You inherit a codebase that is poorly documented and difficult to maintain. How would you approach improving the code quality and developer experience?

Inheriting a poorly documented and difficult-to-maintain codebase can be a challenge, but with a strategic approach, you can significantly improve the code quality and developer experience.

Here's how I would tackle this situation as an Engineering Manager II:

1. Assessment and Prioritization

- **Code Review and Analysis:** Organize a thorough code review involving senior engineers to understand the codebase's overall structure, identify critical areas with poor quality, and assess the level of technical debt.
- **Documentation Audit:** Evaluate the existing documentation for completeness and accuracy. Identify areas where documentation is missing or outdated.
- **Prioritization:** Based on the code review and documentation audit, prioritize areas for improvement considering factors like criticality to functionality, maintainability impact, and potential development effort required.

2. Communication and Team Buy-in

- **Team Meeting:** Hold a team meeting to discuss the state of the codebase, the challenges it presents, and the proposed improvement plan.
- **Benefits and Ownership:** Clearly communicate the benefits of improved code quality and developer experience. Foster a sense of ownership within the team by emphasizing how better code leads to higher productivity, fewer bugs, and easier maintenance.

3. Documentation Improvement Strategy

- **Standardization:** Establish clear documentation standards for the codebase. This might include defining the level of detail required, preferred tools, and version control practices for documentation.
- **Gradual Improvement:** Encourage developers to incrementally improve documentation as they work on bug fixes or new features. This "living documentation" approach ensures documentation stays up-to-date with the evolving codebase.
- **Knowledge Sharing:** Organize knowledge-sharing sessions where team members can document specific functionalities or complex parts of the codebase.

4. Code Refactoring and Improvement

- **Technical Debt Management:** Develop a plan to address technical debt. This might involve refactoring critical pieces of code, eliminating code duplication, and improving code readability and maintainability.
- **Continuous Improvement:** Integrate code quality checks and code reviews into the development workflow. Use static code analysis tools to identify potential issues early and encourage developers to write clean and maintainable code.

5. Developer Experience Initiatives

- **Tooling and Automation:** Evaluate tools that can improve developer experience, such as code linters, formatters, and automated testing frameworks.
- **Pair Programming and Code Reviews:** Promote pair programming sessions and code reviews as opportunities for knowledge sharing, identifying improvement areas, and maintaining code quality standards.
- **Training and Upskilling:** Consider providing training opportunities for the team on best practices for documentation, clean coding, and refactoring techniques.

6. Monitoring and Iteration

- **Track Progress:** Monitor progress on documentation improvement and code quality metrics. This helps assess the effectiveness of the implemented strategies and identify areas for further improvement.
- **Iterative Approach:** Be prepared to iterate on your approach as you progress. New challenges might emerge during the process, requiring adjustments to the improvement plan.

Remember: Improving a large codebase takes time and effort.

By following these steps, establishing clear communication, and fostering team ownership, you can create a sustainable approach to improving code quality and developer experience within your team.