UTT

UNIVERSIDAD TECNOLÓGICA DE TIJUANA

GOBIERNO DE BAJA CALIFORNIA

TEMA:

Strategy versioning.

BY:

Derian Omar Tarango Mendez

GROUP:

10 B

COURSE:

Mobile software development

PROFESSOR:

Ray Brunett Parra Galaviz

Tijuana, Baja California, 10 of january 2025

Strategy Versioning in Software Development

1. Introduction

Versioning strategies are essential in software development to manage changes, ensure compatibility, and maintain control over software updates. This document outlines different versioning strategies, their importance, and best practices for implementing them.

2. Importance of Versioning

1. **Traceability: Helps track changes and identify issues in specific versions.**
2. **Compatibility: Ensures backward and forward compatibility in software updates.**
3. **Release Management: Enables structured deployment of new features and fixes.**
4. **Collaboration: Facilitates teamwork by providing clear reference points for development and testing.**

3. Common Versioning Strategies

3.1 Semantic Versioning (SemVer)

- **Format: MAJOR.MINOR.PATCH (e.g., 1.4.2).**
- **Rules:**
  - **Increment the MAJOR version for breaking changes.**
  - **Increment the MINOR version for backward-compatible feature additions.**
  - **Increment the PATCH version for backward-compatible bug fixes.**
- **Example:**
  - **Version 2.0.0 introduces breaking API changes.**
  - **Version 2.1.0 adds a new feature without breaking existing functionality.**

3.2 Sequential Versioning

- **Format: Simple numeric sequence (e.g., 1, 2, 3).**
- **Use Case: Quick releases with minimal versioning complexity.**
- **Example: Internal tools or MVPs.**

3.3 Date-Based Versioning

- **Format: YYYY.MM.DD or YYYY.MM (e.g., 2025.01.09).**
- **Use Case: Systems with regular updates, such as cloud services or SaaS.**
- **Example: A monthly release cycle where 2025.01 is for January 2025 updates.**

3.4 Branch-Based Versioning

- **Format: References specific branches in version numbers (e.g., release-1.0, feature-x-1.2.3).**
- **Use Case: Collaborative environments using GitFlow or similar workflows.**
- **Example:**
  - **main branch holds stable releases.**
  - **feature/new-ui branch is used for developing a new interface.**

4. Selecting a Versioning Strategy

4.1 Project Complexity

- **Simple Projects: Use sequential or date-based versioning.**
- **Complex Systems: Adopt Semantic Versioning for precise control.**

4.2 Frequency of Updates

- **Frequent Updates: Date-based versioning simplifies tracking.**
- **Infrequent Updates: Semantic versioning ensures clarity in changes.**

4.3 User Impact

- **Public APIs: Semantic versioning communicates breaking changes effectively.**
- **Internal Tools: Sequential versioning is sufficient.**

4.4 Integration with Tools

- **Package Managers: Ensure compatibility with tools like npm, pip, or Maven.**
- **CI/CD Pipelines: Automate version increments based on your chosen strategy.**

5. Example: Versioning for an API

1. **Strategy: Semantic Versioning.**
2. **Initial Release: 1.0.0 with core endpoints.**
3. **Patch Update: 1.0.1 fixes a bug in the user login endpoint.**
4. **Minor Update: 1.1.0 adds a new search feature.**
5. **Major Update: 2.0.0 introduces breaking changes to authentication flows.**

6. Best Practices

1. **Document Versioning Rules: Ensure all team members understand and follow the strategy.**
2. **Use Version Control Systems: Integrate versioning into Git workflows for seamless updates.**

3. **Communicate Changes: Publish release notes for each version to inform stakeholders.**
4. **Automate Versioning: Use CI/CD tools to auto-increment versions and tag releases.**