ENHANCING COLLABORATION AND CODE QUALITY USING PAIR PROGRAMMING

Gustavo de la Cruz Martínez and Selene Marisol Martínez Ramírez

Instituto de Ciencias Aplicadas y Tecnología, Universidad Nacional Autónoma de México, Circuito Exterior S/N, Ciudad Universitaria, 04510, Mexico City

ABSTRACT

Pair programming, a fundamental practice in Extreme Programming and agile methodologies, is widely recognized for enhancing collaboration, improving code quality, and promoting knowledge sharing. This article explores the principles, benefits, and challenges of pair programming across traditional, hybrid, and large-scale agile environments. Drawing from empirical studies, case analyses, and real-world implementations, it highlights how pair programming fosters teamwork, accelerates problem-solving, and ensures adherence to coding standards. The effectiveness of this practice is influenced by factors such as task complexity, developer expertise, and alignment of team goals. In hybrid work settings, modern tools facilitate real-time collaboration, bridging gaps between in-person and remote participants. Despite challenges such as increased effort costs, role ambiguity, and technical barriers, pair programming remains a flexible and valuable methodology for achieving high-quality, maintainable software. The article underscores the importance of adapting pair programming practices to specific team dynamics and evolving work environments to maximize its impact on software development.

KEYWORDS

Pair Programming, Code Quality, Hybrid Work, Software Engineering Practices.

1. INTRODUCTION

Pair programming, a cornerstone of agile development methodologies like Extreme Programming (XP), emphasizes teamwork to improve code quality and promote knowledge sharing among developers. As software development evolves into more distributed and hybrid environments, pair programming is a valuable practice, adapting through innovative tools and methods.

Drawing from insights from recent research and case studies, this article explores the best practices, challenges, and tools for modern pair programming, focusing on its application in both traditional and hybrid work settings.

2. AGILE DEVELOPMENT

Agile development is a philosophy of adaptability, collaboration, and iterative progress. Established in 2001 through the Agile Manifesto, its principles focus on delivering value quickly and effectively by emphasizing flexibility over rigid processes. Agile development methods are built to respond to change, prioritize customer needs, and encourage teamwork [1] [2].

Agile development has some basic principles, among which are the following:

DOI: 10.5121/ijsea.2024.15604

- Customer collaboration over contract negotiation. Agile practices emphasize active customer involvement throughout the project to ensure the evolving product meets their expectations. This interaction reduces the risks associated with unclear requirements [1] [3].
- Welcoming change. Agile succeeds in environments where requirements are constantly evolving. Iterative cycles, or sprints, allow teams to adapt quickly to changes without sacrificing productivity [2] [3].
- Continuous delivery. Agile teams focus on delivering small, functional product increments in short timeframes, ensuring stakeholders see value early in the development cycle [1] [3].
- Individuals and interactions over processes and tools. Agile prioritizes human collaboration and interaction over strict adherence to tools or methodologies. Techniques like pair programming embody this principle by fostering close collaboration between developers [1] [3].

In 1990, Beck proposed Extreme Programming (XP) as an approach to agile development designed to enhance software quality and responsiveness to changing requirements by promoting practices that support collaboration, frequent iterations, and adaptive planning. XP is built on values and principles that align closely with agile philosophies [3].

Continuous feedback is a cornerstone of XP, applied at multiple levels—code reviews, automated tests, and customer interactions; this feedback cycle enables teams to detect issues early and implement improvements quickly. XP encourages developers to focus on designing and implementing only what is essential to fulfill the current requirements; the goal is to avoid over-engineering and focus on delivering functional, maintainable code. XP assumes that requirements will evolve; its iterative process and strong customer collaboration enable teams to adapt quickly while staying aligned with the overarching project objectives. XP teams work in short cycles, often delivering small, functional software increments; this ensures that stakeholders see progress frequently and can provide feedback. XP fosters a culture where team members respect each other's contributions, enabling open communication and effective problem-solving [3].

Extreme Programming (XP) translates its core principles into practices that guide teams in achieving high-quality software, improved collaboration, and adaptability to change. These practices address every stage of the software development process, ensuring both technical and organizational effectiveness [3]:

- Test-Driven Development (TDD). The TDD process promotes a clear understanding of requirements, defect prevention by identifying issues early, and a complete set of tests to validate future changes. This approach ensures that the code meets the outcomes' requirements and remains maintainable over time.
- Pair programming. Two developers work together on the same task, one writing code and the other reviewing and strategizing.
- Continuous integration. Code is regularly merged into a shared repository, where automated tests are executed with each integration to prevent new changes from breaking the build, enable early bug detection, and ensure a stable and reliable codebase.
- Code refactoring. Refactoring involves restructuring existing code to improve its readability, simplicity, and performance without changing external behavior. This practice reduces technical debt, enhances code maintainability, and supports the iterative nature of XP.
- On-site customer. An on-site customer representative works closely with the development team to clarify requirements immediately, ensure the product aligns with business goals, and facilitate quick decision-making.

- Simple design. XP advocates for designing only what is necessary to meet the current requirements. The goal is to avoid over-engineering, deliver functionality faster, and keep the design adaptable for future needs.
- Small releases. Teams frequently deliver small, functional increments of the software. These releases enable rapid customer feedback, allow stakeholders to see progress, and reduce the risk of significant, unseen issues.
- Collective code ownership. The entire team shares ownership of the codebase, allowing any member to change any part of the code. This approach prevents bottlenecks caused by code "ownership" and encourages shared responsibility for code quality.
- Coding standards. XP promotes adherence to agreed-upon coding standards. These guidelines ensure consistency across the codebase, improve readability and maintainability, and make the project accessible to all team members.
- System metaphor. XP encourages using a shared metaphor to describe the system's architecture and behavior. This practice provides a common language for team members and simplifies complex technical discussions.
- Sustainable pace (Energized Work). XP prioritizes maintaining a manageable work pace to prevent burnout. Teams maintain a consistent work rhythm and prioritize regular breaks and healthy working hours.
- The planning game. The team and the customer discuss and approve objectives at the beginning of an iteration. With this information, they plan the upcoming iteration and assign tasks for each team member.

While XP offers significant benefits, it also presents challenges. For example, teams and organizations accustomed to traditional development methods may find XP's practices challenging to adopt, and practices like TDD and pair programming can require more time initially [4].

XP tends to be most effective with small to medium-sized teams. Scaling XP practices to larger organizations requires additional coordination and adaptation.

3. PAIR PROGRAMMING

Pair programming is a collaborative software development practice where two developers work together on the same codebase in real-time. This practice is designed to enhance code quality, foster knowledge sharing, and strengthen team dynamics [1] [3].

The developers assume two primary roles during a session [3] [5] [6]:

- Driver: The developer at the keyboard actively writes the code. Their focus is on implementing the immediate task at hand.
- Navigator: The partner reviews the code in real-time, thinking strategically about potential pitfalls, identifying bugs, and ensuring alignment with broader design goals.

Pair programming goes beyond writing code simultaneously; it involves an iterative process of brainstorming, reviewing, and improving code. The interaction between the Driver and Navigator fosters continuous feedback loops, which result in cleaner, more maintainable code [3] [6]. Role-switching during the session also prevents monotony and encourages both partners to engage equally.

Pair programming can offer the following benefits [3] [5] [6]:

- Improved code quality. Having two sets of eyes on the code reduces errors and increases adherence to coding standards. The Navigator often catches mistakes or suggests optimizations that the Driver might overlook.
- Knowledge sharing. Pair programming facilitates continuous learning. Junior developers gain hands-on experience under the guidance of senior developers, while seasoned programmers benefit from fresh perspectives or novel approaches introduced by their peers.
- Faster problem solving. Collaborative problem-solving accelerates debugging and decision-making. Discussing design choices in real time often leads to more innovative solutions.
- Team cohesion. The practice builds trust and strengthens team relationships. Over time, it creates a shared understanding of the codebase, reducing dependencies on specific individuals.

While the benefits are substantial, pair programming does present some challenges [6]:

- Interpersonal dynamics: Differences in work styles, communication preferences, or skill levels can create tension between partners. Clear expectations and respectful communication are critical to overcoming this.
- Increased initial time investment: Pair programming can initially slow down simple tasks, requiring both developers to agree on solutions. However, this often pays off in reduced debugging time and improved code quality later.
- Fatigue: Pair programming requires intense focus and constant interaction, which can be exhausting for developers. Regular breaks and structured sessions can help mitigate this.

For pair programming to succeed, organizations must ensure alignment at multiple levels, including shared goals, effective communication, and consistent support for collaborative practices.

3.1. Tools for Effective Pair Programming

As pair programming evolves, particularly in remote and hybrid work settings, leveraging the right tools is essential to maintaining seamless collaboration, real-time communication, and shared code editing. Some of the tools used to support pair programming are discussed below [7] [8].

1. Visual Studio Code Live Share. A popular Visual Studio Code extension that enables developers to work together in real time. It enables shared editing, debugging, and terminal access without setting up remote environments. Some key features are real-time code sharing and editing, shared debugging sessions, and live access to terminals and servers. It is ideal for remote teams; there is no need to synchronize configurations as everything runs on the host's environment.
2. JetBrains Code With Me. A collaborative tool embedded within JetBrains IDEs such as IntelliJ IDEA, PyCharm, and WebStorm. It allows team members to collaborate in real time on the same project. It provides secure project sharing, built-in video and voice calls, and remote debugging capabilities. It best suits teams already using JetBrains products, providing a seamless experience with minimal setup.
3. Tuple. A screen-sharing and remote pair programming tool designed specifically for developers. It offers low-latency performance for macOS users. It offers high-quality

screen sharing with minimal lag, remote control features for seamless collaboration, and developer-friendly shortcuts and tools. Its optimization for extended pair programming sessions is mentioned, emphasizing performance and usability.

4. Teletype for Atom is an add-on for the text editor that enables real-time collaboration by allowing developers to share their workspace. It offers real-time text editing, easy setup, and lightweight operation. It is perfect for teams that prefer Atom for its simplicity and flexibility.
5. Git and Git-based platforms. Git is a tool for version control and teamwork, with platforms such as GitHub and GitLab offering enhanced features like pull requests, issue management, and shared repositories. Their main features are version control for collaborative coding, code reviews, pull request management, and integration with CI/CD pipelines. Facilitates asynchronous collaboration and code sharing, complementing real-time pair programming tools.
6. Communication platforms. These tools are essential for maintaining seamless communication during remote pair programming sessions. They provide screen sharing, remote control options, and video and voice calling for real-time discussions. Some alternatives include Discord, ideal for informal and frequent collaboration, particularly for developers comfortable with a casual interface; Zoom, which provides robust video conferencing and screen sharing for more formal settings; and Microsoft Teams, which combines communication with task and project management tools, ideal for integrated workflows.
7. Collaborative whiteboards. While not directly for coding, these tools are valuable for brainstorming, planning, and visualizing concepts during pair programming. They also help design architecture and discuss solutions during pair programming.
8. Automating Tests. Selenium is a highly effective tool for automating tests on web applications, making it a vital component of modern software development. It supports testing across various browsers, including Chrome, Firefox, Safari, and Edge, ensuring that an application works correctly in different environments and platforms.
9. Repl.it: An online platform that enables coding and execution in multiple languages without requiring local installations, offering real-time collaboration perfect for pair programming sessions.
10. CodeSandbox: Similar to Repl.it, this tool allows you to quickly create web applications and share them with others for real-time collaboration.

3.2. Remote Pair Programming

Remote or distributed pair programming adapts the traditional practice to accommodate team members working from different locations. Tools like Visual Studio Code Live Share, Tuple, and JetBrains Code With Me enable developers to collaborate in real-time, sharing codebases, debugging sessions, and even terminal access. Effective remote pair programming requires robust communication, often supplemented by video or voice calls, to maintain the interpersonal dynamics of traditional pairing. Although this approach eliminates geographical barriers and enables diverse team structures, it also presents challenges like coordinating across time zones, maintaining stable internet connections, and bridging communication gaps caused by the lack of face-to-face interaction.

Success in distributed pair programming depends heavily on choosing the right tools, fostering strong communication practices, and building a collaborative team culture that transcends physical boundaries. This adaptation has become increasingly relevant in the shift toward hybrid and remote work environments, maintaining the collaborative benefits of pair programming while accommodating modern workplace dynamics.

Communication challenges often stem from the lack of physical presence, which can disrupt the Driver-Navigator dynamic critical to pair programming success.

3.3. Effectiveness of Pair Programming

Hannay *et al.* [6] systematically review 18 empirical studies to evaluate the effectiveness of pair programming compared to individual programming. It focuses on three key constructs: quality (number of test cases passed or number of correct solutions of programming tasks), duration (total time taken to complete the tasks that had been assessed), and effort (total effort spent by the respective groups). This study indicates that pair programming shows a small positive effect on code quality overall; quality improvements are most evident in complex tasks, where collaborative problem-solving between pairs enhances correctness and reduces defects; the quality effect varies across studies due to differences in task complexity, developer expertise, and operational definitions of "quality."

Regarding duration, pair programming results in a medium positive effect on task completion time; for simple tasks, pairs complete tasks faster than individuals, as collaboration enables efficient brainstorming and coding; however, for complex tasks, duration benefits diminish as collaboration demands increase. In effort, pair programming has a medium negative effect; the total effort increases as two developers work on the same task, effectively doubling the person-hours compared to solo programming; the additional effort is justified in cases where quality improvements or faster task completion outweigh the cost.

The analysis identifies task complexity and developer expertise as significant moderators.

The analysis highlights task complexity and developer expertise as key factors influencing the effectiveness of pair programming. For low-complexity tasks, pairs tend to complete work faster but at the expense of quality, as the collaborative benefits are less impactful. Conversely, pair programming significantly improves quality for high-complexity tasks, but this comes with increased effort and, in some instances, longer completion times. Regarding developer expertise, junior developers benefit the most from pair programming, achieving correctness levels comparable to senior solo programmers. Intermediate developers see moderate gains in both duration and quality. In contrast, senior developers often experience minimal advantages, and in some cases, their performance may decline due to the inefficiencies of over-collaboration. These findings emphasize the need to consider task and team dynamics when implementing pair programming.

Hannay *et al.* conclude that pair programming could be more uniformly effective. Its benefits depend significantly on task complexity, developer expertise, and project priorities. Organizations should adopt pair programming selectively, focusing on scenarios where its strengths—improved quality and faster completion times—justify the additional effort. Further research into moderating factors, such as team dynamics and training, is recommended to optimize its application.

Tkalich *et al.* [8] examine the implementation and effectiveness of pair programming in hybrid work environments, where teams operate across a mix of in-person and remote settings. The study collected data using a combination of surveys, interviews, and case studies involving teams in hybrid work environments. This approach focused on understanding the technical and interpersonal dynamics of pair programming across distributed and in-office settings. Data collection emphasized real-world observations of how hybrid work impacts traditional pair programming practices, including role fluidity, tool usage, and communication efficiency. By capturing qualitative and quantitative insights from participants, the study identified key challenges and adaptations required for effective collaboration in hybrid models.

Tkalich *et al.* indicate that when implemented effectively, hybrid pair programming continues to offer many of the benefits seen in traditional pair programming settings. It enhances code quality through real-time review and collaboration while facilitating knowledge transfer, making it especially valuable for onboarding remote team members. However, these advantages come with increased effort costs due to the complexity of coordinating hybrid sessions. Several challenges are inherent in hybrid pair programming. Technical barriers, such as latency, software compatibility issues, and difficulties accessing shared development environments, can hinder seamless collaboration. Cultural and interpersonal gaps also pose challenges, as building rapport and trust is more difficult in a hybrid model, potentially impacting team cohesion.

Furthermore, role ambiguity can arise; with clear communication, the Driver and Navigator roles may remain clear, diminishing the effectiveness of the practice. To address these challenges and ensure success, organizations should invest in training their teams on the technical and interpersonal aspects of hybrid pair programming. Leveraging tools supporting collaborative workflows is essential to provide an equitable experience for remote and on-site participants. Additionally, incorporating retrospectives into the workflow helps gather feedback and iteratively refine hybrid practices for continuous improvement.

Lambrechts [5] examines how pair programming performs within large-scale agile environments. Challenges such as inter-team dependencies, stakeholder coordination, and process standardization often complicate the implementation of practices like pair programming. The study identifies that efficacious alignment, where stakeholders share a common understanding, motivation, and decision-making process, is critical in overcoming these challenges.

The study highlights several factors that facilitate effective pair programming in large-scale settings. Developers' motivation is crucial; teams are more likely to adopt pair programming when they recognize its benefits, such as improved code quality and faster knowledge transfer. Additionally, strong stakeholder involvement ensures that pair programming aligns with broader project objectives and organizational goals. Effective communication channels within and across teams help quickly manage interdependencies and resolve conflicts. Clear role definitions and decision-making frameworks also reduce ambiguity in collaborative practices.

The research demonstrates that pair programming can deliver significant benefits in large-scale agile environments when aligned with organizational goals and supported by robust communication and decision-making frameworks. These benefits include enhanced knowledge sharing, improved code quality, and stronger team cohesion. However, the study also acknowledges the challenges, such as increased coordination overhead, cultural resistance, and the effort required to maintain alignment across distributed teams. To address these challenges, the study recommends integrating pair programming into broader agile processes, such as sprint planning and retrospectives, to ensure alignment with team objectives. Alignment tools, such as shared repositories, collaborative platforms, and real-time communication tools are also emphasized to streamline workflows. Regular evaluations of pair programming's impact through feedback loops and performance metrics can help organizations adapt and optimize their practices.

4. CASE STUDY: IMPLEMENTING XP AND PAIR PROGRAMMING IN A UNIVERSITY WEB SYSTEM RENEWAL PROJECT

In 2023, an institute within a Mexican university embarked on a project to modernize its web-based academic and administrative information management system. The development team opted for an Extreme Programming methodology because it was suitable for small teams and could effectively integrate short-term student collaborators. The core team consisted of three developers (two programmers and a project coordinator), with one programmer as the senior developer and the other as the on-site customer.

Each year, student collaborators joined the team for 10 months. In 2023, four students participated, increasing the team to seven members, while in 2024, eight students joined, forming a team of 11 members. This collaborative dynamic required tailored processes to ensure productivity and seamless integration of new members.

4.1. Implementation of Extreme Programming Practices

The project adopted XP practices to maximize flexibility and efficiency while maintaining high-quality code output. Below are the critical practices implemented:

- Initial Training for Students. In the first month of their participation, a 40-hour training course was conducted to onboard student collaborators. This course covered the technology stack (Laravel), Git for version control, Visual Studio Code as the primary development environment, and the fundamentals of XP.
- Pair Programming. Pair programming was central to the project. Pairs used Discord for real-time collaboration and Zoom for synchronous doubt-clearing sessions. The practice ensured continuous code review, skill-sharing between team members, and faster problem resolution.
- Weekly Meetings. The team met weekly to review progress, share updates, and address challenges. These sessions also served as a platform for retrospective feedback, aligning with XP's small releases principle.
- Version Control with Git. Git was employed for continuous integration, allowing team members to merge their branches regularly and maintain a cohesive codebase.
- Feedback and Iterative Development. The on-site customer role was pivotal in ensuring the team met client needs. Feedback was integrated into each iteration, minimizing misalignment between development goals and client requirements.
- Simple Design and Refactoring. The code was continually evaluated for simplicity and efficiency. Developers practiced refactoring to enhance readability and maintainability without altering functionality.
- Testing: Functionality tests were conducted during integrations to ensure code quality and system stability.

4.2. Evaluation of Practices

To assess the effectiveness of XP and pair programming, team members completed a Likert scale questionnaire to rate their experience, from (1) negative to (5) positive. The questions evaluated:

- General Experience: Perception of XP methodology.
- Small Releases: Usefulness of weekly feedback.
- On-Site Customer: Value of customer-like insights for task planning.

- Planning Game: Cost of accommodating client changes post-review.
- Pair Programming: Overall experience with pair programming.
- Simple Design: Satisfaction with the generated code.
- Collaboration and Respect: Degree to which ideas were shared and valued in pair programming.
- Refactoring: Frequency of code improvements during error correction or task refinement.
- Continuous Integration: Regularity of Git integrations.
- Testing: Completeness of functionality tests during integration.

Table 1 organizes the questions, indicating the practice associated with each question.

Table 1. Practices and questions

Practice	**Question**
General Experience	How do you evaluate your experience following the Extreme Programming methodology?
Small Releases	How useful were the weekly feedback sessions?
On-Site Customer	How useful were the comments from the team member acting as the customer in guiding tasks?
The Planning Game	How costly were client-requested changes after a general review?
Pair Programming	How do you evaluate your experience with the practice of pair programming?
Simple Design	How satisfied are you with the code generated?
Collaboration and Respect	When working with your pair, how much were your ideas considered?
Collaboration and Respect	When working with your pair, how useful were your partner's ideas?
Refactoring	How often did you rewrite your code to improve readability, simplicity, or efficiency without changing its behavior?
Refactoring	When fixing an error, how much did you rewrite the code to improve readability, simplicity, or efficiency?
Continuous Integration	How frequently did you integrate your branch with Git?
Testing	How complete were the functionality tests during integrations?

4.3. Analyze of experience

The general experience with the XP methodology was positive (Figure 1).

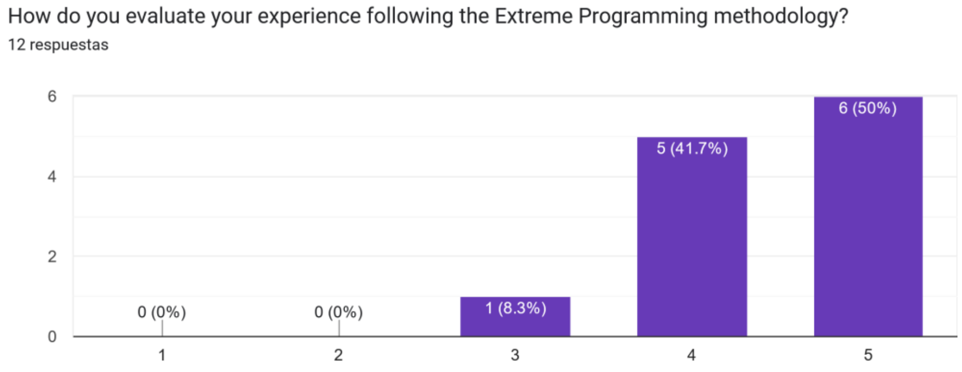

Figure 1. General Perception

Weekly feedback helped make continuous releases (Figure 2).

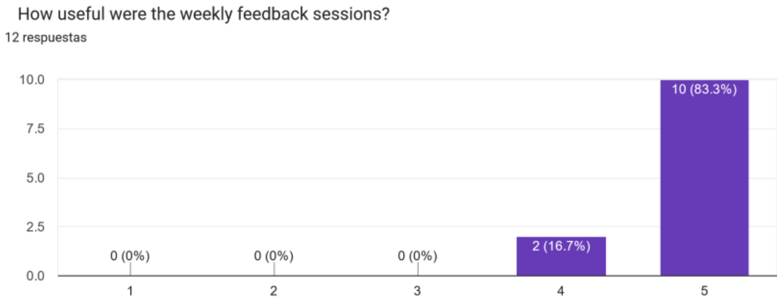

Figure 2. Small Releases

The on-site customers comments were of great value in resolving the doubts (Figure 3).

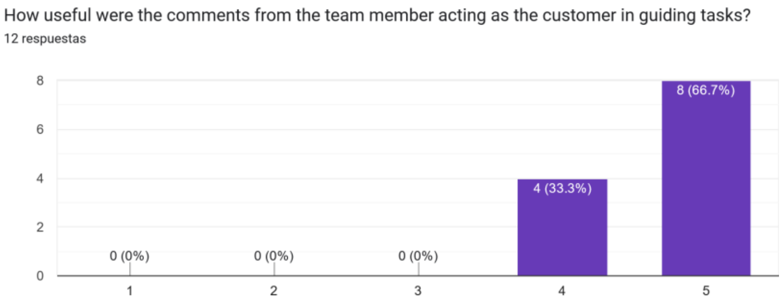

Figure 3. On-Site Customer

Although the requested changes modified the system specifications, the design was flexible and inexpensive (Figure 4).

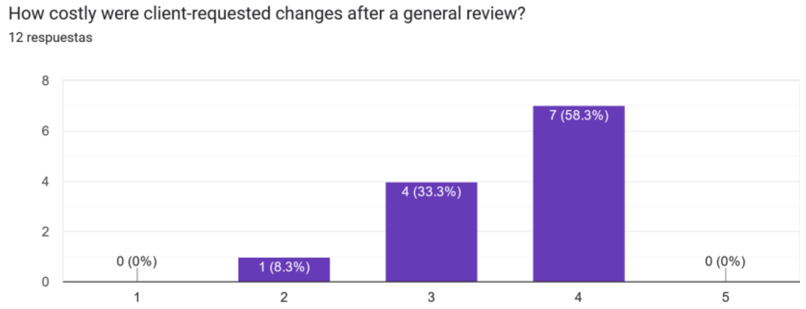

Figure 4. Planning Game

The development team perceived the pair programming experience as positive (figure 5).

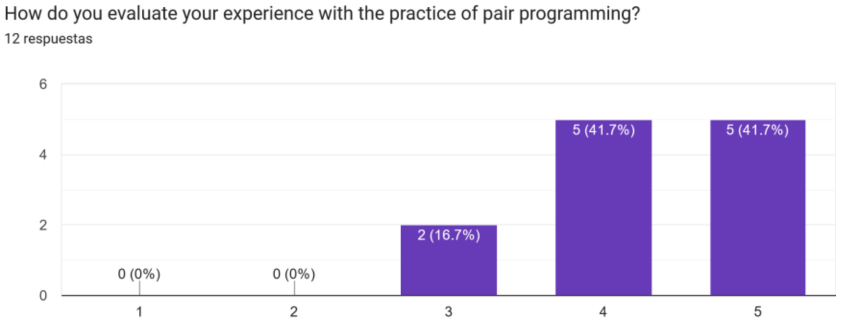

Figure 5. Pair Programming

A simple design made the team satisfied with the code produced (Figure 6).

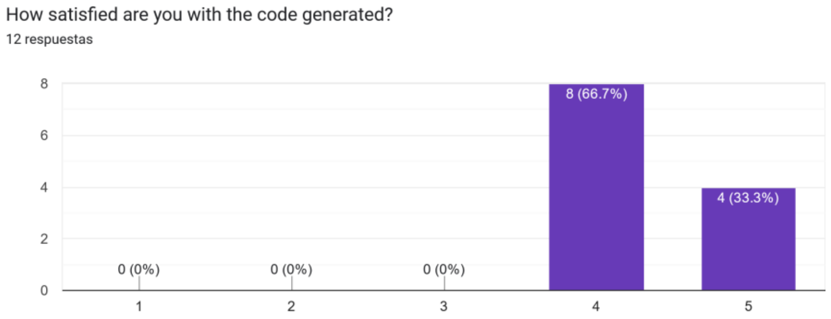

Figure 6. Simple Design

Collaboration and respect are central to pair programming; programmers perceive that their ideas are considered when solving a task (Figure 7) and respect the ideas of their peers (Figure 8).

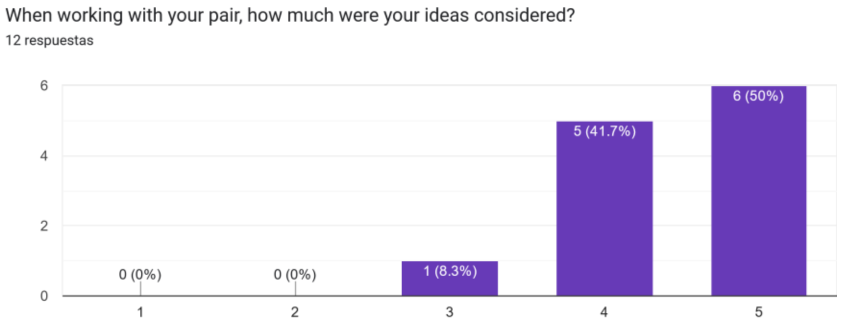

Figure 7. Collaboration and Respect

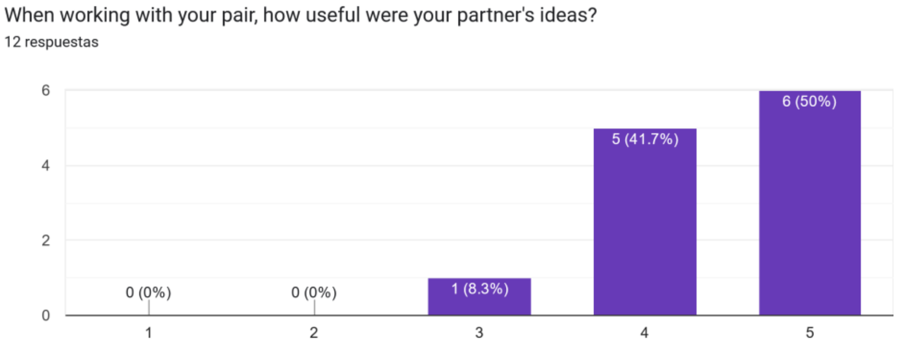

Figure 8. Collaboration and Respect

It is observed that the frequency of corrections to refactoring the code is not high (Figure 9) and that attention is concentrated on fixing errors (Figure 10).

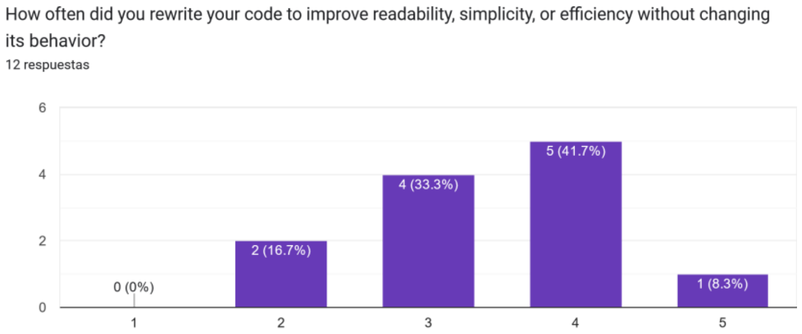

Figure 9. Collaboration and Respect

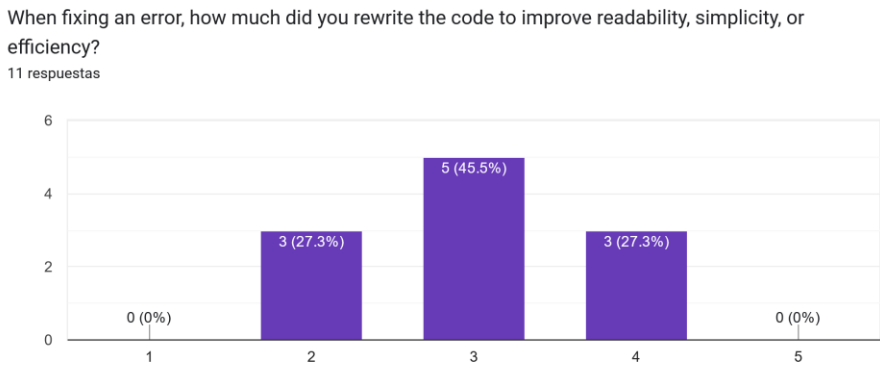

Figure 10. Collaboration and Respect

Continuous integration is central to agile development; the team needs to improve in this practice (Figure 11).

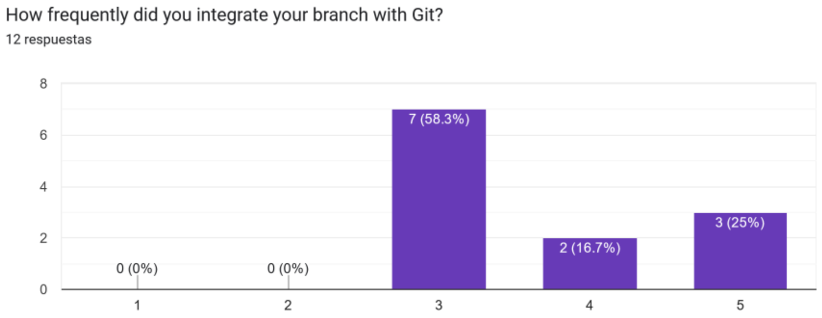

Figure 11. Continuous Integration

Testing is another practice that needs to be improved, as there is no uniform behavior on the team (Figure 12).

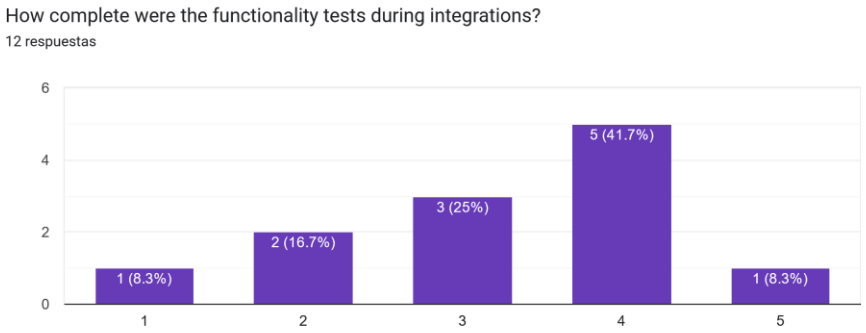

Figure 12. Testing

4.4. Challenges and insights

The project highlighted the following challenges and insights:

- Integrating new team members: The structured training program ensured students were productive quickly, but maintaining consistent skill levels required careful pairing.
- Hybrid work model: Remote collaboration necessitated reliance on tools like Discord and Zoom, which occasionally posed communication barriers but effectively supported the workflow overall.
- Flexibility: XP's iterative nature allowed the team to incorporate client feedback efficiently, but it also required a disciplined approach to planning and integration to avoid scope creep.

4.5. Outcomes

The project successfully demonstrated the feasibility of XP and pair programming in a hybrid team with temporary collaborators. Key outcomes included:

- Improved code quality: Pair programming and continuous feedback loops ensured a clean, maintainable codebase.
- Skill development: Student collaborators gained practical experience and effectively integrated their contributions into the team's workflow.
- Increased collaboration: The emphasis on teamwork and respect fostered a positive working environment.
- Adaptability: The methodology allowed the team to respond effectively to evolving requirements.

5. CONCLUSIONS

Pair programming, as a foundational practice of agile methodologies, continues to demonstrate its effectiveness in enhancing collaboration, improving code quality, and fostering knowledge transfer. This article highlights how pair programming, whether in traditional, hybrid, or large-scale agile environments, aligns with Extreme Programming principles to support iterative, adaptive, and team-oriented development processes.

The case study illustrates the effective application of XP and pair programming in a university setting, particularly within a hybrid work model. The combination of structured onboarding, collaborative tools, and iterative development practices enabled the team to overcome challenges and deliver a functional, modernized system. The experience underscores the importance of aligning tools, practices, and team dynamics to achieve project success in complex, evolving environments.

The research and case studies discussed reveal that pair programming offers substantial benefits, including increased team cohesion, faster problem-solving, and better adherence to coding standards. However, its success depends on several factors, such as task complexity, developer expertise, and the alignment of goals within the team. In hybrid work settings, where physical and virtual collaboration converge, pair programming continues to thrive with the support of modern tools and disciplined communication practices. Tools like Visual Studio Code Live Share and platforms such as Zoom or Discord bridge geographical divides, ensuring the continuity of pair programming's collaborative essence.

Nonetheless, challenges such as role ambiguity, increased effort, and technical barriers remain prevalent. Addressing these issues requires planning, investment in training, and selecting appropriate tools tailored to the team's needs. The iterative feedback loops of XP practices and retrospective evaluations can further refine pair programming's implementation and maximize its benefits.

Finally, pair programming is not a one-size-fits-all solution but a flexible practice whose value is most evident in contexts requiring high collaboration and code quality. Its scalability across diverse environments—from small university projects to large-scale agile enterprises—emphasizes its relevance in modern software development. As organizations adapt to evolving work models, the principles of pair programming, collaboration, respect, and shared responsibility remain critical to achieving development success.

REFERENCES

[1] Shrivastava, A., Jaggi, I., Katoch, N., Gupta, D., & Gupta, S. (2021, July). A systematic review on extreme programming. In Journal of Physics: Conference Series, Vol. 1969, No. 1, pp. 012046. IOP Publishing. https://doi.org/10.1088/1742-6596/1969/1/012046

[2] Bhadoriya Sanjay Singh and Parikh Saurabh (2023). Extreme Programming in Software Development. Journal of Innovative Engineering and Research (JIER) Vol. 6, Issue 2, October 2023, pp. 16-19

[3] Shore, J., & Warden, S. (2021). The art of agile development. O'Reilly Media, Inc.

[4] Arawjo, I. (2023). Programming and Culture. Ph. D thesis. Cornell University.

[5] Lambrechts, G (2022). The effect of efficacious alignment on Pair Programming in large scale agile environments. Master thesis. Open University of the Netherlands, faculty of Science.

[6] Jo E. Hannay, Tore Dybå, Erik Arisholm, Dag I.K. Sjøberg, (2009). The effectiveness of pair programming: A meta-analysis, Information and Software Technology, Volume 51, Issue 7, 2009, Pages 1110-1122, ISSN 0950-5849, https://doi.org/10.1016/j.infsof.2009.02.001.

[7] Hammer, R. (2022). An Examination of Tools and Practices for Distributed Pair Programming. KTH Royal Institute of Technology.

[8] Tkalich, A., Moe, N. B., Andersen, N. H., Stray, V., & Barbala, A. M. (2023, October). Pair programming practiced in hybrid work. In 2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (pp. 1-7). IEEE.

AUTHORS

Gustavo de la Cruz Martínez. D.Sc. in Computer Science from UNAM. Founding member of the Future Classroom project at ICAT, UNAM. Their work is reflected in the creation of various articles and participation in conferences in the fields of technology and education, as well as in the development of interactive spaces for children's interactive museums in Mexico and the construction of educational software products and applications of machine learning in education.

Selene Marisol Martínez Ramírez. Dr. in Design - Information Visualization Responsible for PAPIME PE109623, entitled "The Classroom of the Future of the CCH Sur”. She has taught undergraduate courses at UNAM, undergraduate and master's courses at UTEL, undergraduate and master's courses at UNITEC, specialty subjects at CUSA. She has also participated in 5 ICAT Classroom of the Future diplomas as a tutor. She has directed undergraduate theses in Computer Science. She has developed curricular design for