        data = [{'date': '2023-06-13', 'product': 'Apple', 'quantity': 5}]
        self.assertEqual(sort_purchase_history(data), data)

    def test_multiple_entries_newest(self):
        data = [
            {'date': '2023-06-13', 'product': 'Apple', 'quantity': 5},
            {'date': '2023-06-15', 'product': 'Banana', 'quantity': 3},
            {'date': '2023-06-12', 'product': 'Cherry', 'quantity': 7}
        ]
        expected = sorted(data, key=lambda x: datetime.strptime(x['date'], '%Y-%m-%d'),
    reverse=True)[:5]
        self.assertEqual(sort_purchase_history(data), expected)

    def test_multiple_entries_oldest(self):
        data = [
            {'date': '2023-06-13', 'product': 'Apple', 'quantity': 5},
            {'date': '2023-06-15', 'product': 'Banana', 'quantity': 3},
            {'date': '2023-06-12', 'product': 'Cherry', 'quantity': 7}
        ]
        expected = sorted(data, key=lambda x: datetime.strptime(x['date'], '%Y-%m-%d'))[:5]
        self.assertEqual(sort_purchase_history(data, newest=False), expected)

    def test_five_entries(self):
        data = [
            {'date': '2023-06-11', 'product': 'Eggplant', 'quantity': 2},
            {'date': '2023-06-13', 'product': 'Apple', 'quantity': 5},
            {'date': '2023-06-15', 'product': 'Banana', 'quantity': 3},
            {'date': '2023-06-12', 'product': 'Cherry', 'quantity': 7},
            {'date': '2023-06-14', 'product': 'Date', 'quantity': 4}
        ]
        expected_newest = sorted(data, key=lambda x: datetime.strptime(x['date'], '%Y-%m-%d'
    ), reverse=True)
        expected_oldest = sorted(data, key=lambda x: datetime.strptime(x['date'], '%Y-%m-%d'
    ))
        self.assertEqual(sort_purchase_history(data), expected_newest)
        self.assertEqual(sort_purchase_history(data, newest=False), expected_oldest)

if __name__ == '__main__':
    unittest.main()
```

Listing 2. Generated test code from ChatGPT

Step 2 (Write Code):

After conducting a visual inspection of the tests, they appeared to be promising, so it was decided to write code that actually passes all test cases. This time, assuming a simple function with no environment-dependent issues, the developer has decided to have the LLM generate all the content.

```
OK, then let's generate the `sort_purchase_history` function that pass all tests.
```

Listing 3. The Prompt to request code generation

```python
def sort_purchase_history(purchases, newest=True, max_results=5):
    # Sort the purchases by date
    sorted_purchases = sorted(
        purchases,
        key=lambda x: datetime.strptime(x['date'], '%Y-%m-%d'),
        reverse=newest
    )

    # Return up to max_results entries
    return sorted_purchases[:max_results]
```

Listing 4. Generated code that pass all tests

Step 3 (Refactor Code):

Having created code that passes all tests, the next step is to refactor it. In the group that the developer belongs to, there is a rule that requires adding type hints to the functions used, so this will be done manually.

```python
def sort_purchase_history(
    purchases: List[Dict[str, str]],
    newest: bool = True,
    max_results: int = 5
) -> List[Dict[str, str]]:
    # Sort the purchases by date
    sorted_purchases = sorted(
        purchases,
        key=lambda x: datetime.strptime(x['date'], '%Y-%m-%d'),
        reverse=newest
    )

    # Return up to max_results entries
    return sorted_purchases[:max_results]
```

Listing 5. Refactored code

Step 4 (Analyze Code & Test):

Finally, the developer used SonarQube to confirm that the created code passed all the tests and that the code's cyclomatic complexity and cognitive complexity were not too high. Through this process, the code and tests were submitted as a pull request and then merged into the actual project code.

The Developer successfully implemented a function to retrieve the latest, or the five oldest, purchase histories from the purchase history. The advantages of this pattern include firstly allowing the LLM to understand what is intended to be created by creating test cases. Then, by generating code that matches the test cases in step 2 (Write Code), the desired code can be obtained without issues, even with ambiguous prompts. Although this was a simple example, for code that cannot be handled by LLM generation alone or for developments that require complex refactoring, human correction is necessary. In addition, dividing the development into steps makes it easier to notice mistakes. A common pattern is to provide what you want to create as a prompt and receive the output, but if the desired product is abstract, the cases where it outputs successfully are few, and it takes time to verify whether it is actually correct. Additionally, if there are errors in the output, it takes even more time to identify the cause of the error. In this pattern, by conducting development in at least four steps and proceeding in stages, verification can be done quickly and easily, making it easier to notice mistakes.

4.7 Consequences

The pattern presents the following advantages:

Productivity (Speed): By delegating trivial tasks to an LLM, programmers can focus on more advanced tasks. Examples: Review whether the algorithm is the best choice, Boundary condition test verification.

Functional Suitability (Correction): The introduction of tests transforms abstract goals into concrete test cases. As a result, the LLM starts to create code that aligns with these test cases. Furthermore, AI-based checks increase the likelihood of identifying previously unnoticed issues and human errors.

Explainability (Maintainability, Readability): AI takes care of even trivial and bothersome tasks. While humans tend to neglect trivial and tedious tasks, using AI-generated solutions can help maintain high maintainability and readability.

5. FUTURE WORK

Herein the effects of pair programming are classified and organized into patterns. In the future, we would like to investigate how the use of AI differs between advanced coders and beginners to refine these patterns. The ratio of driver and navigator role assignments may differ in each case. In addition, we intend to propose patterns in AI pair programming in education and skill development situations.

6. CONCLUSION

The effects of AI pair programming and human pair programming are classified based on where they exert their effects. AI pair programming significantly contributes to Code and Coding, while human pair programming greatly impacts psychological aspects and knowledge-sharing effects (Coder, Pair). The effects of pair programming depend on the proficiency level of the developer. By using the pattern that uses Test Driven Development, developers can develop more effectively and with better quality.

Acknowledgement

We are grateful to our shepherd, Michael Weiss, for his careful and valuable reviews that significantly improved this paper. Also, We would like to thank all participants of the writers' workshop at PLoP 2023.

REFERENCES

Carolina Alves De Lima Salge and Nicholas Berente. 2016. Pair Programming vs. Solo Programming: What Do We Know After 15 Years of Research?. In *2016 49th Hawaii International Conference on System Sciences (HICSS)*. 5398–5406. DOI:http://dx.doi.org/10.1109/HICSS.2016.667

VenuGopal Balijepally, RadhaKanta Mahapatra, Sridhar Nerur, and Kenneth H. Price. 2009. Are Two Heads Better than One for Software Development? The Productivity Paradox of Pair Programming. *MIS Quarterly* 33, 1 (2009), 91–118. `http://www.jstor.org/stable/20650280`

Kent Beck. 2000. *Extreme programming explained: embrace change.* addison-wesley professional.

Andrew Begel and Nachiappan Nagappan. 2008. Pair Programming: What's in It for Me?. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '08)*. Association for Computing Machinery, New York, NY, USA, 120–128. `DOI:http://dx.doi.org/10.1145/1414004.1414026`

Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. 2023. Taking Flight with Copilot: Early Insights and Opportunities of AI-Powered Pair-Programming Tools. *Queue* 20, 6 (jan 2023), 35–57. `DOI:http://dx.doi.org/10.1145/3582083`

Erik Brynjolfsson, Danielle Li, and Lindsey R Raymond. 2023. *Generative AI at Work*. Working Paper 31161. National Bureau of Economic Research. `DOI:http://dx.doi.org/10.3386/w31161`

Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, Zhen Ming, and Jiang. 2023. GitHub Copilot AI pair programmer: Asset or Liability? (2023).

Jan Górecki. 2023. Pair Programming with Large Language Models for Sampling and Estimation of Copulas. (2023).

Jo E. Hannay, Tore Dybå, Erik Arisholm, and Dag I.K. Sjøberg. 2009. The effectiveness of pair programming: A meta-analysis. *Information and Software Technology* 51, 7 (2009), 1110–1122. `DOI:http://dx.doi.org/https://doi.org/10.1016/j.infsof.2009.02.001` Special Section: Software Engineering for Secure Systems.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. (2023).

Hanna Hulkko and Pekka Abrahamsson. 2005. A Multiple Case Study on the Impact of Pair Programming on Product Quality. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. Association for Computing Machinery, New York, NY, USA, 495–504. `DOI:http://dx.doi.org/10.1145/1062455.1062545`

Jez Humble and Gene Kim. 2018. *Accelerate: The science of lean software and devops: Building and scaling high performing technology organizations*. IT Revolution.

Saki Imai. 2022. Is GitHub Copilot a Substitute for Human Pair-programming? An Empirical Study. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 319–321. `DOI:http://dx.doi.org/10.1145/3510454.3522684`

Qianou Ma, Tongshuang Wu, and Kenneth Koedinger. 2023. Is AI the better programming partner? Human-Human Pair Programming vs. Human-AI pAIr Programming. (2023).

Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto. 2019. Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 209–219. `DOI:http://dx.doi.org/10.1109/ICPC.2019.00040`

Abi Noda, Margaret-Anne Storey, Nicole Forsgren, and Michaela Greiler. 2023. DevEX: What Actually Drives Productivity? *Commun. ACM* 66, 11 (oct 2023), 44–49. `DOI:http://dx.doi.org/10.1145/3610285`

Shakked Noy and Whitney Zhang. 2023. Experimental evidence on the productivity effects of generative artificial intelligence. *Science* 381, 6654 (2023), 187–192. `DOI:http://dx.doi.org/10.1126/science.adh2586`

Daniel Pinho and Ademar Aguiar. 2020. The AgilECo Pattern Language: Physical Environment. In *Proceedings of the European Conference on Pattern Languages of Programs 2020 (EuroPLoP '20)*. Association for Computing Machinery, New York, NY, USA, Article 30, 9 pages. `DOI:http://dx.doi.org/10.1145/3424771.3424790`

Md. Mostafizer Rahman and Yutaka Watanobe. 2023. ChatGPT for Education and Research: Opportunities, Threats, and Strategies. *Applied Sciences* 13, 9 (2023). `DOI:http://dx.doi.org/10.3390/app13095783`

Viviane Santos, Alfredo Goldman, Eduardo Guerra, Cleidson De Souza, and Helen Sharp. 2013. A Pattern Language for Inter-Team Knowledge Sharing in Agile Software Development. In *Proceedings of the 20th Conference on Pattern Languages of Programs (PLoP '13)*. The Hillside Group, USA, Article 20, 21 pages.

Advait Sarkar, Andrew D. Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence? (2022).

Ahmed Tlili, Boulus Shehata, Michael Agyemang Adarkwah, Aras Bozkurt, Daniel T. Hickey, Ronghuai Huang, and Brighter Agyemang. 2023. What if the devil is my guardian angel: ChatGPT as a case study of using chatbots in education. *Smart Learning Environments* 10, 1 (22 Feb 2023), 15. `DOI:http://dx.doi.org/10.1186/s40561-023-00237-x`

Michael Weiss. 2022. Patterns for Managing Remote Software Projects. In *Proceedings of the 27th Conference on Pattern Languages of Programs (PLoP '20)*. The Hillside Group, USA, Article 20, 8 pages.

Laurie Williams and Robert R Kessler. 2003. *Pair programming illuminated*. Addison-Wesley Professional.

Di Zhang, Mohd Anuaruddin Bin Ahmadon, and Shingo Yamaguchi. 2022. Human-AI Pair Programming by Data Stream and Its Application Example. In *2022 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*. 1–4. DOI:http://dx.doi.org/10.1109/ICCE-Asia57006.2022.9954649

Zibin Zheng, Kaiwen Ning, Jiachi Chen, Yanlin Wang, Wenqing Chen, Lianghong Guo, and Weicheng Wang. 2023. Towards an Understanding of Large Language Models in Software Engineering Tasks. (2023).