Car Rental System
Maintenance and Support Document

Student Name: Yanqian Chen

Student ID: 270595725

Supervisor: Mohammad Norouzifard

Yoobee College of Creative Innovation
31.01.2025

1. Strategies for Managing Software Maintenance

Importance: Software maintenance is an indispensable part of the system lifecycle. For a car rental system, continuous maintenance ensures timely bug fixes and vulnerability patching, adaptation to evolving market demands and technological landscapes, and extends the system's lifespan.

Specific Strategies:

Bug Fixing and Issue Tracking: We will establish a robust bug reporting and tracking mechanism to promptly respond to user feedback and system monitoring alerts, quickly locate and fix bugs, ensuring system stability and reliability.

Performance Optimization and Improvement: We will conduct regular performance assessments to identify system bottlenecks and implement code optimization, database tuning, and other measures to improve system responsiveness and processing capacity, optimizing user experience.

Security Updates and Vulnerability Patching: We will closely monitor security vulnerability information, promptly update system dependencies (libraries and frameworks), and patch security vulnerabilities to protect user data and system security.

Preventive Maintenance: In addition to reactive maintenance, we will also conduct code reviews, architecture assessments, and other preventive maintenance tasks to proactively identify potential issues and reduce future maintenance costs.

Benefits: An effective software maintenance strategy significantly enhances system stability, security, performance, and user satisfaction, while reducing long-term operating costs.

2. Versioning Strategies

Importance: Version control is a core practice in modern software development. For a car rental system, a clear version control strategy helps us effectively manage code changes, support parallel development, easily track historical versions, and simplify the software release and upgrade process.

Specific Strategies:

Utilize Git for Version Management: We will use Git as the primary version control tool, leveraging its branching, merge request, and other features to achieve efficient code collaboration and version iteration.

Standardized Branching Model: We will adopt a clear branching model (such as Gitflow or similar), distinguishing between the main/master branch, develop branch, feature branches, and release branches, ensuring clear code organization and standardized processes.

Detailed Commit Messages: We require developers to write clear and meaningful commit messages, describing the purpose and content of each code change, facilitating change history tracking and code reviews.

Version Tagging and Release: For each significant version release, we will apply a clear version tag, making it easy to trace and manage different versions of the code.

Benefits: A standardized version control strategy improves team collaboration efficiency, reduces code conflict risks, facilitates issue tracking and rollback, and supports rapid and reliable software releases and upgrades.

3. Strategies for Backward Compatibility

Importance: Backward compatibility is crucial for user experience, especially during system upgrades and iterations. For a car rental system, maintaining backward compatibility ensures that existing users can continue to use the system normally after upgrades, avoiding user disruption or data loss due to upgrades.

Specific Strategies:

API Compatibility: If the system provides external API interfaces, we will strive to maintain API backward compatibility. When making API changes, we will prioritize adding new APIs or extending existing ones, rather than directly modifying or deleting old APIs, to avoid impacting clients relying on older APIs.

Database Compatibility: When database schema changes are necessary, we will carefully assess the impact on existing data and adopt smooth migration strategies, such as using database migration tools, to ensure that schema upgrades do not lead to data loss or application crashes.

Configuration Compatibility: We will endeavor to maintain configuration file compatibility, avoiding the need for users to reconfigure the system after upgrades. If configuration file formats change, we will provide clear upgrade guides and migration tools.

Gradual Deprecation of Old Features: For old features that need to be deprecated, we will provide advance notice and offer alternative solutions, giving users ample time to adapt and avoiding user experience degradation due to sudden feature removal.

Benefits: A well-considered backward compatibility strategy minimizes the impact of system upgrades on existing users, maintains user trust, reduces user migration costs, and enhances user satisfaction.