CSCI3100: Software Engineering

Test Plan Template

April 16, 2025

Table of contents

0.1 Document Revision History

Version	Revised By	Revision Date	Comments
1.0	CSCI3100 instructors	17 Apr 2025	Initial draft

1 Test Plan for Pokeman Game

This is a sample test plan for a game titled “Pokeman Gold”. The most important sections include:

1. Scope and Objectives — Outlining what will be tested and what is excluded from testing.
2. Test Cases and Scenarios — Expanding on the “Scope and Objectives” by providing a detailed breakdown of the scope, along with testing procedures and pass/fail criteria. Similar test cases and scenarios can be merged together.
3. Team Roles and Responsibilities — Defining who is responsible for each task.
4. Testing Approach & Timeline and Schedule — Detailing how and when the tests will be carried out.

This long document is just for demonstration and completeness. Your actual test plan does not need to be overly lengthy. Please do not spend much time, and simply provide very brief and clear text in the important sections.

1.1 1. Scope and Objectives

This section outlines what will and will not be tested, helping to avoid scope creep and maintain focus throughout the testing process. It also establishes clear boundaries for the testing efforts.

1.1.1 Scope:

The test plan focuses on ensuring the functionality, performance, and user experience of the Pokeman game. The following areas will be tested:

- Core gameplay mechanics (e.g., catching Pokeman, battling, levelling up, evolving).
- User interface (UI) and user experience (UX).

- Multiplayer functionality (if applicable).
- Game progression and storyline.
- Compatibility across supported platforms (e.g., PC, console, mobile).
- Save/load functionality.
- In-game economy (e.g., currency, items, shops).
- Audio and visual effects.
- Localization and language support.

1.1.2 Out of Scope:

- Testing of third-party integrations not directly related to gameplay (e.g., external payment systems).
- Hardware-specific performance testing beyond the officially supported devices.
- Modding or hacking scenarios.
- Some hard-to-test aspects of the software (please see Risk assessment and mitigation).

1.1.3 Objectives:

- Verify that the game meets functional and non-functional requirements, and provides a bug-free experience.
- Ensure the game is engaging, intuitive, and free of critical design flaws.
- Confirm that the game performs well under various conditions (e.g., high player load, low-end devices).
- Validate that the game is ready for release and meets quality standards.

1.2 2. Test Cases and Scenarios

This section provides a concise summary of the types of test cases and scenarios to be executed, their purpose, and their alignment with project goals. This ensures that stakeholders, including non-technical team members, have a clear understanding of what is being tested without needing to delve into the code repository. Links to the specific test cases in the code repository can also be provided.

Outline the situations and conditions that need to be verified.

1.2.1 Example Test Cases for Functional Requirements:

1. Catching Pokeman:
    - Steps: Enter a wild Pokeman encounter, throw a Poke Ball, and attempt to catch the Pokeman.

- Expected Result: Pokeman is caught successfully or escapes based on game logic.
- Pass/Fail Criteria: Pokeman is added to the player's collection if caught.
- Test cases:
  - Test Case ID: TC001
    - Repository Link: GitHub - Test Code for TC001
  - Test Case ID: TC002
    - Repository Link: GitHub - Test Code for TC002

2. Battle Mechanics:
   - Steps: Initiate a battle with another trainer or wild Pokeman, select moves, and complete the battle.
   - Expected Result: Damage calculations, status effects, and win/loss conditions work as intended.
   - Pass/Fail Criteria: Battle outcomes align with game rules.
3. Evolution:
   - Steps: Level up a Pokeman to its evolution threshold or use an evolution item.
   - Expected Result: Pokeman evolves into the correct form.
   - Pass/Fail Criteria: Evolution animation plays, and the Pokeman's stats and appearance update correctly.
4. Multiplayer Testing:
   - Steps: Connect to another player for a trade or battle.
   - Expected Result: Connection is stable, and actions (e.g., trading Pokeman) are completed successfully.
   - Pass/Fail Criteria: No disconnections or data loss during multiplayer interactions.
5. Save/Load Functionality:
   - Steps: Save the game, close the application, and reload the save file.
   - Expected Result: Game state is restored accurately.
   - Pass/Fail Criteria: No data corruption or loss.
6. Localization:
   - Steps: Switch the game language and navigate through menus and dialogues.
   - Expected Result: Text is displayed correctly in the selected language.
   - Pass/Fail Criteria: No missing or incorrect translations.

1.2.2 Example Test Cases for Non-Functional Requirements

Non-functional requirements focus on the quality attributes of the Pokeman game, such as performance, scalability, usability, reliability, and security.

1.2.2.1 1. Performance Testing

1. Game Load Time

- Objective: Verify that the game loads within the acceptable time limit.
- Steps:

    1. Launch the game on a supported platform.
    2. Measure the time taken to load the main menu from the splash screen.

- Expected Result: The game loads within 5 seconds on high-end devices and 10 seconds on low-end devices.
- Pass/Fail Criteria: Pass if the load time is within the acceptable range; fail otherwise.

2. Frame Rate Stability

- Objective: Ensure the game maintains a stable frame rate during gameplay.
- Steps:

    1. Play the game in different scenarios (e.g., battles, open-world exploration, multiplayer).
    2. Use a frame rate monitoring tool to measure FPS (frames per second).

- Expected Result: The game maintains a minimum of 60 FPS on high-end devices and 30 FPS on low-end devices.
- Pass/Fail Criteria: Pass if FPS remains stable without significant drops; fail otherwise.

3. Stress Testing

- Objective: Test the game's performance under heavy load.
- Steps:

    1. Simulate a scenario with many NPCs, animations, and effects (e.g., a crowded city or a large battle).
    2. Monitor CPU, GPU, and memory usage.

- Expected Result: The game remains responsive, and resource usage stays within acceptable limits.
- Pass/Fail Criteria: Pass if the game does not crash or lag excessively; fail otherwise.

1.2.2.2 2. Usability Testing

1. Navigation Intuitiveness

- Objective: Verify that the game's menus and controls are intuitive for new players.
- Steps:

   1. Ask a new player to navigate through the main menu, inventory, and settings without prior instructions.
   2. Observe their interactions and note any confusion or errors.

- Expected Result: The player can navigate the menus and controls without difficulty.
- Pass/Fail Criteria: Pass if the player completes tasks without significant confusion; fail otherwise.

2. Accessibility Features

- Objective: Ensure the game supports accessibility options for players with disabilities.
- Steps:

   1. Enable accessibility features (e.g., colourblind mode, subtitles, adjustable font sizes).
   2. Play the game with these features enabled.

- Expected Result: Accessibility features function correctly and improve the experience for players with specific needs.
- Pass/Fail Criteria: Pass if all accessibility features work as intended; fail otherwise.

1.2.2.3 3. Reliability Testing

1. Save/Load Consistency

- Objective: Verify that the game saves and loads data reliably.
- Steps:

   1. Save the game at various points (e.g., during battles, in the overworld, in menus).
   2. Exit the game and reload the save file.

- Expected Result: The game state is restored accurately without data corruption.
- Pass/Fail Criteria: Pass if the save/load functionality works without issues; fail otherwise.

2. Crash Recovery

- Objective: Test the game's ability to recover from unexpected crashes.
- Steps:

   1. Simulate a crash (e.g., force close the application or disconnect power).
   2. Relaunch the game and check the last saved state.

- Expected Result: The game resumes from the last saved state without data loss.
- Pass/Fail Criteria: Pass if the game recovers successfully; fail otherwise.

1.2.2.4 4. Security Testing

1. Data Integrity

- Objective: Ensure that player data (e.g., save files, account information) is not corrupted or altered maliciously.
- Steps:

    1. Attempt to modify save files or account data externally.
    2. Relaunch the game and check for any unauthorized changes.

- Expected Result: The game detects and prevents tampering with player data.
- Pass/Fail Criteria: Pass if data integrity is maintained; fail otherwise.

2. Multiplayer Security

- Objective: Verify that multiplayer interactions are secure and free from exploits.
- Steps:

    1. Simulate a multiplayer session (e.g., trading Pokeman, battling).
    2. Attempt to exploit the system (e.g., duplicate items, manipulate trade outcomes).

- Expected Result: The game prevents exploits and ensures fair multiplayer interactions.
- Pass/Fail Criteria: Pass if no exploits are successful; fail otherwise.

1.2.2.5 5. Compatibility Testing

1. Cross-Platform Compatibility

- Objective: Ensure the game functions correctly on all supported platforms (e.g., PC, console, mobile).
- Steps:

    1. Play the game on each supported platform.
    2. Test core features (e.g., battles, catching Pokeman, saving/loading).

- Expected Result: The game performs consistently across all platforms.
- Pass/Fail Criteria: Pass if there are no platform-specific issues; fail otherwise.

2. Device Compatibility

- Objective: Verify the game's performance on a range of devices (e.g., high-end, mid-range, low-end).
- Steps:

    1. Install and play the game on devices with varying specifications.

- Expected Result: The game runs smoothly and without significant issues on all tested devices.
- Pass/Fail Criteria: Pass if the game performs well on all supported devices; fail otherwise.

1.3 3. Resource Allocation

This section outlines the personnel, tools, environments, and other resources required to execute the test plan effectively. Proper resource allocation ensures that the testing process is efficient, well-organised, and capable of meeting the project's objectives within the given timeline. In larger organisations, there is typically a specialised team responsible for testing.

1.3.1 1. Team Roles and Responsibilities

1.3.1.1 Key Team Members:

1. Test Lead/Manager:
   - Oversees the entire testing process.
   - Develops the test strategy and ensures alignment with project goals.
   - Assigns tasks to team members and monitors progress.
   - Communicates testing progress, risks, and results to stakeholders.
2. Test Engineers/Quality Assurance (QA) Testers:
   - Write, execute, and maintain test cases.
   - Perform functional, non-functional, and exploratory testing.
   - Log and track defects in the bug tracking system.
   - Collaborate with developers to verify bug fixes.
3. Automation Test Engineers:
   - Develop and maintain automated test scripts for regression, performance, and smoke testing.
   - Ensure automation tools are configured and functioning correctly.
   - Identify areas where automation can improve efficiency.
4. Developers (for Unit and Integration Testing):
   - Perform unit testing on individual components.
   - Collaborate with testers during integration testing to resolve issues.
   - Provide technical support for debugging and fixing defects.
5. UI/UX Designers:
   - Validate the user interface and user experience.
   - Ensure that the game meets accessibility and usability standards.
6. Localization Specialists:
   - Verify translations and ensure the game supports multiple languages.
   - Test region-specific features and content.

7. Development and operations (DevOps)/IT Support:
    - Set up and maintain the testing environments (e.g., servers, databases, tools).
    - Ensure continuous integration/continuous deployment (CI/CD) pipelines are functioning.
8. Product Owner/Business Analyst:
    - Provide clarification on requirements and acceptance criteria.
    - Validate that the game meets business objectives during User Acceptance Testing (UAT).

1.3.1.2 Example Resource Allocation Table:

Role	Name	Responsibilities	Time Commitment
Test Lead	TK Lam	Oversee testing, manage team, report progress.	Full-time
QA Tester	Kei	Execute test cases, log bugs, perform regression.	Part-time
Automation Engineer	Lam TK	Develop and maintain automated test scripts.	As needed
Developer	TKL	Perform unit testing, assist with integration issues.	As needed
UI/UX Designer	LTK	Validate UI/UX and accessibility.	Part-time
Localization Specialist	K	Test translations and region-specific content.	Part-time
DevOps Engineer	L	Maintain test environments and CI/CD pipelines.	As needed
Product Owner	T	Validate requirements and acceptance criteria.	As needed

1.3.2 2. Tools and Software

1.3.2.1 Testing Tools:

1. Test Management Tools:
    - In many development workflows, test cases for unit testing and integration testing are often stored directly in the code repository. These tests are typically written and maintained by developers and are executed automatically as part of the CI/CD pipeline, such as `Github Actions`. Unlike unit and integration tests, regression

test cases are often more comprehensive and may reside in standalone repositories. These repositories are managed separately because:

- They may include large datasets or complex test scenarios.
- They are often executed less frequently (e.g., nightly builds or before major releases).
- They may require specialized tools or environments.

While CI/CD tools handle the automation of test execution, specialized test management tools such as `Testiny` provide a more comprehensive solution for managing the entire testing lifecycle. These tools help with:

- Organizing test cases.
- Tracking test execution.
- Reporting test results.
- Managing test environments.

2. Bug Tracking Tools:
   - Tools like `Jira`, `Bugzilla`, `Github`, and `Gitlab` to log, track, and manage defects.
3. Automation Tools/Human:
   - Tools like Selenium for automating test cases for web applications with graphical user interfaces.
   - In game development, the UI tends to undergo frequent changes throughout the process, and manual testing is essential since user experience (UX) is crucial for creating a product designed to be enjoyable.
4. Performance Testing Tools:
   - Tools like `cProfile` or Unity Profiler to test the game's performance under load.
5. Compatibility Testing Tools:
   - Tools like `BrowserStack` to test the game on multiple devices and platforms.
6. Version Control Tools:
   - Tools like `Git` to manage test scripts and documentation.
7. Game-Specific Debugging Tools:
   - Tools like `Unity Profiler` or `Unreal Engine Debugger` to identify performance bottlenecks and bugs in the game engine.

1.3.3 3. Testing Environments

1.3.3.1 Environment Types:

1. Development Environment:

- Used by developers for unit testing and initial debugging.
- Includes incomplete or partially implemented features.

2. Testing/Staging Environment:
   - A replica of the production environment used for system, integration, and regression testing.
   - Includes all features and configurations required for testing.
3. Production Environment (for UAT):
   - Used for final validation during User Acceptance Testing.
   - Mimics the live environment to ensure the game is ready for release.

1.3.3.2 Environment Setup:

- Hardware Requirements:
  - Devices for compatibility testing (e.g., PC, consoles, mobile devices).
  - High-performance machines for performance and stress testing.
- Software Requirements:
  - Game builds, test tools, and debugging utilities.
  - Access to databases, APIs, and other backend systems.

1.3.3.3 Example Environment Allocation Table:

Environment	Purpose	Tools/Resources	Owner
Development	Unit testing, debugging.	IDEs, debugging tools	Developers
Staging/Testing	System and regression testing.strategies	Selenium	QA Team
Production (UAT)	User Acceptance Testing.	Final game build, production-like setup	Product Owner

1.3.4 4. Time Allocation

1.3.4.1 Effort Estimation:

- Test Planning and Preparation: 10–15% of the total testing effort.
- Test Case Execution: 50–60% of the total testing effort.
- Bug Reporting and Retesting: 20–25% of the total testing effort.
- Regression Testing: 10–15% of the total testing effort.

1.3.4.2 Example Time Allocation Table:

Activity	Effort (%)	Responsible Team Member(s)
Test Planning	10%	Test Lead
Test Case Creation	15%	QA Testers
Test Execution	50%	QA Testers, Automation Engineers
Bug Reporting/Retesting	20%	QA Testers, Developers
Regression Testing	15%	QA Testers, Automation Engineers

1.3.5 5. Budget Allocation

1.3.5.1 Key Budget Considerations:

1. Personnel Costs:
    - Salaries for QA testers, automation engineers, and other team members.
2. Tool Licences:
    - Costs for test management, automation, and performance testing tools.
3. Hardware and Devices:
    - Costs for purchasing or renting devices for compatibility testing.
4. Training:
    - Costs for training team members on new tools or methodologies.

1.4 4. Testing Approach

Describe the type of testing to be performed (such as unit testing, integration testing, or user acceptance testing) and the methodologies to be used.

1.4.1 Types of Testing:

- Unit Testing: Validate individual components (e.g., battle mechanics, inventory system).
- Integration Testing: Test interactions between different game modules (e.g., multiplayer functionality, db module and inventory system), from developers' point of view.
- System Testing: Verify the game as a whole from the user's point of view, including gameplay, UI, and performance.
- Regression Testing: Ensure that new updates or fixes do not introduce new bugs.
- User Acceptance Testing (UAT): Gather feedback from players to ensure the game meets expectations.

1.4.2 Methodologies:

- Manual testing for gameplay and user experience.
- Automated testing for repetitive tasks (e.g., regression testing).
- Exploratory testing to uncover edge cases and unexpected issues.

1.5 5. Timeline and Schedule

Create a testing timeline (if suitable) that includes key milestones, dependencies, and deadlines for different testing phases. The test arrangements for two extremes of software development process are listed below:

1.5.1 1. Waterfall Model

In the Waterfall model, testing is a distinct phase that occurs after the development phase is completed. The timeline is linear, and testing activities are planned sequentially.

1.5.1.1 Dependencies:

- Completion of game development milestones.
- Availability of testing environments and tools.
- Timely resolution of reported bugs.

1.5.1.2 Key Phases and Timeline:

1. Test Planning and Preparation: Weeks 1–2
    - Review requirements and design documents.
    - Create a detailed test plan.
    - Write test cases and scenarios.
    - Set up the testing environment and tools.
2. Unit Testing: Weeks 3–4
    - Test individual components or modules developed by the team.
    - Verify that each module meets its functional requirements.
3. Integration Testing: Weeks 5–6
    - Test interactions between integrated modules.
    - Ensure data flow and communication between components work as expected.
4. System Testing: Weeks 7–8

- Test the entire system as a whole.
- Validate that the system meets functional and non-functional requirements.

5. User Acceptance Testing (UAT): Weeks 9–10
   - Conduct testing with end-users or stakeholders.
   - Validate that the system meets business requirements and is ready for release.
6. Bug Fixing and Regression Testing: Weeks 11–12
   - Fix bugs identified during testing phases.
   - Perform regression testing to ensure fixes do not introduce new issues.
7. Final Validation and Release: Week 13
   - Conduct final exhaustive testing for core features to ensure the system is stable.
   - Prepare the system for deployment.

1.5.1.3 Waterfall Model Timeline Example:

Phase	Duration	Activities
Test Planning & Preparation	Weeks 1–2	Create test plan, write test cases, set up environment.
Unit Testing	Weeks 3–4	Test individual modules.
Integration Testing	Weeks 5–6	Test interactions between modules.
System Testing	Weeks 7–8	Test the entire system for functionality and performance.
User Acceptance Testing	Weeks 9–10	Validate the system with end-users.
Bug Fixing & Regression	Weeks 11–12	Fix bugs, perform regression testing.
Final Validation & Release	Week 13	Conduct exhaustive testing for core features, prepare for deployment.

1.5.2 2. Agile Model

In the Agile model, testing is integrated into the development process and occurs iteratively throughout the project lifecycle. Testing activities are planned for each sprint (unit of development time to finish certain goals, usually 2-week), and the timeline is flexible to accommodate changes in requirements.

1.5.2.1 Key Phases and Timeline:

1. Sprint Planning and Preparation: Day 1 of each sprint
   - Collaborate with the team to define acceptance criteria for user stories.
   - Write initial test cases and scenarios.
   - Prepare the test environment and tools.
2. Testing During the Sprint: Ongoing (Daily)
   - Execute test cases for completed user stories.
   - Perform exploratory testing for edge cases.
   - Log and report bugs to developers.
   - Conduct regression testing to ensure new changes do not break existing functionality.
3. Mid-Sprint Testing Milestones: Midpoint of each sprint
   - Perform integration testing for newly developed features.
   - Update test cases based on changes or clarifications.
4. End-of-Sprint Testing Activities: Last 1–2 days of each sprint
   - Conduct final acceptance testing for all completed user stories.
   - Perform system testing to validate the overall functionality.
   - Execute smoke tests to ensure the application is stable for the sprint review/demo.
5. Post-Sprint Activities: After each sprint
   - Perform regression testing on the integrated build.
   - Update the test suite with new test cases for future regression testing.
   - Automate additional test cases, if applicable.

1.5.2.2 Agile Model Timeline Example for a 2-Week Sprint:

Day	Activity
Day 1	Sprint planning, finalize acceptance criteria, write initial test cases.
Day 2	Begin exploratory testing, prepare test environment, start test case execution.
Day 3–4	Execute test cases for completed user stories, log bugs, perform regression testing.
Day 5	Mid-sprint integration testing, update test cases, exploratory testing.
Day 6–8	Continue test case execution, bug reporting, and regression testing.
Day 9	Final acceptance testing, system testing, smoke testing for sprint review.
Day 10	Sprint review/demo, retrospective, prepare test summary report.
Day 11	Post-sprint regression testing, update test suite, automate test cases.
Day 12	Collaborate on backlog refinement, prepare for the next sprint.

1.5.2.3 Agile Model Key Milestones:

- Testing is continuous and iterative.
- Each sprint delivers a potentially shippable product increment.
- Testing activities are planned and executed within each sprint.

1.6 6. Risk Assessment and Mitigation

Identify potential challenges and obstacles that could impact testing, and the strategies to address or minimize these risks.

1.6.1 Potential Risks:

1. Delays in Development: Could impact the testing schedule.
   - Mitigation: Regular communication with the development team to adjust timelines as needed.
2. High Bug Volume: May overwhelm the testing team.
   - Mitigation: Prioritize critical bugs and allocate additional resources if necessary.
3. Compatibility Issues: Game may not perform well on all supported platforms.
   - Mitigation: Conduct thorough compatibility testing early in the process.
4. Insufficient Resources:
   - Risk: Lack of testers, tools, or devices.
   - Mitigation: Prioritize critical test cases and borrow resources from other teams if needed.
5. Intrinsic Untestability:
   - Risk: Some features cannot be easily tested with automated testing tools. For example, features with highly complex algorithms, AI-driven behaviour, or dynamic decision-making processes that are difficult to predict or replicate in a test environment.
   - Mitigation: Make the software be able to expose as many internal states as possible. For example, for a game AI, provide a debug mode that outputs and logs decision-making steps to help testers understand its behaviour.

1.7 7. Success Criteria

Establish clear criteria for evaluating testing progress and determining when testing objectives have been successfully achieved.

- All critical and high-priority bugs are resolved.
- Game meets performance benchmarks on all supported platforms.
- Positive feedback from UAT participants.
- All test cases pass with no major issues.
- Game is approved for release by stakeholders.

1.8 8. Reporting Requirements

Outline how test results will be recorded, monitored, and shared with stakeholders throughout the testing process.

1.8.1 Documentation:

- Description of the test cases in the code repository or dedicated test repository.
- Test case execution reports.
- Bug reports with detailed descriptions, steps to reproduce, and screenshots/videos.
- Weekly progress updates to stakeholders.

1.8.2 Communication:

- Regular meetings with the development team to discuss testing progress and blockers.
- Final test summary report to stakeholders before release.

This test plan provides a structured approach to ensure the Pokeman game is thoroughly tested and ready for release.