Agile Refactoring: Iterative Improvement for Software Resilience

by Hanzala Bhutto

Submission date: 06-May-2024 02:42AM (UTC+0500)
Submission ID: 2371507946
File name: Agile_Refactoring_DDR_20k-0277.pdf (1.24M)
Word count: 3211
Character count: 19956

Agile Refactoring: Iterative Improvement for Software Resilience

Yousaf Ahmed Siddique, Hanzala, Syed Muhammad Hassan Ali.

A.

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

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

II. Introduction

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, Encapsulation What Varies, and other **[5]**.

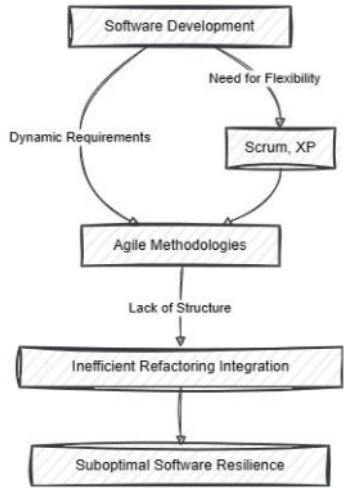

Figure 1.1

Research Questions

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

Figure 2.1

Research Objectives

- 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.

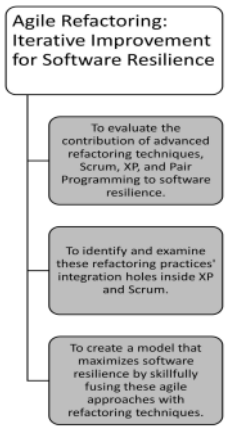

Figure 3.1

Research Hypothesis

***H1*:**
Scrum and XP, when combined with Pair Programming, greatly enhance software adaptation and improvement. However, they do not provide a complete process for improving quality of code and maintainability by using structured refactoring techniques.

***H2*:**
By improving flexibility, maintainability, and quality, a methodological approach to employing advanced refactoring techniques into Scrum, XP, and Pair Programming greatly expands software resilience.

III. 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 processes, discussion on evaluative research of agile techniques in academic settings.	Agile methodologies enhance adaptability and responsiveness, offering a mixture of accepted and arguable practices. Teachers must evaluate the value of these rising practices.[1]	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 software development processes on software standard.[2]	Review of various software models and their effect on quality elements.	Emphasizes the importance of software architecture and iterative models like agile for improving quality.	Reinforces the importance of refactoring in agile methodologies to enhance software quality through improved architecture.	The paper points out the lack of detailed studies on direct effect of agile recoding 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.

Table 1

IV. 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.

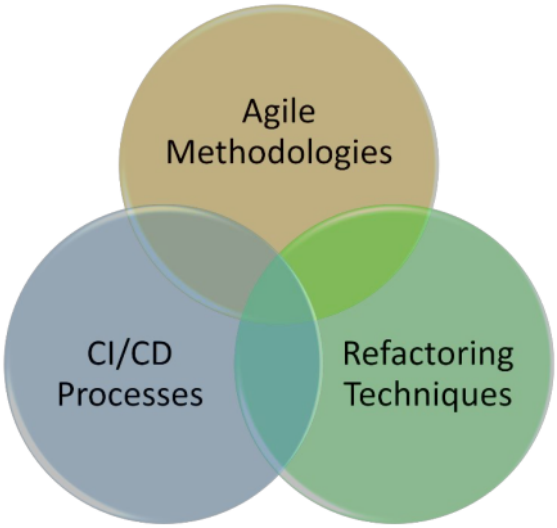

Figure 4.1

V. ARCHITECTURE

Given below we have an organized approach to improving software quality that includes agile methodologies and refactoring approaches at its core. As a result, a hybrid model has been created to encourage iterative development while keeping code quality in mind.

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

XP practices, such as Test-Driven Development (TDD) **[5]** is 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.

Figure 5.1

Following each day sprint, the software team performs a sprint review to analyze finalized things and gather feedback from stakeholders **[1]**. 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 feedback circle takes into account quick system progress, as well as refactoring strategy coordination.

As with the Agile framework's iterative nature, we employ CI/CD procedures **[4]**. Continuous integration makes sure that code changes are continuously integrated into a shared repository and automatically tested, allowing testers to identify issues early and maintain a compatible code base. Continuous delivery automates the deployment process, allowing for recurrent releases with little manual

involvement. It refines software durability by ensuring that code adjustments are tested and delivered in a managed manner, lowering the chances of interference **[4].**

Throughout the development process, data is collected to judge the potency of the integrated refactoring Agile framework. This includes assessments of code standard, such as test inclusion and code difficulty, as well as group execution metrics such as run performance and error rates. Data is examined to know whether refactoring techniques are included into Agile processes to improve software resilience. This assessment proves the hypotheses and provides insights into how the structure might be improved for utmost coding strength.

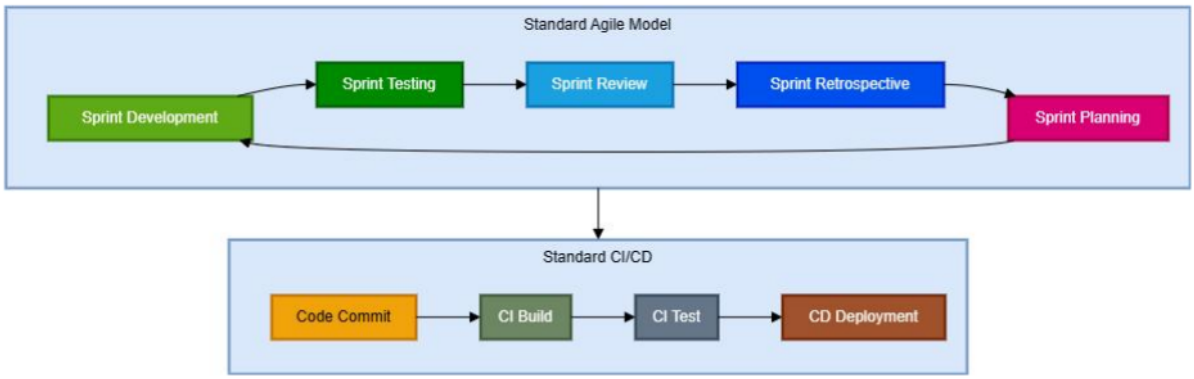

Figure 6

VI. SOLUTION TO THE PROBLEM

Here's an overview of the plan:

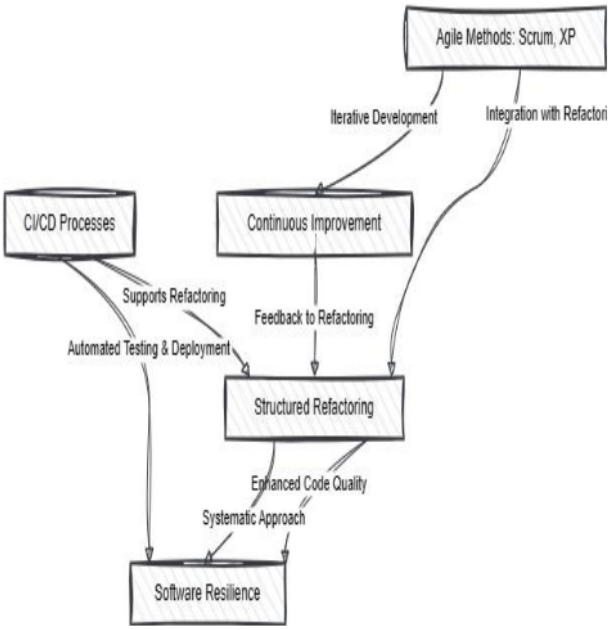

Figure 6.1

Scrum Sprints

Sprints normally last two weeks [1]. On the arrival 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)

Before implementing code, tests are generated [1]. This process reduces problems and creates a natural opportunity for refactoring during the development of a feature. It also encourages an importance on code constitution.

Pair Programming (XP)

Developers collaborate in pairs to promote continuous code review during implementation. This method helps to find refactoring opportunities and maintain uniform coding attributes [1].

Refactoring Techniques

In every part of the sprint, various recoding strategies are applied to increase code maintainability and standard. These techniques include the following:

- **Red-Green Refactoring**: Write a failing test case(Red), then implement it to make the test case pass (Green), followed by restructuring to clean up and improve the

code **[5]**.

- **Simplifying Conditional Expressions**: Enhance code by decomposing parts of the condition into separate methods **[6]**.
- **Composing Method**: Unravel complex methods by breaking them into tiny, more manageable pieces **[6]**.

Sample code for Red-Green Refactoring

```
// Step 1: Write a failing test case (Red)
public void testCalculateSum() {
   Calculator calculator1 = new Calculator();
   int result1 = calculator1.calculateSum(10, 10);
   assertEquals(20, result);
}

// Step 2: Implement the code to make the test succeed
(Green)
public class Calculator {
   public int calculateSum(int a1, int b1) {
      return a1 + b1;
   }
}

// Step 3: Rewrite the code to improve quality
public class Calculator {
   public int calculateSum(int a1, int b1) {
      // Refactor by extracting the operation into a separate
method
      return add(a1, b1);
   }

   private int add(int a1, int b1) {
      return a1 + b1;
   }
}
```

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 collected together
public void printAll() {
   printFlag();

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

// Extract Method (Composing Method) [6]
// Move code to new function and call this function
public void printAll() {
   printFlag();

   printDetails(getAmount());
}

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

Sprint Review and Retrospective

The team holds a sprint review to assess completed work and solicit input from shareholder at the end of each sprint. This part is required to keep the focus on quality of code and find further refactoring opportunities **[1], [3]**.

Additionally, a retrospective is done to inspect what things went good, what may be upgraded, and what changes may be required for the following sprint **[1]**. This constant feedback loop promotes continual improvement, which includes the use of refactoring methods.

(CI/CD) Process

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

- **Continuous Integration:** Code revisions are automatically tested and incorporated into a synced repository, allowing developers to identify errors at the start and maintain a constant code base **[4]**.
- **Continuous Delivery:** Deployment is automated, allowing for well ordered updates with small effort. This method reduces the likelihood of disruptions and guarantees that software is released in a supervised manner **[4].**

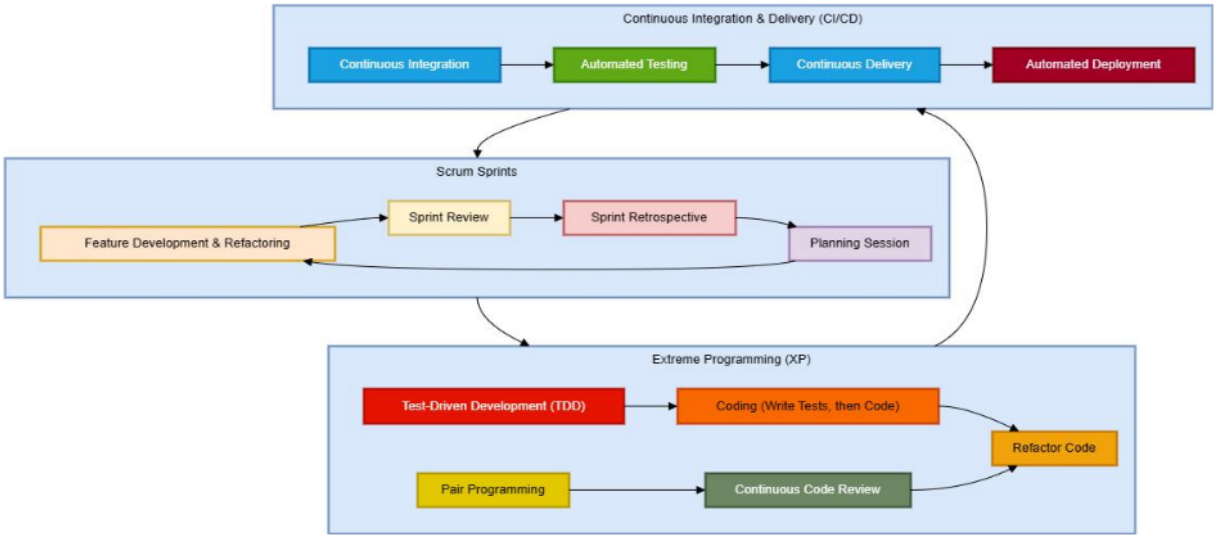

Figure 7.1

VII. 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 harder to maintain and extend due to poor coding. The software team decided to adopt a hybrid approach combining Scrum, (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.**[2]** They organized development into sprints, on average lasting two weeks. During each sprint, the team focused on developing new features or addressing bug fixes **[2]**. However, while Scrum provided a framework for iterative development, it did not explicitly address coding concerns or technical debt reduction which raised some important issues with in the process.

Issues With Standardized Scrum:

- **Increasing Complexity**: With each new feature or bug fix, the code repository became increasingly complex and tougher to maintain, leading to the collection of technical debt.
- **Lack of focus on code quality**: The primary significance was on delivering new features and meeting sprint objectives, often at the cost of code quality and maintainability.
- **Difficulty in refactoring**: Refactoring activities were not orderly 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 (XP) practices and recoding 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. Programmers worked collectively in pairs, constantly reviewing each other's code and identifying opportunities for refactoring.
This aided them not only to build new features but also upgrade the existing code. They everyday reviewed their progress and looked for ways to make things work well. Throughout the sprint, the team applied various refactoring techniques to improve code quality and maintainability like Red-green Refactoring and Refactoring by Abstraction. The team implemented Continuous Integration, automatically

testing and integrating code changes into a common repository, enabling early detection of issues and continuing a harmonious code base. Overall, their approach assist 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):

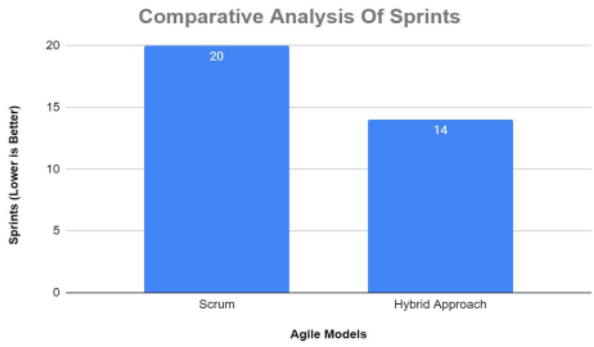

Figure 8.1

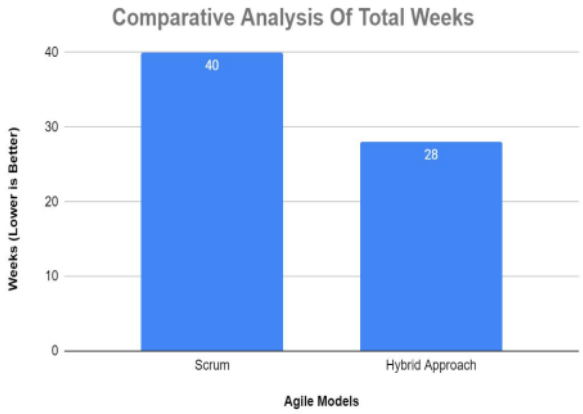

Figure 8.2

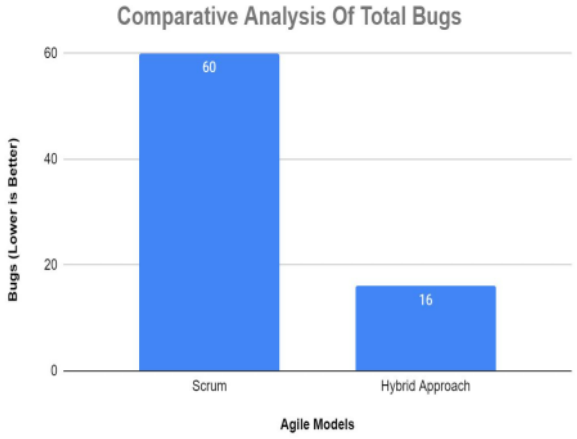

Figure 8.3

VIII. APPLICATIONS

Use case 1: Enhancing a Classical Code base
A development team was charged with changing an outdated code repository. They employed Scrum for iterative sprints and XP's to enable collaborative reworking. They used Encapsulation that varies, Composing Method to steadily improve code quality and eliminate technical debt, resulting in a more enhanced code base.**[2]**

Use case 2: Enhancing Software Quality in a Start-up
A start-up adopted Agile methods to improve software quality. They included refactoring tasks in their Scrum sprints and employed TDD 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.**[2]**

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 **[4].** 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.

IX. CONCLUSION

This study thoroughly investigated the interaction of agile methodologies and polished refactoring techniques as key mechanisms for increasing software resilience **[1], [5]**. We discovered that strategic integration of agile practices such as Scrum, (XP), and Pair Programming with structured refactoring techniques significantly improves the adaptability and quality of 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 has constant integration and continuous delivery (CI/CD) processes **[4]**, has been shown to improve the consistency and reliability of software products.

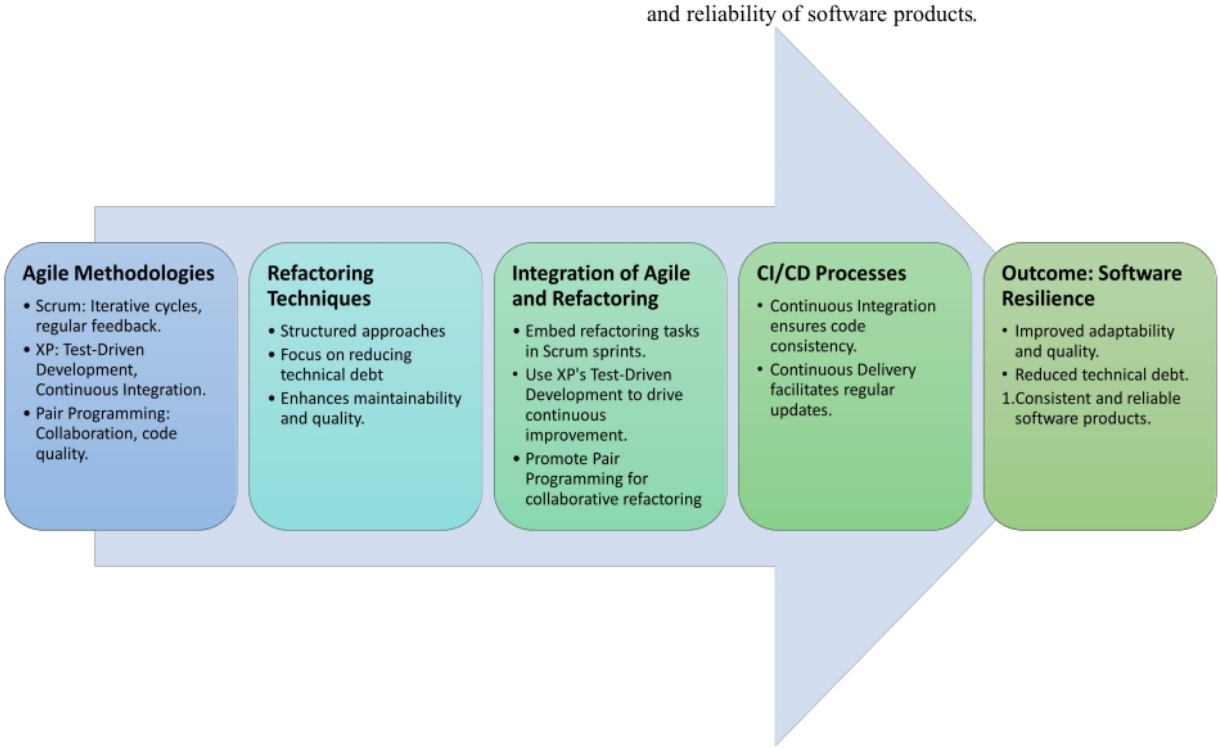

Figure 9

X. FUTURE WORK

By assessing several case studies, we have showcased real-world use cases of our hybrid technique, demonstrating notable improvements in legacy systems and start-up environments through selected usage of agile and refactoring methodologies **[3].** These case studies certify our research findings, demonstrating that a well-carried out 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 raise by methodical refactoring practices is huge and little known. 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. Additional research in this sector should focus on refining these integration strategies and investigating their scalability across other types of software projects to improve software quality universally.

XI. REFERENCES:

[1] Gregory W. Hislop, Michael J. Lutz, J. Fernando Naveda, W. Michael McCracken, Nancy R. Mead, Laurie A. Williams. "Integrating Agile Practices into Software Engineering Courses" , Computer Science Education, 2010.

[2] Brijendra Singh, Shikha Gautam. "The Impact of Software Development Process on Software Quality: A Review" , 2016 8th International Conference on Computational Intelligence and Communication Networks (CICN), 2016.

[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

[7] Bingyang Wei. "Teaching Test-Driven Development and Object-Oriented Design by Example" , 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2023.

[8] Abdullah Almogahed, Hairulnizam Mahdin, Mazni Omar, Nur Haryani Zakaria, Yeong Hyeon Gu, Mohammed A. Al-masni, Yazid Saif. "A refactoring categorization model for software quality improvement" , PLOS ONE, 2023

Agile Refactoring: Iterative Improvement for Software Resilience

ORIGINALITY REPORT

2%	2%	0%	0%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	aitpoint.com Internet Source	1%
2	archive.org Internet Source	1%
3	speakerdeck.com Internet Source	<1%
4	link.springer.com Internet Source	<1%

Exclude quotes	On	Exclude matches	< 4 words
Exclude bibliography	On		