Abstract:

To maintain service continuity, software systems must be able to withstand and recover from disruptions without suffering significant performance loss[2]. This paper explores how integrating agile methodologies such as Scrum, Extreme Programming (XP), and Pair Programming with refactoring techniques like Red-Green Refactoring, Encapsulation What Varies, Composing Method, Simplifying Conditional Expressions, Moving Features Between Objects, Preparatory Refactoring, and User Interface Refactoring significantly boosts software resilience[5], [6]. Our analysis shows how these agile methods when combined with focused refactoring efforts improve the software's capacity to adjust, retain functionality, and withstand changing circumstances.

Keywords : Agile Methodologies, Refactoring Techniques, Software Resilience, Scrum, XP

Introduction

Problem Statement

Software development is a dynamic field with constantly changing requirements that make it difficult to maintain resilience and quality [2]. Although agile approaches like as Scrum and XP, enhanced by Pair Programming, provide flexibility as shown in Figure 1.1, their capacity to improve software resilience is limited by their unstructured integration with sophisticated refactoring techniques like Red-Green Refactoring, Refactoring by Abstraction, and other.

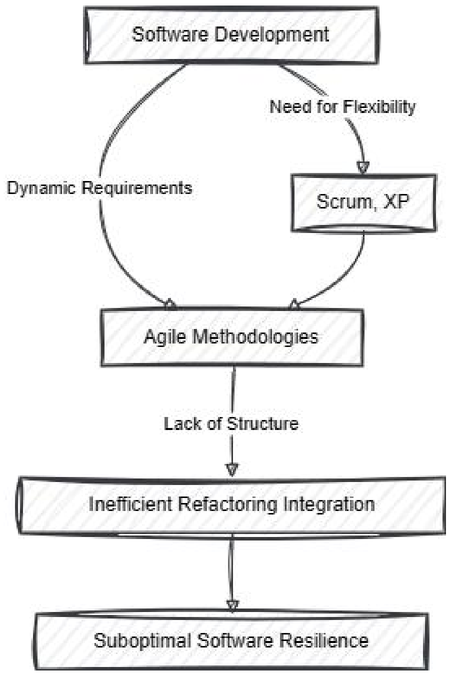


Figure 1.1

Research Questions

1. How do advanced refactoring techniques and particular agile methodologies help improve software resilience?
2. What are the gaps in Pair Programming, XP, and Scrum's integration of these refactoring techniques? [1]
3. How might the best possible software resilience be achieved by integrating these approaches with refactoring techniques?

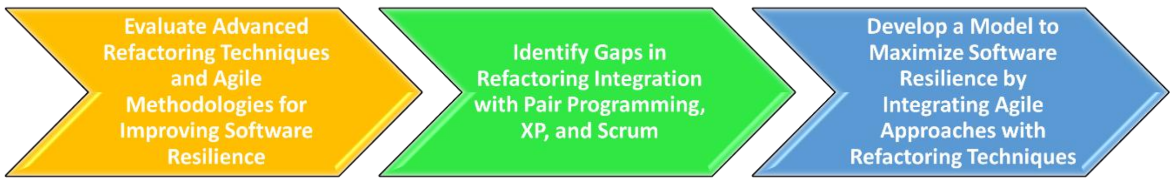

Evaluate Advanced Refactoring Techniques and Agile Methodologies for Improving Software Resilience
Identify Gaps in Refactoring Integration with Pair Programming, XP, and Scrum
Develop a Model to Maximize Software Resilience by Integrating Agile Approaches with Refactoring Techniques

Figure 2.1

Research Objectives

1. To evaluate the contribution of advanced refactoring techniques, Scrum, XP, and Pair Programming to software resilience.
2. To identify and examine these refactoring practices' integration holes inside XP and Scrum.
3. To create a model that maximizes software resilience by skillfully fusing these agile approaches with refactoring techniques.

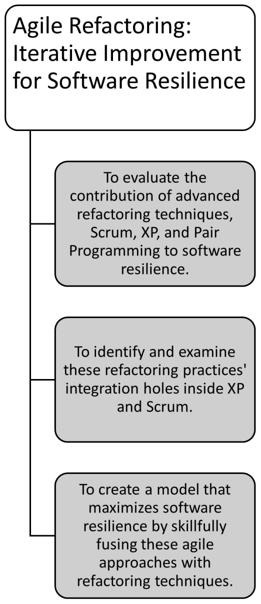

Agile Refactoring: Iterative Improvement for Software Resilience
To evaluate the contribution of advanced refactoring techniques, Scrum, XP, and Pair Programming to software resilience.
To identify and examine these refactoring practices' integration holes inside XP and Scrum.
To create a model that maximizes software resilience by skillfully fusing these agile approaches with refactoring techniques.

Figure 3.1

Research Hypothesis

H1: Scrum and XP, when combined with Pair Programming, greatly enhance software adaptation and improvement. However, they do not provide a complete method for improving code quality and maintainability by using structured refactoring approaches.
H2: By improving flexibility, maintainability, and quality, a methodological approach to incorporating sophisticated refactoring techniques into Scrum, XP, and Pair Programming greatly increases software resilience.

Literature Review:

Source	Objective	Methodology	Key Findings	Implications for Agile Refactoring	Research Gap
[1]	Explore the integration of agile practices in software engineering education.	Overview of agile methodologies, discussion on evaluative research of agile practices in academia.	Agile methodologies enhance adaptability and responsiveness, offering a mix of accepted and controversial practices. Educators must assess the value of these emerging practices.	Highlights the potential of agile methodologies, including refactoring, to improve software engineering education.	The study suggests a need for comprehensive models that integrate agile practices, including refactoring, into curricular frameworks.
[2]	Analyze the impact of	Review of various	Emphasizes the importance of	Reinforces the importance	The paper points out the lack

	software development processes on software quality.	development models and their impact on quality attributes.	software architecture and iterative models like agile for improving quality.	of refactoring in agile methodologies to enhance software quality through improved architecture.	of detailed studies on the direct impact of agile refactoring practices on specific quality attributes.
[3]	Measure and improve agile processes within small software development companies.	Case study on the implementation of agile practices and their refinement.	Agile processes enhance project adaptability and team responsiveness, leading to improved product quality and customer satisfaction.	Suggests that iterative improvement, including refactoring, is crucial for agile success in small teams.	Identifies a gap in systematic approaches to integrating and measuring the effectiveness of refactoring within agile methodologies.
[4]	Explore practitioners' perspectives on refactoring techniques for improving software quality.	Survey and interviews with software practitioners about their experiences and outcomes with refactoring.	Refactoring is deemed critical for maintaining and enhancing software quality, with varying approaches and techniques in practice.	Validates the role of refactoring in agile methodologies as essential for continuous improvement of software quality.	Highlights a research gap in comparative studies of refactoring practices within different agile frameworks and their impact on the resilience and adaptability of software systems.

Methodology:

This section discusses how the proposed research questions in contrast with the problem statement have given a model that has the combination of agile approaches with refactoring techniques to have maximum performance from every aspect.

First tackling the research questions for being able to understand the basic essence of this research, as mentioned, agile methodologies like Scrum and XP are iterative approaches that perform regular feedback and adaptation [1]. Refactoring techniques contribute by improving code quality, reducing complexity, and facilitating easier maintenance [3]. When combined, these approaches can create a robust framework for software resilience.

Secondly, the primary gap is the lack of a systematic approach to integrating refactoring techniques within agile practices [4]. Scrum often focuses on delivering features, while refactoring tends to be overlooked. Similarly, XP emphasizes coding practices but may not always include structured refactoring as part of its core methodology and the list goes on.

Thirdly, integration can be optimized by embedding refactoring tasks within Scrum sprints, using XP's Test-Driven Development (TDD) to drive continuous improvement, and promoting Pair Programming for collaborative refactoring [4]. Additionally, regular retrospectives can help identify areas for refactoring and maintain focus on code quality.

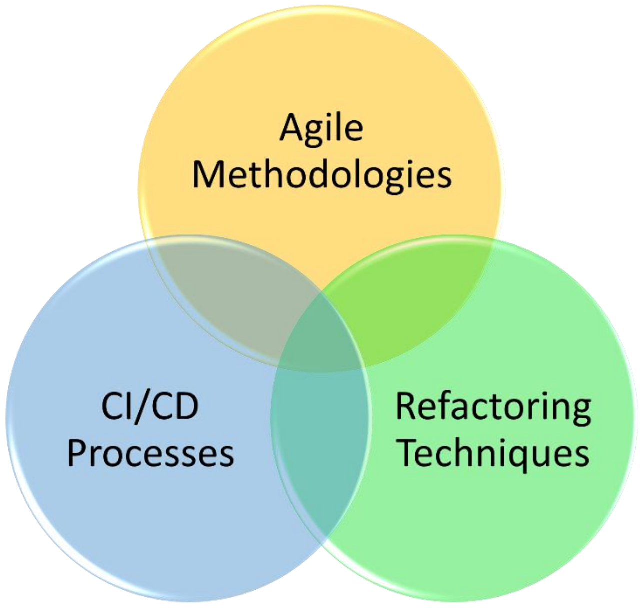


Figure 4.1

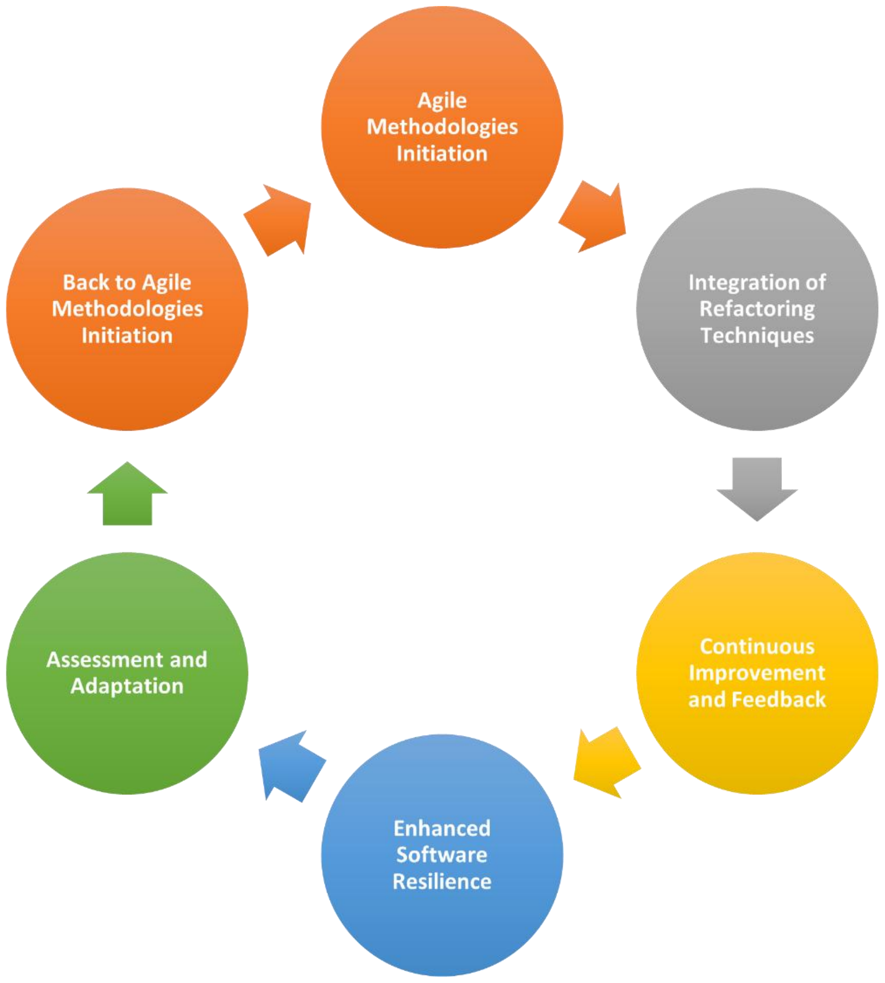
Architecture

Here is an organized approach to improving software resilience that includes Scrum, Extreme Programming (XP), and refactoring approaches at its core. As a result, a framework has been created to encourage iterative development while keeping code quality in mind.

The development process was divided into sprints, which usually lasted two weeks [1]. Each run begins with a scheduled meeting in which the improvement team identifies tasks related to refactoring. The team focuses on refactoring projects based on their impact on programming strength and quality, with Scrum serving as the foundation.

XP practices, such as Test-Driven Development (TDD) [5] and Match Writing computer programs, are integrated into the Scrum outline work. TDD is used to ensure that code quality remains consistent throughout the development process. Developers encourage a "test-first" approach, which involves writing tests before implementing functionality. This method allows for natural refactoring and fosters early issue detection. Match Writing computer applications are used to promote collaboration among designers. Working in pairs allows designers to constantly review each other's code and recommend refactoring upgrades. Additionally, this method facilitates knowledge sharing and reduces the likelihood of technical debt being established.

The structure incorporates a variety of refactoring processes, including as Red-Green Refactoring, Encapsulation What Varies, and Composing Method, [5], [6] to improve the code's structure, readability, and maintainability as it progresses through the sprints. within Pair Programming sessions, developers are encouraged to identify refactoring opportunities and implement them within the sprint.


Agile Methodologies Initiation
Integration of Refactoring Techniques
Continuous Improvement and Feedback
Enhanced Software Resilience
Assessment and Adaptation
Back to Agile Methodologies Initiation

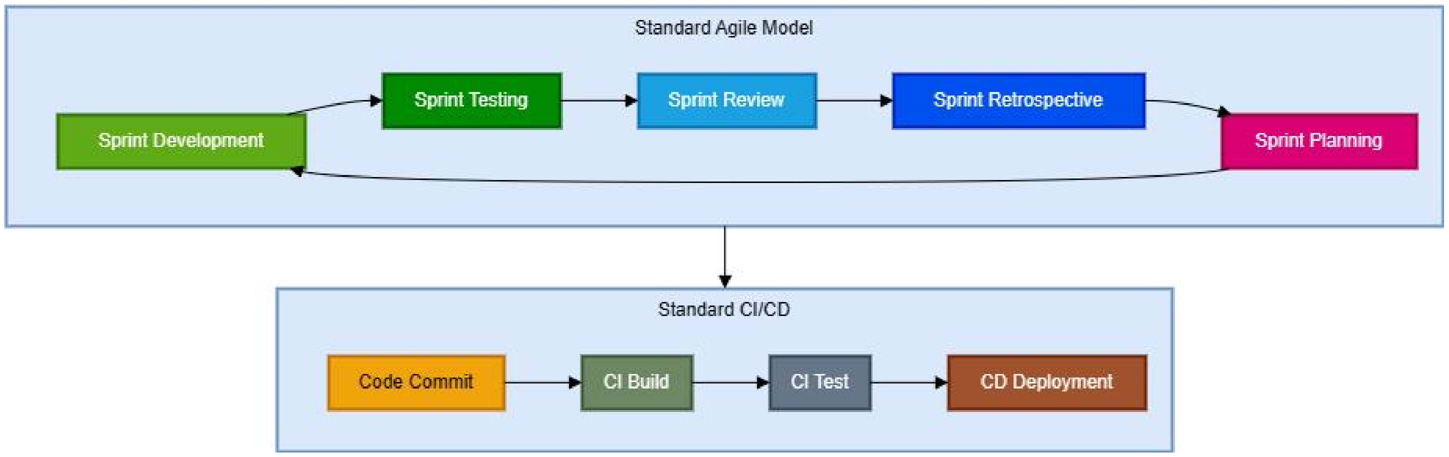
5.1

Following each sprint, the team performs a sprint review to analyze finished work and gather input from stakeholders.

This audit identifies more refactoring opportunities and ensures that code quality remains high. The run review follows the audit. During the review, the group discusses what worked well, what could be improved, and any necessary progressions for the next run. This consistent criticism circle takes into account Nimble system progress, as well as refactoring strategy coordination.

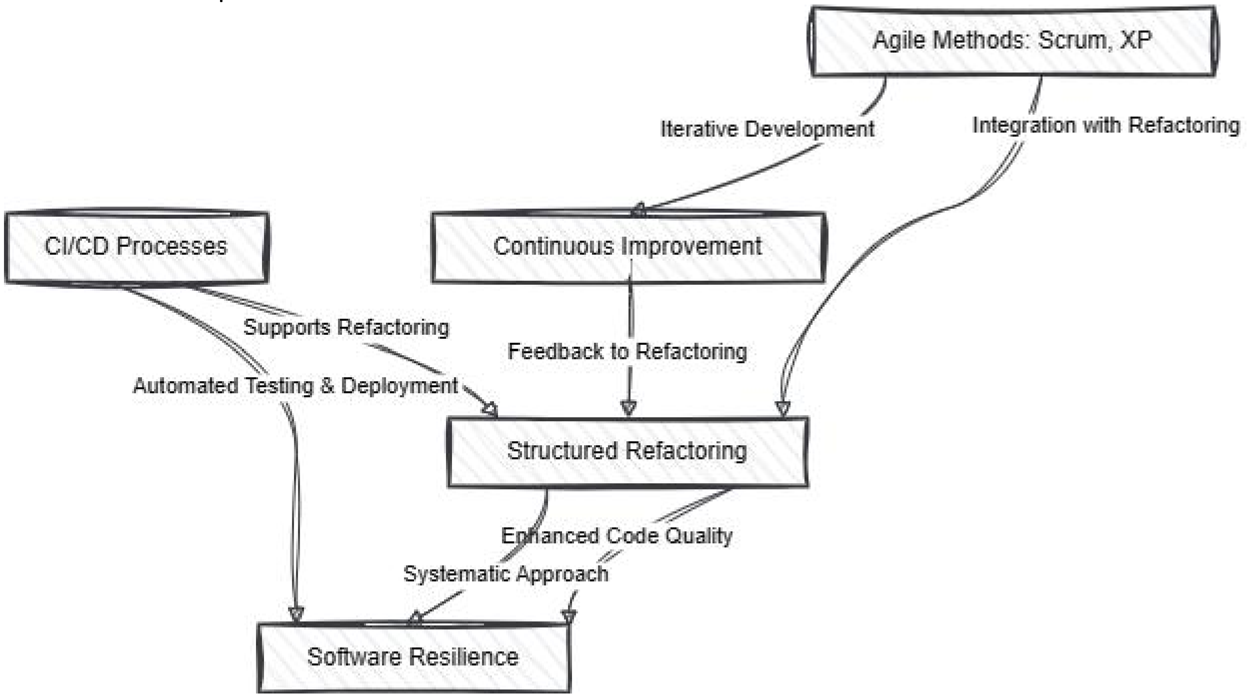
As with the Agile framework's iterative nature, we employ CI/CD procedures [4]. Continuous integration ensures that code changes are constantly integrated into a common repository and automatically tested, allowing developers to identify issues early and maintain a consistent code base. Continuous delivery automates the deployment process, allowing for frequent releases with minimal manual involvement. It improves software resilience by ensuring that changes are tested and delivered in a controlled manner, lowering the chance of disruptions [4].

Throughout the development process, data is collected to assess the effectiveness of the integrated refactoring Agile framework. This includes assessments of code quality, such as test inclusion and code complexity, as well as group execution metrics such as run performance and error rates. We examine the data to see whether refactoring techniques are included into Agile processes to improve software resilience. This assessment validates the exploration hypotheses and provides insights into how the structure might be improved for maximum programming strength.



Solution to the problem

Here's an overview of the plan:



Scrum Sprints
Development is divided into sprints, which typically last two weeks [1]. At the start of each sprint, the team conducts a planning session to identify feature development and refactoring activities. This framework ensures that refactoring is incorporated into the iteration cycle.

Test-Driven Development (XP)
Developers create tests before implementing code [1]. This method reduces problems and creates a natural opportunity for refactoring during feature development. It also encourages an emphasis on code quality.

Pair Programming (XP)
During implementation, developers collaborate in pairs to promote continuous code review. This method helps to find refactoring opportunities and maintain uniform coding standards [1].

Refactoring Techniques
Throughout the sprint, various refactoring strategies are applied to increase code maintainability and quality. These methods include the following:

- **Red-Green Refactoring**: Write a failing test (Red), then implement code to make the test pass (Green), followed by refactoring to clean up and improve the code [5].
- **Simplifying Conditional Expressions**: Improve code by decomposing parts of the condition into separate methods [6].
- **Composing Method**: Simplify complex methods by breaking them into smaller, more manageable pieces [6].

Sample code for Red-Green Refactoring

```
// Step 1: Write a failing test (Red)
@Test
public void testCalculateSum() {
    Calculator calculator = new Calculator();
    int result = calculator.calculateSum(5, 10);
    assertEquals(15, result);
}

// Step 2: Implement the code to make the test pass (Green)
public class Calculator {
    public int calculateSum(int a, int b) {
        return a + b;
    }
}

// Step 3: Refactor the code to improve quality
public class Calculator {
    public int calculateSum(int a, int b) {
        // Refactor by extracting the operation into a separate method
        return add(a, b);
    }

    private int add(int a, int b) {
        return a + b;
    }
}
```

Sample code for Simplifying Conditional Expressions [6]

```
// Before Simplifying Conditional Expressions
if (date.before(SPRING_BEGIN) || date.after(SPRING_END)) {
    charge = quantity * autumnRate + autumnServiceCharge;
}
else {
    charge = quantity * springRate;
}


// After Simplifying Conditional Expressions
if (isAutumn(date)) {
    charge = autumnCharge(quantity);
}
else {
    charge = springCharge(quantity);
}
```

Sample code for Composing Method [6]

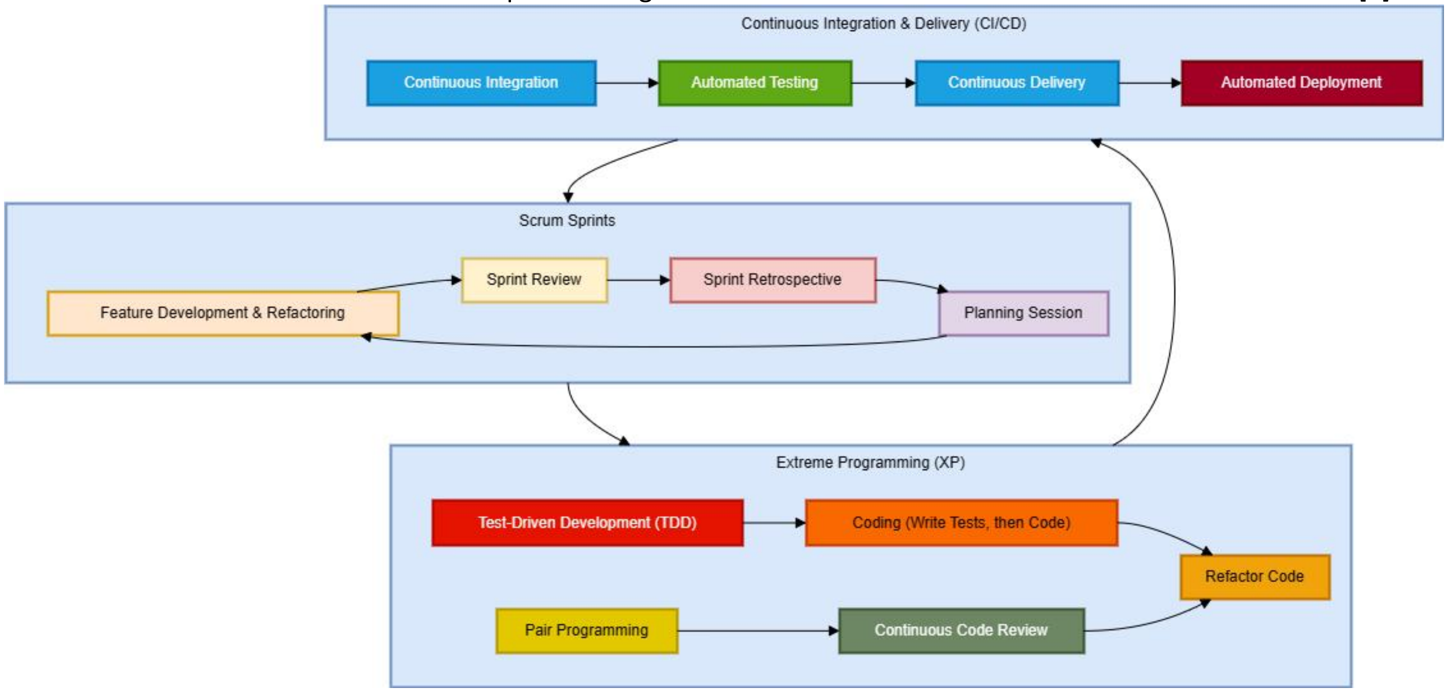
```
// Code fragment that is grouped together
public void printAll() {
    printFlag();

    // Print details.
    System.out.println("name:  " + name);
    System.out.println("amount:  " + getAmount());
}

// Extract Method (Composing Method)
// Move code to new function and call this function
public void printAll() {
    printFlag();

    printDetails(getAmount());
}

public void printDetails(double amountt){
    // Print details.
    System.out.println("name:  " + name);
    System.out.println("amount:  " + amountt);
}
```

Sprint Review and Retrospective
At the end of each sprint, the team does a sprint review to assess completed work and solicit input from stakeholders. This stage is required to keep the focus on code quality and find additional refactoring opportunities [1], [3].

Following the sprint review, a retrospective is performed to examine what went well, what may be improved, and what changes may be required for the following sprint. This constant feedback loop promotes continual improvement, which includes the use of refactoring techniques.

Continuous Integration and Continuous Delivery (CI/CD)
Applying Continuous Integration and Continuous Delivery (CI/CD) procedures guarantees that code changes are consistently integrated, tested, and deployed [4]. This method lowers the danger of technical debt and gives continuous feedback on code quality.

- **Continuous Integration**: Code updates are automatically tested and incorporated into a shared repository, allowing developers to identify bugs early on and maintain a consistent code base [3].
- **Continuous Delivery**: Deployment is automated, allowing for regular updates with little effort. This method reduces the likelihood of disruptions and guarantees that software is released in a controlled manner [3].



Hypothetical Analysis Of Case Study: Improving A Legacy Code base

Background:
A development team was provided with updating an outdated code base. The existing code base had grown over time, making it tougher to maintain and extend due to poor code quality. The software team decided to adopt a hybrid approach combining Scrum, Extreme Programming (XP) practices, and refactoring techniques to improve the code base's resilience and maintainability.

Previous Approach (Scrum):
Initially, the team utilized Scrum as their primary agile methodology. They organized development into sprints, typically lasting two weeks. During each sprint, the team would focus on implementing new features or addressing bug fixes. However, while Scrum provided a framework for iterative development, it did not explicitly address code quality concerns or technical debt reduction which raised some concerning issues with in the process.

Issues With Standardized Scrum:
1. Increasing Complexity: With each new feature or bug fix, the code base became increasingly complex and harder to maintain, leading to the accumulation of technical debt.
2. Lack of focus on code quality: The primary emphasis was on delivering new features and meeting sprint goals, often at the expense of code quality and maintainability.
3. Difficulty in refactoring: Refactoring activities were not systematically integrated into the development process, making it challenging to improve the code base's structure and quality.

Numeric Data:

- Total Sprints: 20 sprints
  - 12 sprints user stories
  - 8 sprints bug fixes
- Total Weeks: 40 weeks
- Bugs Encountered: 60 bugs
- Average Duration Per Sprint: Total Weeks/Total Sprints = 40/20 = 2 Weeks per sprint
- Average Bugs Per Sprint: Total Bugs/Total Sprints = 60/20 = 3 Bugs per sprint

Hybrid Approach (Scrum + XP + Refactoring) Applied:
To address these issues, the team adopted a hybrid approach that combined Scrum with Extreme Programming (XP) practices and refactoring techniques to improve the quality of the legacy code base.
The team continued to use Scrum as the foundation for their iterative development process. The team adopted Pair Programming, a core XP practice, during implementation. Developers worked collaboratively in pairs, continuously reviewing each other's code and identifying opportunities for refactoring. This helped them not only build new features but also improve the existing code. They regularly reviewed their progress and looked for ways to make things better. Throughout the sprint, the team applied various refactoring techniques to improve code maintainability and quality like Red-green Refactoring and Refactoring by Abstraction. The team implemented Continuous Integration, automatically testing and integrating code changes into a shared repository, enabling early detection of issues and maintaining a consistent code base. Overall, their approach helped them work more efficiently and deliver high-quality software.

Numeric Data:

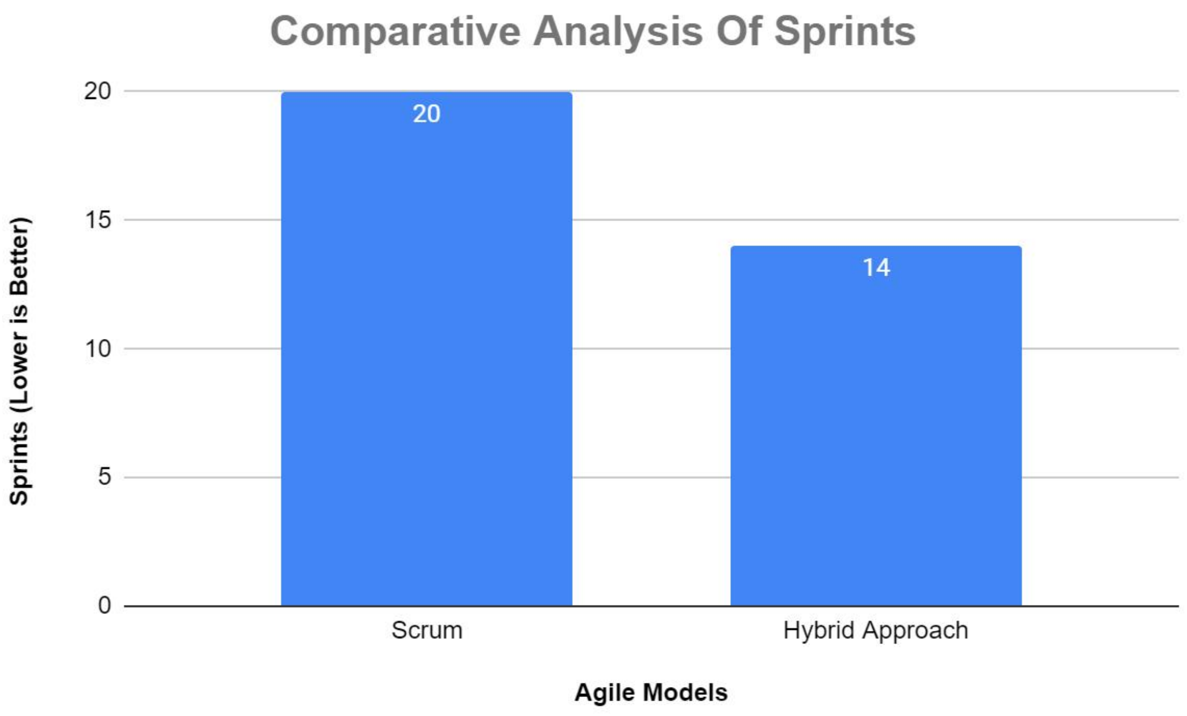
- Total Sprints: 14 sprints
  - 12 sprints user stories
  - 2 sprint bug fixes
- Total Weeks: 28 weeks
- Bugs Encountered: 16 bugs
- Average Duration Per Sprint: Total Weeks/Total Sprints = 28/14 = 2 Weeks per sprint
- Average Bugs Per Sprint: Total Bugs/Total Sprints = 16/14 = 1.14 = 1 Bug per sprint

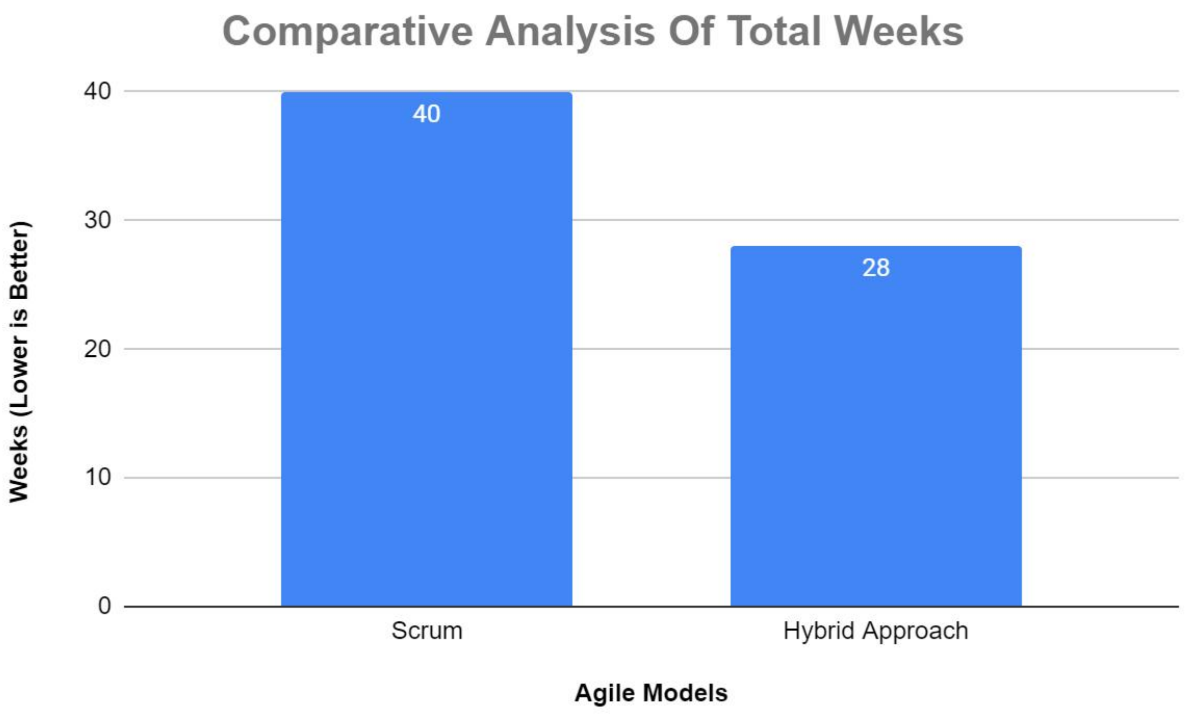
Comparative Analysis Of Both Approaches (Tabular Results):
When we compare the results of scrum and our hybrid approach on the hypothetical case study, we find that we have improved software resilience with our hybrid model which is mentioned below:

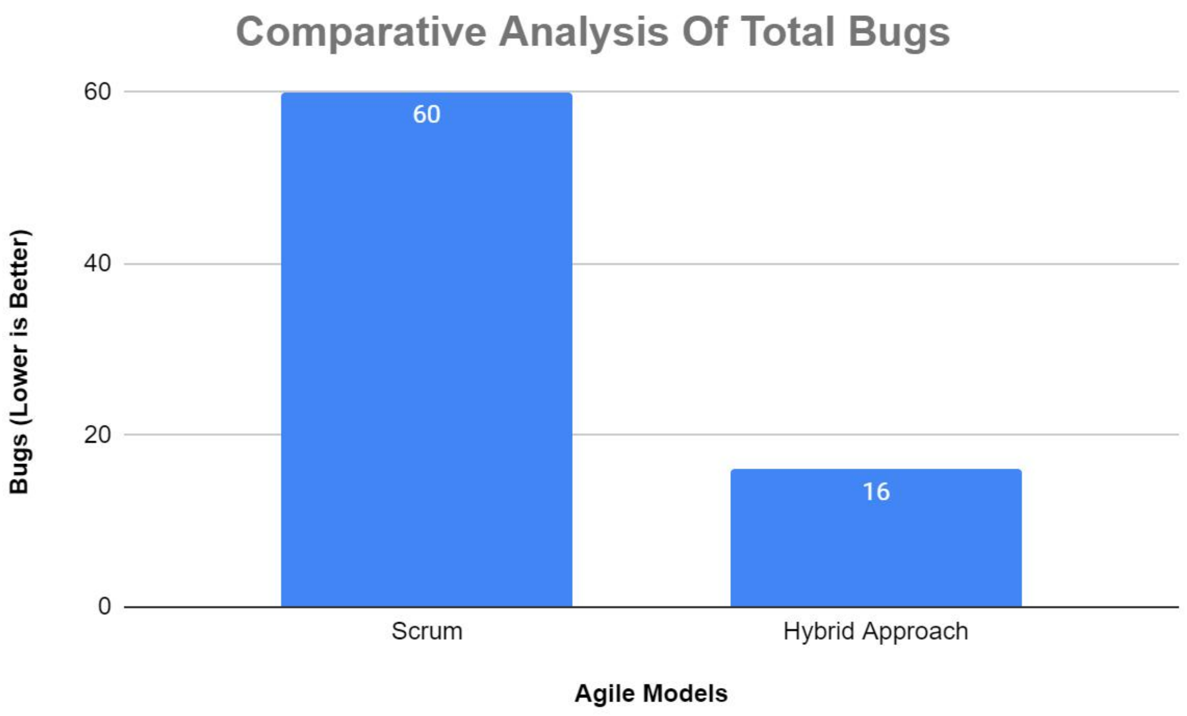
- Reduction in overall sprints
- Lesser Bugs to Fix
- Improvement in code quality and overall architecture due to Refactoring
- Increase in Team Productivity as new user stories are the main focus of every new sprint

Metric	Previous Approach (Scrum)	Hybrid Approach (Scrum + XP + Refactoring)	Difference
Total Sprints	20 sprints	14 sprints	6 sprints
Total User Story Sprints	12 sprints	12 sprints	0 sprints
Total Bug Fix Sprints	8 sprints	2 sprints	6 sprints
Average Bugs per Sprint	3 bugs	1 bug	2 bugs
Average Duration/Sprint	2 weeks	2 weeks	0 weeks

Comparative Analysis Of Both Approaches (Graphical Results):







Applications

1. **Use case 1: Enhancing a Classical Code base**
   A development team was charged with updating an outdated code base. They employed Scrum for iterative sprints and XP's Pair Programming to enable collaborative reworking. They used Refactoring by Abstraction and Composing Method to steadily improve code quality and eliminate technical debt, resulting in a more resilient code base.
2. **Use case 2: Enhancing Software Quality in a Start-up**
   A start-up team used Agile methods to improve software quality. They included refactoring tasks in their Scrum sprints and employed Test-Driven Development to ensure code quality. The team used Red-Green Refactoring and Simplifying Conditional Expressions to improve code maintainability, which led to fewer defects and more stable releases.

We write case studies that show how the framework is used in real-world situations to show how well it works [3]. These case studies demonstrate how software resilience can be improved by combining refactoring, XP, and Scrum practices. The contextual investigations incorporate insights concerning the improvement interaction, challenges confronted, and how the structure overcame those difficulties.

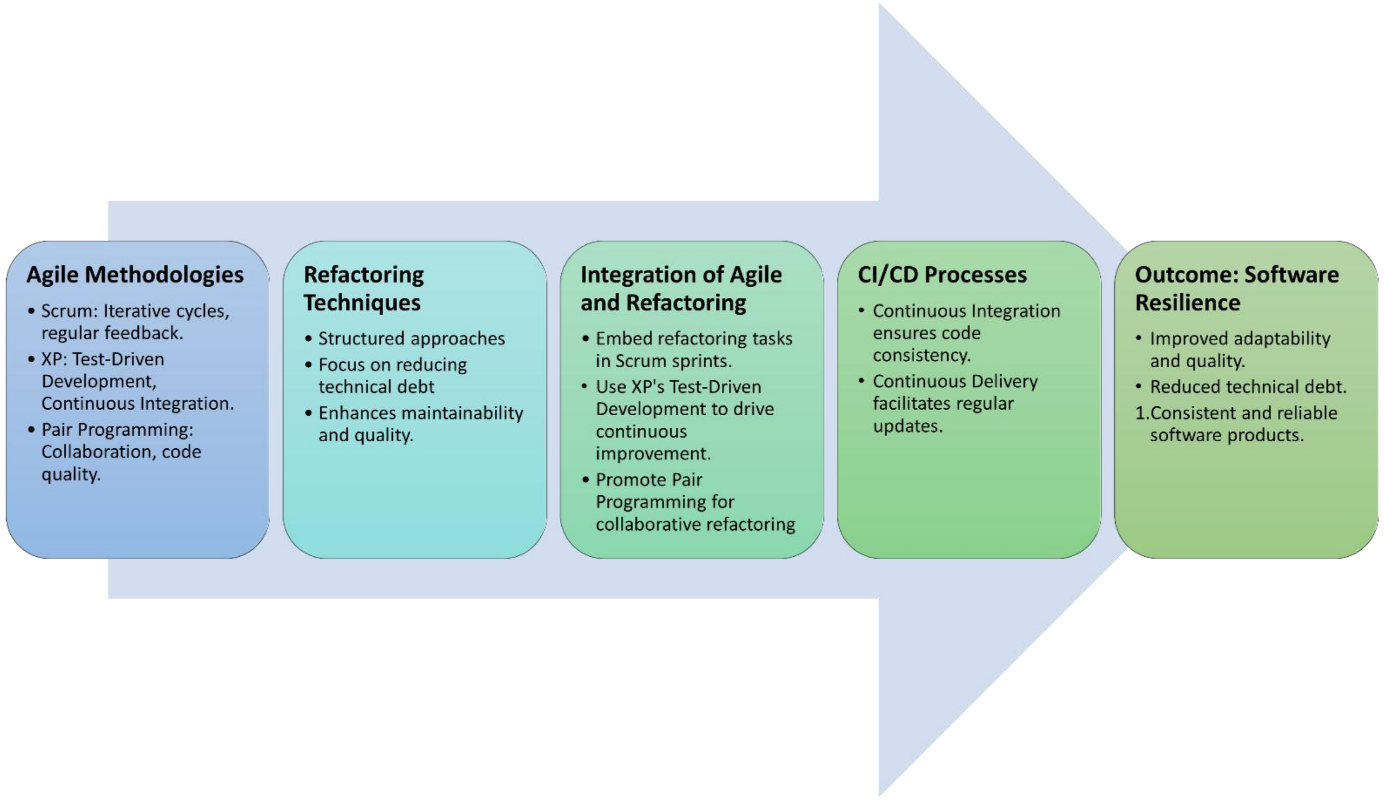
The methodology aims to establish a solid foundation for enhancing software resilience by incorporating refactoring, Scrum, and XP practices. The iterative idea of Scrum, joined with the cooperative acts of XP and the organized refactoring methods, gives a complete way to deal with further developing code quality and practicality. This strategy makes way for additional examination and testing to approve the effect of this system on programming strength.

Conclusion

This study thoroughly investigated the interaction of agile methodologies and sophisticated refactoring techniques as key mechanisms for increasing software resilience [1], [5]. We discovered that strategic integration of agile practices such as Scrum, Extreme Programming (XP), and Pair Programming with structured refactoring techniques significantly improves the adaptability and quality of software systems. Our findings highlight the need to take a systematic approach to incorporating refactoring tasks into agile techniques' iterative cycles [3]. This integration not only enables continuous code quality improvement, but it also solves the common issue of technical debt, ensuring program integrity and robustness over time. The architecture of our suggested solution, which includes continuous integration and continuous delivery (CI/CD) processes [4], has been shown to improve the consistency and reliability of software products.

Future Work

By assessing several case studies, we have exhibited real-world uses of our framework, demonstrating significant improvements in legacy systems and start-up environments through targeted usage of agile and refactoring methodologies [3]. These case studies corroborate our research findings, demonstrating that a well-executed combination of agile approaches and refactoring may result in a strong, maintainable, and high-quality software infrastructure. As a result, the potential of agile approaches boosted by methodical refactoring practices is immense and unexplored. We urge for more widespread adoption of these approaches in software development processes to encourage cultures in which continuous improvement is the norm, resilience is built-in, and software quality is never compromised. Further research in this sector should focus on refining these integration strategies and investigating their scalability across other types of software projects to improve software resilience universally.


Agile Methodologies
• Scrum: Iterative cycles, regular feedback.
• XP: Test-Driven Development, Continuous Integration.
• Pair Programming: Collaboration, code quality.
Refactoring Techniques
• Structured approaches
• Focus on reducing technical debt
• Enhances maintainability and quality.
Integration of Agile and Refactoring
• Embed refactoring tasks in Scrum sprints.
• Use XP's Test-Driven Development to drive continuous improvement.
• Promote Pair Programming for collaborative refactoring
CI/CD Processes
• Continuous Integration ensures code consistency.
• Continuous Delivery facilitates regular updates.
Outcome: Software Resilience
• Improved adaptability and quality.
• Reduced technical debt.
1.Consistent and reliable software products.

References:

[1] G. W. Hislop, M. J. Lutz, J. Fernando Naveda, W. Michael McCracken, Nancy R. Mead, and Laurie A. Williams, "Integrating Agile Practices into Software Engineering Courses," SUBMITTED DRAFT.

[2] B. Singh and S. Gautam, "The Impact of Software Development Process on Software Quality: A Review," in Proc. IEEE CICN, Dec. 2016, DOI: 10.1109/CICN.2016.137.

[3] M. Choras, T. Springer, R. Kozik, L. López, S. Martínez-Fernández, P. Ram, P. Rodríguez, and X. Franch, "Measuring and Improving Agile Processes in a Small-size Software Development Company," IEEE Access, DOI: 10.1109/ACCESS.2020.2990117.

[4] A. Almogahed and M. Omar, "Refactoring Techniques for Improving Software Quality: Practitioners' Perspectives," Journal of Information and Communication Technology, vol. 20, no. 4, pp. 511-539, Oct. 2021.

[5] "7 Code Refactoring Techniques in Software Engineering," GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/7-code-refactoring-techniques-in-software-engineering/amp/.

[6] Refactoring Guru. (n.d.). Refactoring.Guru - Design Patterns & Refactoring [Online]. Available: https://refactoring.guru/refactoring/catalog