Test Prioritization

In Software Testing

Improving Testing Efficiency Through Better Planning

Group 2

Table of Contents

Introduction

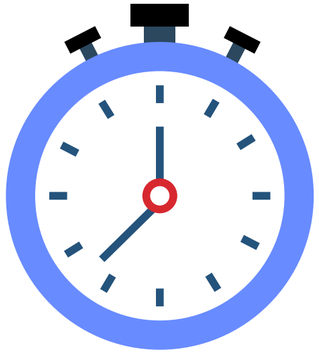
Test prioritization means selecting which test cases should be done first, especially when there isn’t enough time or resources to run all of them

- Choosing which test cases to run first
- Useful when time or resources are limited
- Focuses on important or risky parts of the software
- Helps improve testing speed and quality

Objectives of Test Prioritization

Improve Fault Detection

Find important bugs earlier



Increase Efficiency

Save time and resources

Align with Business Goals

Test what matters most to users or customers

Types of Test Prioritization

General Test Prioritization

- Test cases are prioritized once and reused for future versions.
- Focuses on overall importance of features or functions.
- Common in stable systems with fewer changes.

Example

In a banking app, tests for login, account summary, and fund transfer are always prioritized, regardless of changes in the latest release.

Types of Test Prioritization

Version-Specific Test Prioritization

- Prioritization is tailored to the changes in each new version or build.
- Emphasizes areas affected by recent updates, bug fixes, or new features.
- Useful in agile and fast-changing environments.

Example

If an e-commerce app update includes a new discount feature, tests related to discounts, checkout, and payment are prioritized in that version.

Criteria for Prioritizing Tests

Business Impact

Test cases related to features that are critical for business operations should be prioritized higher.

Example:
In a ride-sharing app, if the "Book a Ride" functionality fails, it directly affects revenue and customer trust. So, tests for ride booking are prioritized before others like profile updates or referral systems.

Failure History

Test cases that failed in previous versions or that have a history of defects are strong candidates for early execution.

Example:
In a hotel booking platform, the "date selection" module often had bugs in the past. Even if untouched in the current release, it is tested early to ensure no regression has occurred.

Criteria for Prioritizing Tests

Recent Code Changes

Tests covering newly modified code should be executed early to catch new bugs quickly.

Example:
If developers just reworked the payment gateway integration in an e-commerce app, test cases for payment processing, card validation, and checkout should be prioritized in the current test cycle.

Customer Requirements

Prioritize tests that validate high-priority user stories or contractual requirements.

Example:
In an enterprise HR software project, the client specifies that the monthly payroll generation must be error-free as a contractual deliverable. Therefore, payroll-related test cases are always run first during every test cycle.

Techniques Used in Test Prioritization

Manual Prioritization

- Done by testers or developers based on experience
- They choose test cases that seem most important for the current situation.
- Example: If login fails, the app is useless—so that test goes first.

Automated Prioritization

- Uses tools and algorithms to decide priority.
- Can be based on:
  - Code coverage (how much of the code a test covers),
  - Past defects,
  - Machine learning (in advanced systems).
- Saves time when the test suite is very large.

Risk-Based Prioritization

- Focuses on areas of the application that are high-risk or more likely to fail.
- Tests are prioritized based on how serious a failure would be.

Example Scenario

In a mobile banking app with 100 test cases but time to run only 40, it's important to prioritize key features like login, balance checks, fund transfers, and OTP security. Less critical features like theme colors, font settings, and notification sounds can be tested later. Focusing on high-risk areas first helps prevent major issues and ensures a safer, more reliable product.

Benefits of Test Prioritization

Faster Feedback

- Developers can fix the issues early.

Reduces Testing Time and Cost

- Focuses resources on what matters most.

Early Detection of Critical Bugs

- Saves from customer complaints or security issues.

Supports Quick Decision-Making

- Helps teams when there's pressure to release software quickly.

Challenges

Complexity and Scale
Since there are so many different execution paths in modern software systems (like embedded systems and AI-driven apps), thorough verification is challenging.

Evolving Requirements
Because initial validation may become out of date, frequent changes in user requirements or regulations (such as ECSS standards in space software) make V&V more difficult.

Resource Constraints
Thorough V&V is hampered by a lack of time, money, and experience, particularly for safety-critical systems like aerospace software or medical devices.

Decidability Limits
The halting problem suggests that there is a trade-off between accuracy and efficiency because no tool can fully identify every bug.

Human Error:
Code reviews and other manual V&V processes are prone to oversight, especially in large teams.

Tools

Static Analysis Tools
Code is analyzed without execution by tools like SonarQube and Coverity, which help identify possible problems (like memory leaks) early.

Model Checking
Ideal for concurrent systems, tools such as SPIN and NuSMV explore all states to verify system properties.

Dynamic Analysis
AddressSanitizer and Valgrind monitor runtime activity to spot performance snags or memory errors.

Theorem Proving
Formal proofs are provided by interactive tools such as Coq and Isabelle/HOL, which are appropriate for critical systems but necessitate expertise.

Automated Testing Frameworks
By automating the execution of test cases, JUnit and Selenium improve the effectiveness of validation.

Conclusion

Despite inherent difficulties like complexity, changing requirements, and resource limitations, software verification and validation, or V&V, is an essential procedure to guarantee software safety and dependability. Efficiency and accuracy have been greatly increased by developments in tools like model checkers (like SPIN), static analyzers (like SonarQube), and AI-driven platforms (like DeepCode). However, drawbacks like some bugs' indecision and the requirement for human knowledge still exist, calling for a well-rounded strategy. Practical assurance is made possible by combining best practices, such as early V&V, precise specifications, and varied testing, with prioritization tools like TestRail and Jira. The key to future advancement is combining automation and human supervision to successfully handle new problems.

Group Members

21CSE0146	Hasmoon
21CSE0171	Ulindu
21CSE0179	Isuru
21CSE0186	Dilshan

Thank You

for your attention