HOCHSCHULE CAMPUS WIEN
UNIVERSITY OF APPLIED SCIENCES

Quality & Testing Handbook

Agile - Scrum Software Project

Mastermind game with CLI and GUI, developed with Agile practices and a strong emphasis on software quality.

Submitted by

Adnan Eminovic
Larissa Herbst
Markus Maximus
Seyed Hossein Meymandi Nezhad
Laurenz Stelzl
Mohammadali Vaezi

Vienna, 2025

Study course:	536.308. Software Quality and Test
at:	Hochschule Campus Wien,
supervised by:	Prof. Philipp Rosenberger

Project GitHub Repository:
https://github.com/alivaezii/mastermindgame
Documentation and Wiki:
https://github.com/alivaezii/mastermindgame/wiki

List of abbreviations

Abbreviation	Full Term	Description
API	Application Programming Interface	An interface enabling communication between software components
CI	Continuous Integration	Automated process of building, testing, and validating code changes
CI/CD	Continuous Integration / Continuous Deployment	Automated pipeline for integration, testing, and deployment
CLI	Command Line Interface	Text-based user interface for interacting with the system
DoD	Definition of Done	A set of criteria defining when a feature is considered complete
E2E	End-to-end	Testing method validating the complete system workflow
ET	Exploratory Testing	Experience-based testing without predefined scripts
ETC	Exploratory Test Case	Documented scenario used in exploratory testing
GUI	Graphical User Interface	Visual interface for user interaction
IDE	Integrated Development Environment	Software environment for development and debugging
IEEE	Institute of Electrical and Electronics Engineers	Organization providing software engineering standards
ISO	International Organization for Standardization	Global standards organization
JSON	JavaScript Object Notation	Lightweight data; interchange format
KPI	Key Performance Indicator	Metric used to evaluate quality and performance
LOC	Lines of Code	Measure of code size
PvC	Player vs Computer	A game mode where the player competes against the system
PvP	Player vs Player	A game mode where players compete against each other
QA	Quality Assurance	Activities ensuring software quality standards

TDD	Test-Driven Development	A development approach where tests are written before code
UI	User Interface	Components enabling user interaction
UT	Usability Testing	Evaluation of ease of use and user experience
UX	User Experience	Overall user perception and interaction quality
VCS	Version Control System	System managing source code versions (e.g., Git)
CI Pipeline	Continuous Integration Pipeline	Automated workflow executing tests and quality checks
JUnit	Java Unit Testing Framework	Unit testing framework for Java applications
pytest	Python Testing Framework	Unit testing framework for Python applications

Table of contents

1. Team & Roles

- **Product Owner:** Defines game requirements and acceptance criteria; prioritizes features.
- **Scrum Master:** Facilitates Agile process; ensures testing practices follow the test strategy.
- **Senior Developer:** Leads architecture design, performs code reviews, and mentors juniors on writing testable code.
- **Junior Developers:** Implement game features and associated unit tests; assist in test design and documentation.
- **Tester (QA Engineer):** Plans the overall test strategy and test cases; executes tests (unit, integration, exploratory, GUI, E2E, usability); leads formal inspections and reports on quality metrics.

2. Introduction and Scope

This Test Handbook documents the software testing strategy, methodologies, and results for the Mastermind Game project. It is intended as both an educational resource in software quality engineering and a practical test plan for the development team. The scope covers all levels of testing, from code-level checks to user experience evaluations, aligned with Agile development and Continuous Integration/Continuous Delivery (CI/CD) practices.

The Mastermind game implementation includes a core game engine, a Command Line Interface (CLI), a Graphical User Interface (GUI) prototype, and a persistent scoreboard (for high scores). Given the criticality of game logic correctness and user experience, a multi-faceted testing approach is employed. This handbook outlines our test policy, test design, test execution (static, dynamic, and exploratory), test results, key metrics, and continuous improvement plans. It also provides templates and examples for test documentation (see Appendix) to ensure clarity and repeatability in testing.

Agile Context: The project was developed in sprint iterations under Scrum. Testing activities were integrated throughout development ("shift left" approach), meaning tests and reviews were done continuously rather than waiting until the end. Every code change underwent automated testing in a CI pipeline, enabling rapid feedback and high confidence in software quality. Quality criteria are also part of the team's Definition of Done (DoD); no user story is complete without meeting all test and quality benchmarks (detailed below).

Note: Throughout this document, we reference relevant software testing standards and principles (e.g., ISTQB's testing categories and IEEE/ISO test process standards) to ensure our approach aligns with industry best practices. We also employ the Goal, Question, Metric (GQM) model to connect our testing goals with specific questions and measurable metrics. For example, a goal of "reliability" prompts the question, "Are we testing enough of the code?" We answer this by measuring code coverage.

3. Test Policy and Strategy

Testing Objectives: Our test policy emphasizes early defect detection, automation, and continuous quality. We aim to achieve a high level of reliability and maintainability for the Mastermind game. Concretely, the objectives are:

- **Robustness:** Detect and fix defects in game logic and UI before release, preventing user-visible failures.
- **Coverage:** Exercise all critical code paths (target ≥85% code coverage in automated tests).
- **Consistency:** Ensure the game works across supported Python versions (3.10 and 3.11) and environments.
- **Maintainability:** Enforce coding standards (PEP8 via flake8) and prevent complexity from growing unchecked.
- **Transparency:** Make test results and quality metrics visible to the whole team (through CI reports and badges).

Testing Principles: We adopt the "Test Pyramid" strategy: more fast, low-level tests (unit tests) and fewer high-level tests. This means:

- **Unit tests** form the broad base of the pyramid (many tests for isolated functions).
- **Integration tests** cover combinations of modules (moderate number of tests).
- **UI/E2E tests** are kept to a necessary minimum (a few end-to-end scenarios) due to their complexity and maintenance cost. This balance follows widely accepted best practices that favor more unit tests over fragile end-to-end tests to keep feedback fast and reliable.

Each test is designed in accordance with IEEE/ISO guidelines for test case design and clear traceability. We use the Given, When, Then structure for test cases to clarify setup, action, and expected outcome.

Deterministic testing is a strict rule: randomness is controlled using fixed seeds or dependency injection to ensure tests are repeatable. For example, when testing random secret code generation, we seed Python's random number generator to produce a predictable sequence.

Test Types Overview: The test strategy includes a mix of static and dynamic testing techniques:

- **Static Testing:** Code reviews and inspections are performed on critical modules without executing the code. We also use automated static analysis tools (flake8 for linting, black and isort for code formatting) to catch style issues, code smells, and potential errors early.

- **Dynamic Testing:** Various levels of runtime testing are done, including:
  - **Unit Tests**: focused tests of individual functions and classes (e.g., core logic in engine.py, scoreboard.py).
  - **Integration Tests**: tests involving multiple components together (e.g., CLI commands invoking the game engine and scoreboard).
  - **System Tests / End-to-End Tests**: complete workflow tests of the entire application (CLI or GUI) from start to finish.
  - **Regression Tests**: re-running our full test suite whenever changes are made, to ensure new code doesn't break existing functionality.
  - **Exploratory Tests**: unscripted testing to uncover unforeseen issues.
  - **Usability Tests:** evaluations of the user interface and experience.

- **Additional Approaches:** We incorporate risk-based testing (prioritizing testing efforts based on the risk and impact of potential failures) and embrace elements of test-driven development (TDD) when adding new features. Risk-based testing means that features with higher failure risk or user impact (e.g., the core scoring algorithm) receive more intense testing and review. TDD is practiced on critical modules: developers sometimes write a unit test first for a new game rule, see it fail, then implement just enough code to pass the test in a short cycle. This ensures test coverage grows alongside code and helps to design simpler, testable code.

Roles and Responsibilities: Quality is a whole; team responsibility. Developers write and maintain unit tests for the code they implement. The dedicated Tester designs broader test scenarios (integration, exploratory, usability) and coordinates test execution. The Product Owner helps define acceptance criteria for features, which the team translates into test cases. During code reviews, all reviewers consider testability and may request additional tests for edge cases. This shared responsibility ensures that testing is not siloed.

Workflow and Quality Gates: We have integrated testing into our development workflow via CI/CD:

- Developers run tests locally and use pre-commit hooks to auto-run linters (flake8) and formatters (black, isort) before any commit is made. This prevents simple quality issues from even entering the codebase.
- Every push or Pull Request triggers a GitHub Actions CI pipeline that runs the whole test suite on multiple Python versions, checks coding style, and computes code coverage.
- We have defined Quality Gates that must be passed before changes are merged. All automated tests must pass, linting/style checks must report no errors, and code coverage must remain above 85%. If any gate fails, the CI marks the build as failed, blocking the merge ("red build cannot be merged to main"). These gates enforce our Definition of Done criteria.
- **Definition of Done (DoD):** A user story or feature is considered "done" only when:
    - All new and existing unit/integration tests pass (100% pass rate).
    - Code coverage is ≥ 85% (no significant drop due to new code).
    - No new lint or static analysis issues are introduced (code conforms to style and quality guidelines).
    - Documentation (code comments, user docs) for the feature is updated.
    - The feature meets acceptance criteria verified by the Product Owner (including passing any specified acceptance tests).

By enforcing these gates in CI, we ensure that the software reaching the main branch maintains a consistent quality level. This automated quality control aligns with industry standards for high-reliability software and supports the Agile principle of "Done means releasable" on each increment.

Test Design & Traceability: We design tests to cover both typical scenarios and edge cases. For each requirement or user story, corresponding test cases ensure the implementation meets the specification (requirements coverage*).* We maintain traceability by tagging test cases (or test functions) with references to requirement IDs or user story IDs in comments or documentation. This follows IEEE 29119 guidelines for linking tests to requirements for full coverage. For instance, if the game rule "No duplicate colors when no duplicates option is set" is a requirement, we have specific tests to verify that rule (both at the unit level for logic and at the integration/E2E level for gameplay).

We also leverage the original Java version's JUnit tests as a basis: during migration to Python, each JUnit test was translated into an equivalent pytest function (with naming conventions adjusted). This ensured that all original requirements tested in the Java version are also tested in the Python version. For example, a JUnit test testSecretGuess() became Python test_secret_guess_correct() in pytest, using Python's assert in place of JUnit's assertions. Each such test function includes a comment referencing the original requirement or test case ID for traceability.

Toolchain: The testing toolchain is built around Python's ecosystem:

- **Pytest**: primary test framework (for unit, integration, and even simple E2E tests in CLI). Pytest's fixtures are used for setup/teardown where needed, and tests are organized into modules that reflect the application's modules.
- **Pytest-cov**: coverage plugin to measure code coverage of tests. The CI produces a coverage report (see Appendix for sample).
- **Flake8**: linter to catch coding standard violations or potential bugs (e.g., unused variables).
- **Black and isort**: automated formatter and import sorter to maintain consistent code style.
- **Mypy**: (optional) static type checker to ensure type hint correctness (we used type hints in critical functions and verified no type errors; the "no type errors" KPI was tracked, meaning the code passes static type analysis).
- **GitHub Actions**: automates running all the above tools on each commit/PR. It uses a matrix to test on different Python versions and possibly different OS (to ensure portability). The CI workflow also builds a Docker image of the CLI game for a consistent runtime.
- **Docker**: used to containerize the application for testing in an isolated environment. We created a Dockerfile so anyone can run the Mastermind CLI in the same environment as CI, eliminating "it works on my machine" issues. Docker is also used in CI for deploying the

test environment (ensuring the same dependencies and configurations on every run).

Our test strategy is regularly reviewed against industry standards and course guidelines. We considered the ISTQB testing principles (for example, understanding that exhaustive testing is impossible, so we prioritize tests based on risk and equivalence classes, and the principle of early testing, which is why we conduct code inspections and unit tests early). We aligned our documentation with IEEE 829/IEC 29119 formats for test plans and test cases where applicable.

4. Static Testing: Formal Code Inspection

Static testing involves examining code and documents without executing them. The centerpiece of our static testing is a formal code inspection of critical code, supplemented by automated static analysis tools.

Formal Code Inspection: We conducted a structured code review on the engine.py module, which contains the core game logic for Mastermind. This inspection was carried out following a process similar to the IEEE 1028 standard for software inspections (which defines specific roles and steps for formal reviews):

- **Roles:** A Senior Developer served as the Inspection Moderator. The Author was the Junior Developer responsible for writing engine.py. Two Reviewers, the Tester and another Junior Developer, carefully examined the code. This division of roles ensured that multiple perspectives, author, tester, and reviewer, were applied.
- **Preparation:** Before the inspection meeting, each participant independently studied the code and noted any potential issues, questions, or ambiguities. We used a checklist to focus our review (covering correctness of game logic, adherence to coding standards, proper error handling, and completeness of documentation).
- **Inspection Meeting:** Over two sessions, the team walked through the code line by line. The Moderator guided the process, taking the code in logical segments (e.g., the guess validation function, the scoring function, etc.). As each segment was presented, the Reviewers raised any findings. Discussions were kept factual and issue-focused (avoiding blame on the author, in line with best practices for inspections).
- **Documentation of Findings:** All identified issues were logged in a review document, categorized by severity and type:
  - Defects, bugs, or logical errors in the code.
  - Improvements, suggestions for better design or clarity (e.g., refactoring opportunities).

  - Style Violations, deviations from Python style guidelines or project conventions (like naming, formatting issues).

Key Findings: The formal inspection proved invaluable. We discovered several issues that were not obvious at runtime:

- **Logic Bug:** In the function validate_guess(), if a guess shorter than the secret code is provided, the original code would index out of range (because it attempted to compare positions without checking length). We identified this potential bug and recommended adding a length check at the start of the function. This was subsequently fixed by raising a ValueError when the guess length is incorrect.
- **Edge Case Handling:** In the scoring calculation, we realized that if the player had zero attempts left (i.e., they failed to guess the code at all), our score computation might attempt a division by zero or give unintended results. We added a condition to gracefully handle the "no attempts remaining" case (e.g., awarding 0 points if the player exceeds the maximum attempts, rather than performing an invalid calculation).
- **Code Duplication:** The inspection noted some duplicated code in setting up game modes (e.g., similar logic between Player vs Player and Player vs Computer modes). We recommended refactoring these into a single helper function to improve maintainability.
- **Style Issues:** A few style issues were caught that automated linters also pointed out, such as overly long lines (beyond the 120-character limit we set) and some unused imports. For example, the engine.py file had an unused import and lines that exceeded our line length rule, which we corrected.
- **Documentation:** The inspection found that some public functions lacked docstrings or had unclear names. We added docstrings to functions like validate_guess() and score() to clarify their purpose, parameters, and return values.

By conducting this formal inspection, we achieved early defect removal, catching problems at the static analysis phase before they could manifest as bugs in testing or in production. This aligns with our quality objective of detecting issues as early as possible (the ISTQB principle of "early testing"). The inspection also spread knowledge among team members; reviewers became more familiar with the code, which later helped them write better tests and maintain it.

Static Analysis Tools: In addition to manual code reading, we employed tools:

- **Linting (Flake8):** Our code is continuously linted. Flake8 automatically caught issues like unused variables, imports, and styling problems. For instance, Flake8 flagged an unused variable in the scoreboard logic, which indicated a minor bug that we fixed (the variable was intended to be used in a loop but wasn't).

- **Formatting (Black, isort):** We used Black to auto-format code to a consistent style (PEP8 compliant) and isort to sort imports. This eliminated arguments over code style and allowed the team to focus on logic during reviews. It also means the repository consistently passes formatting checks in CI.
- **Type Checking (Mypy):** We treated type hints as a form of static documentation. Running Mypy (with strict optional settings) on key modules ensured that our function signatures and return types made sense and caught mismatches. Although Python is dynamically typed, adding this static analysis gave us greater confidence (for example, ensuring that functions that return tuples of ints actually return the correct types).

Overall, static testing (manual inspections + automated analysis) improved code quality and caught issues that dynamic testing alone might miss, such as design problems, naming confusion, and missing test coverage for edge cases. It complements dynamic testing by addressing the "building the product right" aspect (verification) even before we run the program. We plan to continue using formal inspections for major new modules or significant refactorings, while lighter peer code reviews (via GitHub Pull Request reviews) are used for routine changes.

5. Dynamic Testing

Dynamic testing is the execution of software with test inputs. We structured dynamic testing into multiple levels (unit, integration, system) and ensured they are all run regularly as part of regression testing. Our dynamic tests are automated with Pytest for efficiency and consistency.

5.1 Unit Testing

Scope: Unit tests target the most minor testable parts of the software, primarily individual functions or methods in the game engine and scoreboard modules. For Mastermind, critical units include:

- The secret code generation and validation functions (e.g., generate_secret(), validate_guess()).
- The feedback calculation for "bulls and cows" (correct color in correct place vs correct color in wrong place).
- Score calculation logic (points awarded based on number of attempts, etc.).
- Utility functions (like input validators, format converters).
- The Scoreboard class methods for saving/loading high scores.

Design: Each unit test is written to isolate the function under test:

- We avoid external dependencies in unit tests. For example, tests for the engine logic do not require the GUI or CLI; they call the engine functions directly.
- Where needed, we use stubs or mocks to isolate units. However, in this project, many units are pure functions (e.g., computing bulls and cows), so explicit mocking was minimal. For interactions with the filesystem (e.g., scoreboard saving to file), tests use Python's tempfile or Pytest's tmp_path fixture to avoid altering real data.
- We follow the Arrange; Act; Assert (Given; When; Then) structure: first set up inputs, then execute the function, then assert the output or state is as expected. Each test is named descriptively (e.g., test_validate_guess_rejects_wrong_length() clearly indicates what is being verified).

Examples:

- Testing the feedback mechanism: We verify that if the secret is "RGBY" and the guess is "RBGY", the function returns two bulls and two cows (since two colors are in the correct position and the other two are correct colors in wrong positions). The unit test calls validate_guess("RGBY", "RBGY") and asserts the result equals (2, 2).
- Testing boundary conditions: For secret length N, we test that any guess of length ≠ N raises a ValueError. Similarly, if the game rules forbid duplicate colors, we test the function that checks guesses to ensure it flags duplicates appropriately.
- Testing score computation: If the player guesses correctly on the first try, the score formula should give maximum points. We simulate scenarios (e.g., correct on 1st try vs. on last try) and verify that the computed score matches the expected outcome.

Execution: Unit tests run at high speed (each test runs in milliseconds). We currently have a comprehensive suite of unit tests (dozens of test cases) covering typical cases and edge cases for all core functions. This gives us a solid foundation; if any low-level logic breaks, a unit test will likely catch it immediately. The code coverage from unit tests alone is high (the engine module is ~98% covered by unit tests, and the scoreboard is ~95%).

Dynamic Test Data: We use fixed test data in unit tests for consistency. For randomness (like secret code generation), we inject a seed or override the random generator in tests. For example, in testing generate_secret(), we set random to true.seed(0) before calling it so that it produces a known sequence of colors, which we can assert. This makes tests deterministic.

5.2 Integration Testing

Scope: Integration tests verify that multiple components work together as intended. We focus on the integration between:

- The CLI interface and the game engine.
- The game engine and the scoreboard (persistence).
- (Potentially in the future, the GUI and the engine, though currently our GUI testing is mainly done in E2E manual scenarios due to a lack of automation).

CLI Integration Tests: We created tests that simulate using the command-line program with various arguments and inputs, to ensure the CLI front-end correctly invokes the underlying game logic and handles output:

- For example, we test the help option by invoking the CLI (using Python's subprocess or the click testing capabilities if the CLI is built with click) and capturing the output. We assert that the help text contains all expected usage instructions (to verify the integration of the CLI parser and the help documentation).
- We test starting a game in CLI with specific options (like length 4; no duplicates), but instead of a human playing, we automate inputs. One approach is to feed a sequence of inputs to the CLI program (e.g., using pexpect or simply simulating input via monkeypatch in the test). The test then reads the printed outputs to verify that the game progressed correctly.
- Example: Launch the CLI in test mode, simulate a sequence of guesses (some wrong, then the correct code), and ensure that after the correct guess, the CLI prints the victory message and the final score. This checks the integration of the game loop, engine logic, and CLI output formatting.

Engine Scoreboard Integration: We test that when a game ends, the score is correctly written to the scoreboard (a JSON file) and can be retrieved:

- In a controlled test, we run a quick game (perhaps by calling an internal function to simulate a finished game with a given score) and then check that the scores.json (or equivalent) file has the new entry. We use a temporary file for this test to avoid interfering with real data. The test asserts that after saving, reading the scores back yields the score that was just added.
- Another integration test is to start with a known scores.json content (we prepare a small JSON file with some scores), then call the function that adds a new score, and verify the file content is updated correctly. This ensures our file I/O and JSON parsing integration works.

GUI Integration: Full GUI integration tests (automated) are challenging without specialized tools. We did not have a fully automated GUI test in this iteration beyond simple launch tests. Still, we plan to use tools like **PyAutoGUI** or Selenium (with a UI driver) in the future to automate clicking buttons and reading screen output for the Pygame-based GUI. For now, GUI integration was primarily tested manually in E2E scenarios and through unit tests of the GUI logic (e.g., ensuring that the GUI calls the engine with the correct parameters).

Findings: Integration tests have caught issues like:

- Mismatch between CLI options and engine parameters (e.g., passing a string "6" from CLI when the engine expected an int for code length, resolved by proper parsing).
- File handling errors (initially, our scoreboard save function didn't close the file properly; the integration test detected a file lock issue, which we fixed by using a context manager).
- Ensuring that error conditions propagate correctly: e.g., if the engine raises a ValueError for an invalid guess, the CLI catches it and prints a friendly message instead of crashing. An integration test was created to simulate an invalid guess via CLI and assert that the process exits gracefully with an error message.

Conclusion: Integration testing gives confidence that the modules interact correctly. These tests run in CI as well, though they are a bit slower than pure unit tests (primarily if they spawn subprocesses for CLI). We keep them manageable in number and scope. Together with unit tests, they form our "dynamic functional tests" verifying that both individual parts and their combinations work. This layered approach aligns with the test pyramid concept and ensures we catch issues that unit tests alone might miss (such as miscommunication between components).

5.3 System Testing (End-to-End)

System testing validates the entire system behavior against the requirements. In our project, we treat End-to-End (E2E) tests as system tests, exercising the whole application as a user would. (We dedicate §7 to detailed E2E scenarios, but a summary is given here for completeness.)

Our E2E testing includes:

- **CLI Full Game Scenarios:** We simulate a complete game in the CLI from start to finish. For example, a Player vs Computer game where the secret code is fixed to a known value. The test (automated or semi-automated) feeds guesses and verifies that the game outputs correct feedback each time, ending with the correct win/lose result and score. This tests the system's core functionality in a headless manner (helpful for automation).
- **GUI Full Game Scenarios:** We have manual test cases where a tester goes through the GUI: selecting game options, playing the game, and checking that the GUI responds correctly (displaying bulls & cows, showing win/lose screens, updating the high score). While these were not fully automated in code, they are documented and executed during system testing.

- **Cross-component concerns: System tests also check things like whether the final state of one component is correctly reflected system-wide**. For instance, after a game over in the GUI, does the high score appear on the High Scores screen (indicating that the GUI successfully read the scoreboard updated by the engine)?

System testing, especially via E2E, often uncovers integration issues that were not obvious in lower-level tests. It essentially verifies that the entire application meets the requirements as a whole (ISTQB would call this functional system testing, since we focus on game functionality and non-functional aspects like usability within those same flows).

Regression Testing: We don't treat regression testing as a separate test level, but as a process of rerunning tests. Every time new code is added or a bug is fixed, we run the whole test suite (unit, integration, and system tests) to catch regressions. Our CI pipeline is effectively a regression test harness; by running all tests on each commit, it ensures that previously passing tests continue to pass. If any test fails, developers are immediately alerted to a potential regression.

In addition, whenever we fixed a defect, we added a new test (or enhanced an existing one) to cover that specific scenario, preventing that bug from reoccurring ("regression test suite growth"). For example, when we fixed the scoreboard's off-by-one error, we added a test case to ensure the scoreboard now handles the boundary correctly. This practice aligns with the pesticide paradox principle: we update tests to catch new bugs, so over time our test suite becomes more robust and does not miss the same bug twice.

5.4 Smoke Testing

As part of our system testing strategy, we defined a small set of smoke tests to verify basic functionality after each deployment or significant change quickly. A smoke test is a shallow, broad check intended to confirm that the application's major features *"smoke" (start) without immediately catching fire*.

Our smoke tests (which can be done manually or automated) include:

- **Application Launch:** Does the CLI start without errors, and does the GUI window open without crashing? (This catches issues like missing dependencies or syntax errors that unit tests might not reveal in isolation).
- **Basic CLI Game Flow:** Run a very short game where the secret is known, and the first guess is correct. The program should process the guess, declare a win, and exit. This ensures the game loop can run at least one iteration.

- **Basic GUI Interaction:** Launch the GUI, simulate clicking through to start a game, then immediately exit. Ensure there are no obvious errors (e.g., the GUI not responding to the Start button).
- **Help/Version Commands:** Running mastermind; help and mastermind
- version to ensure those core commands execute correctly (no crashes, correct output).

We have automated some of these using CI scripts (for example, after running unit tests, the CI might, as a final step, invoke mastermind to help ensure the packaging is correct and the program can start). Smoke tests run quickly and serve as a build verification test suite. If a smoke test fails, we know something is fundamentally wrong (and likely many other tests would fail too, but smoke tests give a faster check).

These smoke tests are the first to run in any environment, e.g., after deploying the app to a staging environment, a quick smoke test is performed before proceeding to more detailed tests. In CI, we plan to separate smoke tests from complete tests so that smoke tests can run on every push (very fast), while the full suite runs nightly or on merges due to longer runtime. This ensures rapid feedback for basic errors and comprehensive coverage for deeper issues.

6. Exploratory Testing

Exploratory testing is a hands-on, unscripted approach in which testers interact creatively with the software to find defects that predefined test cases may not cover. For the Mastermind game, after formal test cases were executed, we allocated time for exploratory sessions focusing on *error-prone and unpredictable areas*. The tester used experience and intuition to "explore" the application in ways typical of both typical users and curious testers.

We structured exploratory testing with charters, short mission statements describing what to explore and what risks to look for. Two example exploratory test charters we executed:

6.1 ETC- 1: Invalid Guess Inputs

Charter: *"Explore how the game handles malformed or invalid player guesses."*
Description: The tester tries inputs that violate the game rules to see if the system robustly handles them. This includes:

  - Entering guesses that are too short or too long compared to the expected code length.

- Using characters/colors that are not allowed (e.g., entering a letter not in the color code range, or a number when letters are expected, depending on mode).
- Submitting an empty guess (just pressing enter without any input).

  Expected Behavior: The game should gracefully reject invalid inputs, e.g., by displaying an error message and prompting for another guess, rather than crashing or exhibiting incorrect behavior. The rules for input length and allowed characters should be enforced at all interface levels (the CLI should validate input length, and the GUI should prevent illegal input).

  Results: The game handled these scenarios correctly. For instance, an empty input in CLI triggers a friendly message "Invalid input, try again." A guess that's too long gets truncated by the input handler (and a message explains the maximum length). A disallowed character results in an immediate prompt informing the player of valid options. No crashes occurred. We did note a minor improvement: adding a line in the game's help or instructions explicitly stating the required format (e.g., "You must enter exactly N characters for a guess") to help users avoid these errors. This suggestion was taken into our backlog for a documentation update.

6.2 ETC- 2: No-Duplicates Rule Edge Cases

Charter: *"When the no duplicate colors option is enabled, explore edge cases around duplicate inputs."*
Description: Here, we looked at the interplay between game settings and user behavior. We launched a game in *Player vs Computer* mode with the "no duplicates" option (meaning the secret code will have all unique colors). We then attempted to break or mislead this rule:

- On the GUI, try to select the same color twice for the secret code (should be prevented by the UI).
- In gameplay, attempt to enter a guess that contains a duplicate color, even though duplicates are disallowed.
  Expected Behavior: The system should enforce the no duplicate rule consistently:
- In the GUI, ideally, the UI would not allow the user to pick duplicate colors when setting a secret (e.g., once a color is chosen, it could be greyed out for subsequent slots).
- If a duplicate does somehow get entered (especially possible in CLI, where manual typing is involved), the game should reject it with an appropriate message and not count that guess as an attempt.

  Results: The GUI behaved as expected: during secret code selection, once a color peg was used, the corresponding color button became disabled (greyed out), effectively preventing a duplicate secret. During guessing, the GUI similarly disallowed selecting the same color twice in a single guess when the no-duplicates rule was active. In the CLI, when we entered a duplicate letter guess (e.g., “R R G B” where the format expected unique letters), the program detected the violation and printed “Duplicates not allowed” and did not decrement the remaining attempts. The guess was not processed by the engine, thereby enforcing the rule. This was a successful outcome, confirming that our input validation layer is consistent with the game rules across interfaces. No defects were identified in this area.

These exploratory sessions demonstrated that the game was relatively robust to unexpected inputs and “naughty” user behaviors. The fact that we found no crashes and only minor suggestions is a positive indicator of quality. It suggests our earlier scripted tests already covered many edge cases, and the defensive coding (input validation) was effective.

Nonetheless, exploratory testing added value by going beyond the script; it prompted us to think about unusual interactions (such as quickly restarting the game multiple times or trying to cheat by editing the high score file). As another example, the tester tried alt-tabbing and resuming the GUI, ensuring the game could handle window focus changes (it did). We kept notes of all exploratory actions and outcomes for the record.

In the future, we plan to integrate exploratory testing regularly, especially after significant changes or before releases. We may also adopt session-based test management for exploratory testing (time-boxed sessions with clear charters and logging of test ideas and outcomes) to make it more systematic. While not as easily measurable as scripted tests, exploratory testing taps into human creativity and can uncover issues that automated tests would not even consider.

7. End-to-End Testing

End-to-end (E2E) tests verify the application's complete flow in scenarios that mimic real usage. These tests treat the system as a black box, providing inputs through the UI (CLI commands or GUI actions) and observing the outputs and side effects. E2E tests for the Mastermind game ensure that all components (UI, engine, scoreboard, etc.) work together seamlessly to deliver the intended user experience.

We designed and executed two main E2E test scenarios, one for each interface mode:

7.1 E2E- 1: Full CLI Game Victory

Scenario: *Player vs Computer via CLI, with a fixed secret code to ensure a deterministic outcome.*
Setup: Use the CLI to start a new game with known settings. We seed the random number generator (random.seed(42)) so that the secret code the computer chooses is predetermined (for example, "0123" using digits or equivalent colors if mapping digits to colors). We allow the default number of attempts (say 10).
Steps: The player (simulated by the test script) enters a series of guesses. In this scenario, we test an immediate win: the first guess entered is precisely the secret code "0123".
Expected Behavior: The game should recognize the correct guess, report four bulls (and zero cows), and end the game immediately with a victory. The CLI should then display the win message and the final score calculation. According to the game rules, if the player wins on the first attempt, they receive a base score (e.g., 100 points) plus a bonus for each remaining effort. With nine attempts left unused, if the bonus were 10 points per unused attempt, the total score would be 100 + 9*10 = 190 (hypothetically). The program should output something like "Congratulations, you cracked the code!" and show the score.
Results: The CLI behaved correctly. In our actual implementation, the scoring for unused attempts gave a slightly different bonus. Still, the test confirmed the correctness: the output included the congrats message and a score that matched the expected formula. The game loop exited after the win, and the process returned to the shell. We captured the CLI output and verified it contained the correct statements (using assertions in the test script). This E2E test validated that, from input parsing through engine logic to output formatting, the CLI path works as intended for a win scenario. It essentially covered what a real user would experience if they guessed the code right away.

7.2 E2E- 2: Full GUI Game with Leaderboard Update

Scenario: *Two-player game via GUI, testing the GUI gameplay and the high score persistence.*
Setup: Start the Mastermind game in GUI mode (running the Pygame interface). Choose the Player vs Player mode. Player 1 sets a secret code (we ensure they pick a known combination, e.g., [Red, Green, Blue, Yellow] in the first four slots). Player 2 will attempt to guess this code. We have the game configured for at most 10 attempts.
Steps: Using the GUI, Player 2 makes a sequence of guesses. We simulate, or manually perform, for example, three guesses: the first two are wrong, and the third is precisely the secret (meaning Player 2 wins in 3 attempts out of 10). After the guess is entered via the GUI (by clicking colored pegs and submitting), the GUI should show feedback (bulls and cows indicators) for each attempt. When the correct code is guessed, the GUI displays a “You Win” or game-over screen with the results. We then

navigate to the High Scores screen in the GUI.
Expected Behavior: Throughout the game, the GUI should correctly reflect the game state:

- After each guess, the correct number of bulls (correct color & position) and cows (correct color, wrong position) are indicated (often Mastermind GUI uses small pegs or text to show this). We expect these to match the engine's calculation.
- When the game ends with a win, a victory message is shown, and the final score is calculated. For example, if the base score is 100 and 7 attempts were unused (10 ; 3 attempts used = 7), the score might be 100 + 7*10 = 170. This score should be displayed to the players.
- Importantly, the score should be saved to the persistent scores file. We expect that after the game, if we open scores.json (or view it on the game's High Scores screen), the new score for Player 2 will be recorded.
  Results: The GUI test scenario passed on all points:
- The visual feedback after each guess was correct (we cross-checked bulls/cows counts with what the secret vs guess should produce).
- After the third guess was correct, the GUI showed a "Congratulations" pop-up with the final score of 170, which matched our expected calculation.
- We then clicked to view the High Scores, and the GUI listed the new score alongside the players' names. To double-check, we inspected the scores.json file on disk after exiting the game and found an entry corresponding to that game. This confirms the integration of GUI: Engine: Scoreboard: the engine signaled the end of the game and score, the GUI wrote it to file via the scoreboard module, and the GUI was able to read it back for display.

These E2E tests demonstrate that the entire application stack functions correctly for typical use cases. They effectively serve as acceptance tests from a user perspective, verifying end-user requirements: one, that a user can win the game and see their score, and two, that the high score is preserved for future sessions.

E2E tests are critical because they test everything in tandem (from user input, through all layers, to output and data storage). However, they are also costly: they take longer to run, and can be harder to automate (especially GUI parts). We ran the CLI E2E test as an automated part of CI (since it can be scripted fully). The GUI E2E was done manually for this iteration, with detailed steps recorded so it can be repeated. In a professional setting, we would aim to automate GUI E2E tests as well, but that requires additional tools and was considered future work.

To ensure we at least cover basic end-to-end behavior automatically, we added a smoke *test variant that runs the game in the CLI and verifies it can complete* a minimal scenario (as described in §5.4). For the GUI, our future CI/CD improvements might include running the game in headless mode or using a Pygame testing framework (if available).

Traceability: Each E2E scenario was derived from a subset of the product's acceptance criteria. For example, a requirement "The game shall record the player's score in a persistent leaderboard on victory" is directly validated by E2E;2. This traceability ensures that for each high-level requirement, we have at least one end-to-end test confirming it in the integrated system (this aligns with validation: "Are we building the right product?").

In summary, the end-to-end tests gave us confidence that the Mastermind game not only passes unit and integration tests but also delivers the intended user functionality. They are a final safety net before releasing the software to actual users.

8. Usability Testing

Beyond functional correctness, we evaluated the usability of the Mastermind game, especially the GUI (Graphical User Interface) and, to some extent, the CLI (since even a command-line tool has usability aspects for the user). Usability testing focuses on the end user's experience: ease of learning, efficiency of use, error handling and help, aesthetic and enjoyment factors, etc. We conducted informal usability tests with representative users to gather qualitative feedback.

8.1 UT- 1: GUI Learnability and Intuitiveness

Objective: Assess how easily a new user can navigate the game's GUI without prior instruction, and identify any points of confusion. **Method:** A tester with no prior exposure to our Mastermind game was asked to use the GUI to play a round, while thinking aloud. We specifically observed:

- Can the user figure out how to start a game (is the "Start" or mode selection obvious)?
- Does the user understand how to select colors for the secret code and for each guess? Are the UI controls (color pegs, submit button) intuitive?
- Can the user interpret the feedback (bulls & cows indicators) correctly from the interface?
- Overall, is the sequence of actions (set up game: enter guesses: see results) clear from the on-screen prompts and labels? **Expected:** Based on standard UX principles, the GUI should use familiar controls and clear labels. For instance, colored buttons or

pegs should visually indicate selection, a “Submit” button to enter a guess should be clearly labeled, etc. The workflow should be discoverable without reading a manual (since many casual users won’t).
Results: The new user was able to start and play the game without external help, indicating good learnability. Positive observations:
  - The main menu and game mode options were easily understood.
  - The action of clicking a color to select it for a slot was discovered quickly, and the user liked the immediate visual of the color appearing in the slot.
  - The feedback mechanism (small pegs showing bull and cow counts after a guess) was recognized by the user from prior knowledge of Mastermind; our labels and legend helped confirm understanding.
  - No critical confusion was encountered, but one minor point of feedback was that the user did not immediately realize they had to click “Submit Guess” after arranging the colors; they waited a few seconds, expecting auto-submission. To improve this, we decided to add a tooltip or a gentle highlight on the submit button after a guess is arranged, to draw attention. We noted this as a UI enhancement.

We made a few adjustments based on this test: for example, adding tooltips on hover for important buttons (e.g., the “Submit Guess” button now shows a tooltip "Click to submit your guess"). We also clarified text, like changing “Start” to “Start Game” for clarity. These changes align with usability heuristics (visibility of system status, and user control/feedback).

8.2 UT- 2: CLI Guidance and Error Messages

Objective: Even a CLI should be user-friendly for those who prefer or require text mode. We tested the command-line interface’s usability by evaluating help availability and the clarity of error messages. **Method:** We had a tester run the CLI with various commands:

- Check the help text via mastermind; help.
- Intentionally misuse the CLI by providing incorrect arguments (e.g., mastermind; length;1 or an unknown option) to see what feedback is given.
- Observe the clarity of the instructions when the game starts in CLI (does it clearly prompt the user for input? Are error messages understandable if the user types something wrong?). **Expected:** The CLI should follow standard conventions:;; help should list all options with descriptions.
  - If the user enters an invalid command or option, the program shouldn’t crash or silently fail; it should print an error explaining the issue (for example, “Error: length must be ≥1”).

  - Once the game is running, prompts should be self-explanatory (e.g., "Enter your guess:" appears when it's time to input, and if you input the wrong format, an error message like "Invalid guess format, please enter four colors" should appear). **Results:** Our CLI passed these criteria:
  - The help output was comprehensive and formatted clearly (thanks to using Python's argparse or click library, it automatically listed usage, options like length,no duplicates, etc., with the help strings we provided).
  - For invalid arguments, the CLI printed error messages. For instance, running mastermind; length;1 resulted in an error message: "Error: length must be >= 1". This message is generated by our validation logic and was displayed on stderr. It correctly prevented the game from starting with an impossible length.
  - During gameplay, if an invalid guess was entered (as in the exploratory test, an empty guess or wrong length), the CLI output a friendly error and re-prompted, rather than throwing a Python exception. No raw stack traces or cryptic errors were shown to the user, which is good for usability (users see handled error messages).

The CLI's usability is obviously limited by its text-based interface, but for a technical audience, it was found to be adequate and well documented. We included examples in the help text (e.g., command usage), which the tester found helpful.

General Usability Notes: Across both interfaces, we considered accessibility and future improvements:

- The color-based code could be challenging for color; blind users. While not implemented yet, we brainstormed adding a mode or option (or simply using distinct shapes or patterns in addition to colors) to support color; blind accessibility on the GUI pegs. This is noted as a future enhancement.
- Feedback latency: The GUI responds immediately to clicks (thanks to Pygame event loop), but in earlier versions, we had a slight delay after submitting a guess while the engine processed. We optimized that function so the delay is negligible, an essential aspect of usability (performance is one of the non-functional requirements).
- Documentation: We provided a README and in-game help. The usability tests indicated these were sufficient, but more in-depth documentation (like a tutorial for first-time players) could enhance user onboarding. We included a quick rules summary in the GUI's help menu in response to this feedback.

Usability testing is inherently subjective, but it ensures that, beyond meeting functional specs, our game is pleasant and convenient to use. The changes we made (tooltips, more transparent labels, helpful error messages) all contribute to a better user experience, which is a key quality attribute (often neglected if one focuses only on functional tests). We plan to gather more user feedback as the project continues, possibly via surveys or more formal user testing sessions, especially if the game is released to a broader audience.

9. Test Results and Key Metrics: Release (v1.1.0)

After executing the comprehensive test suite and processes described, we collected results and Key Performance Indicators (KPIs) to evaluate the effectiveness of our quality efforts. Below is an analysis of the outcomes:

Test Execution Summary: All planned tests were executed by the end of the final sprint:

- **Unit Tests:** ~50+ unit test functions covering game logic and utility functions. *Result:* 100% passed.
- **Integration Tests:** A dozen integration tests (CLI and file integration). *Result:* 100% passed.
- **System/E2E Tests:** 2 main E2E scenarios executed (1 automated CLI, one manual GUI). *Result:* Passed (observed outcomes matched expected results).
- **Exploratory Sessions:** 2 sessions (documented in §5). *Result:* No bugs found; a few improvement suggestions made.
- **Usability Tests:** 2 sessions (GUI and CLI as in §8). *Result:* No major usability issues; minor UX tweaks implemented.

Code Coverage: We achieved approximately 90% overall code coverage with our automated tests. This exceeds our minimum goal of 85%. The coverage is not uniform across components:

- Core modules like engine.py have very high coverage (~98%) due to extensive unit tests.
- The scoreboard module is also well;covered (~95%, including integration tests for file I/O).
- The GUI module has lower coverage (~85%) since many GUI functions involve rendering and user events that are not easily hit by automated tests. We expected this and treated GUI testing more via manual tests.
- Some utility scripts or corner cases (around error logging, etc.) were around ~80%, and these were deemed acceptable as they are non-critical or simple pass-through code.

A coverage report (see Appendix) highlights a few lines that are not covered, e.g., some exception branches in CLI argument parsing and a debug logging call. We noted these for potential future test cases, but they are low priority (no impact on gameplay). Achieving near 100% coverage was not as crucial as ensuring critical logic is covered, and tests are meaningful (we avoided writing trivial tests just to hit 100%). According to industry insights, ~80- 90% coverage is considered very good for a project of this size, balancing thoroughness with effort.

Defects and Bug Metrics: Over the course of development and testing, all identified defects were resolved:

- Total defects found pre-release**:** *~5* (including the ones from code inspection and any discovered in testing). These included: the guess length bug, a score calculation edge case, an off-by-one error in scoreboard indexing, a minor bug with argument parsing defaults, and one logical oversight in duplicate enforcement.
- **Defects fixed:** 5/5 (100% of known issues were fixed before final release).
- **Post-release defects:** 0 (as of this writing, no new bug reports have been encountered after delivering the project, indicating a high defect removal efficiency).

We measure defect density (defects per 1000 lines of code) as another KPI. With ~5 defects across roughly 1000 LOC (the approximate size of the project), the initial defect density was ~5 per KLOC before fixes. After fixes, the known defect density is 0 per KLOC. This is a very favorable outcome, though it must be tempered by the fact that not all defects are equal, and some might still be latent. We attribute the low defect count to the rigor of our testing and inspection process.

Another metric, defect detection efficiency (DDE), is essentially the percentage of defects found internally (before release) vs the total found. Here, DDE is 100% since end users found no defects. This is a strong indicator of test effectiveness.

Continuous Integration Metrics: Our CI pipeline provides additional quality signals:

- **Build Pass Rate:** All final builds passed the pipeline checks. Earlier in development, a few builds failed (e.g., a new test would fail initially, or a lint error slipped in), but by final integration, every commit to the main branch was green. We maintained a near-90% pass rate on the first attempt for pull requests, rising to 100% in later stages as developers became more disciplined.
- **Average CI Duration:** Each CI run (including installing dependencies, running tests on two Python versions, linting, etc.) took about 2- 3 minutes. This fast feedback loop encouraged developers to run tests frequently. We note that the test suite execution (pytest) itself takes around 20-30 seconds; the rest is environment setup. This is efficient enough for our needs.
- **Coverage Trend:** We tracked coverage over time (via coverage badges or CI logs). It started at around 70% when only unit tests were present and rose to 90% as we added more tests. This upward

trend met our goal of improving test coverage each sprint, an application of the GQM approach where the goal was "improve thoroughness of testing", the question "is coverage increasing towards target?", and the metric "coverage % each sprint".

Quality Gates Outcome: By the end, every commit that was merged satisfied the quality gate:

- **Coverage ≥ 85%:** Achieved (no merge brought it below threshold).
- **Linting errors = 0:** Achieved (flake8 reported no new issues; we occasionally had a warning, which was fixed promptly).
- **All tests pass:** Achieved (no known failing tests at merge time).
- **No critical static analysis warnings:** Achieved (e.g., no security warnings or major code smells; our project is small and straightforward enough that tools like CodeQL or others didn't flag anything serious).

This effectively means the Definition of Done was consistently met, which, in turn, means that each feature and the final product meet the agreed quality criteria.

Test Effectiveness & Efficiency: Another aspect is how practical our tests were at finding bugs. Of the ~5 defects found,

- Code Inspection found two that tests hadn't caught (e.g., potential issues that we then wrote tests for).
- Unit tests caught 1 (the off-by-one scoreboard issue manifested as a failing test when we wrote a new test for a boundary condition).
- Integration/E2E testing revealed 1 (a mismatch in CLI argument use).
- Exploratory/usability found zero critical bugs, but did contribute improvements.

This shows that a combination of methods was necessary; relying on a single approach would not have identified all issues.

In terms of efficiency:

- Writing tests took effort, but they saved debugging time later. For example, once we had tests, we could refactor with confidence, knowing tests would catch any mistakes.
- The automated tests run quickly, meaning we can re-run them often. Contrast this with what would happen if we had only manual testing: it would be far slower, and likely some issues would slip through.

Key Testing KPIs Summary:

Metric	Value/Outcome
Total Test Cases (Automated)	~60 (including unit, integration, E2E)
Automated Test Pass Rate	100% (final release)
Code Coverage	~90% (engine 98%, GUI 85%)
Defects Found (Pre-release)	5 (all fixed)
Defect Removal Efficiency	100% (no known defect escapes to users)
CI Pipeline Pass Rate	~100% for main branch (after initial iterations)
Lint/Style Violations	0 remaining (all addressed)
Testing Effort	~30% of project time (planning, writing, executing tests, and fixes)

These results indicate a high level of software quality at project completion. Not only do the numbers look good (high coverage, zero known bugs, passing builds), but qualitatively, the team and stakeholders have confidence in the product. Thorough testing gives us confidence that the game meets its requirements and will provide a good user experience.

We also recognize that metrics are not the only measure of quality; for instance, 90% coverage doesn’t guarantee there are no bugs, but, in context and combined with our diverse testing methods, it suggests we've tested most of what matters. We will continue to monitor these metrics and any user feedback to catch any issues that might arise.

10. Continuous Improvement and Reflections

Software testing is not a one-time task but an ongoing process. In this section, we reflect on the strengths and weaknesses of the testing methods we employed and on how we plan to improve our quality approach going forward. This introspection follows a kaizen (continuous improvement) mindset and aligns with lessons from standards such as ISO 29119, which emphasize learning and adapting the test process.

- **Unit Testing:**
  Advantages: Fast execution and exact fault localization. Our unit

tests provided immediate feedback on code logic – if a function, like a score calculation, had a bug, a failing unit test pinpointed it. They also serve as documentation of intended behavior. Unit tests allowed us to quickly and repeatedly validate many scenarios (every day and edge cases). This early defect detection is extremely valuable (finding issues during development, aligned with the principle that fixing a bug is cheaper the earlier you see it).
Limitations: Unit tests operate in isolation so that they can miss issues arising from component or system integration. For example, all unit tests might pass individually, but the game could still fail if there's a mismatch in how components connect (which is why we needed integration and E2E tests). Also, writing comprehensive unit tests for every single edge can be time-consuming, and tests can become brittle when the code is refactored frequently (though good naming and focusing on behavior over implementation mitigate this).
Future Improvements: We plan to use parametrized tests in Pytest to cover more input combinations with less code (e.g., systematically test a range of secret/guess combinations using a single test function with multiple parameters). We will also periodically review and refactor our unit tests to ensure they remain relevant and clear as the code evolves (avoiding the "pesticide paradox" where tests become stale and ineffective). As new features are added, we will practice and adopt Test-Driven Development (TDD) even more strongly to write tests upfront and guide the implementation.

- **Formal Code Inspection (Static Review):**
  Advantages: The formal inspection caught design and logic issues that weren't obvious through dynamic testing alone. It enforced coding standards and improved documentation. One significant benefit is knowledge sharing; multiple people deeply understood the engine.py after the inspection, which lowered the bus factor (not just one person knows that code). It also found non-functional issues (such as code structure and duplication) that tests wouldn't flag. This method aligns with the principle "testing can show the presence of defects, not their absence." Inspections actually find things that tests might miss, such as missing requirements or confusing implementations, thereby complementing execution-based testing.
  Limitations: Inspections are time-consuming and require skilled reviewers. For a small project, we managed it, but doing a formal inspection for every module would not scale. It can also become a bottleneck if the team waits for reviews. If overdone, it might slow down development. Also, an inspection's effectiveness depends on the thoroughness and expertise of reviewers; it's a manual process. It could overlook something if the team is not careful or if the checklist is incomplete.
  Future: We will reserve complete formal inspections for high-risk or

critical code (risk-based approach). For less critical changes, we will conduct lighter-weight peer reviews**.** We intend to develop a standard code review checklist (in line with IEEE 1028 recommendations) so that even quick reviews have some rigor. Also, incorporating automated static analysis tools (such as a static security scan or code complexity analysis) can augment manual reviews, catching issues we might miss, including cyclomatic complexity hotspots and security vulnerabilities.

- **Integration and System Testing:**
  (Integration aspects were partly covered in unit and E2E reflections, but combining here for a holistic view.)
  Advantages: Integration tests and system/E2E tests ensure that the parts of the system work together. The advantage here is realism; these tests operate more like how a user or another system would use our software. They found issues with the interface between modules (e.g., data format mismatches and file writing coordination). End-to-end tests, in particular, gave us confidence that the most critical user journeys (such as playing and winning a game) function correctly. They are also a great way to demonstrate the working product to stakeholders (essentially doubling as acceptance tests).
  Limits: These tests are slower and more fragile. If the GUI changes (say, we redesign the interface), any GUI test script might break and need maintenance. When an E2E test fails, diagnosing it can be harder; it could be in the engine, the UI, or the test script itself. Integration tests require careful setup (such as preparing test files and simulating inputs), which can be complex. We encountered some flakiness initially in CLI integration tests when trying to capture output timing, which we had to stabilize by adding proper synchronization. Moreover, writing a small number of E2E tests is feasible, but exhaustive system testing of every scenario is impossible (therefore, we complement with other methods and risk-based selection of E2Es).
  Future: We plan to invest in UI test automation tools to make GUI E2E testing more practical, and for example, using Selenium with a headless browser for web UIs, or for our Pygame GUI, possibly a tool that can simulate user events. Alternatively, we might refactor the GUI to separate the logic, so we can test it with unit tests and have fewer items to verify manually. Also, we will implement smoke tests (as already discussed) to run a basic E2E on every build. This will catch catastrophic issues quickly without needing the full E2E suite each time. Essentially, a quick run of the game (maybe through CLI) on each build serves as a sanity check (smoke test), and the more detailed E2Es can run nightly or in a separate stage. Lastly, we will use contract testing where applicable (e.g., if the game had an API) to ensure integration points remain compatible.
- **Exploratory Testing:**
  Advantages: Exploratory testing is highly flexible and can find

"unknown unknowns." In our project, while it did not find severe bugs, it helped validate that our application handles weird scenarios gracefully. The advantage is that it uses human creativity to go beyond pre-written scripts. It can adapt on the fly – if the tester sees something odd, they can follow that hunch immediately. Exploratory testing also often finds *UX issues or requirements gaps* that a strict test case might not consider (for example, noticing that adding a tooltip would help the user, which isn't a "defect" per se but an enhancement). It's a suitable method for testing error handling and simulating real user behavior.
Limits: It's hard to measure coverage or completeness for exploratory testing. We can't easily quantify how much of the system's behavior space we explored. Its effectiveness largely depends on the tester's skill and knowledge. If not documented, some findings might be lost or not reproducible. We addressed documentation by taking session notes, but still, reproducibility is an issue (for instance, "some strange lag happened once, but we couldn't replicate it" might occur and be hard to pin down). Exploratory testing also doesn't automatically regress – i.e., you can't rerun an exploratory test; you'd have to do it again manually or convert a discovered issue into an automated test.
Future: We aim to make exploratory testing more structured by using Session-Based Test Management (SBTM)**.** This means we'll allocate time for boxed sessions, each with a specific charter (goal), and afterwards debrief and catalog what was tested and what was found. This brings more accountability and record-keeping to exploratory tests. Also, whenever an exploratory test finds a bug or even a strange behavior, we will write a formal test (unit or integration) for it if possible. That way, our automated suite grows from exploratory insights. We're also considering using automated exploratory tools, such as fuzz testers for input fields (e.g., fuzz the guess field with random strings to see if anything breaks). This can be seen as automating a part of exploratory testing (especially for robustness).

- **Usability Testing:**
  Advantages: It ensured the product was not only correct but also user-friendly. This is crucial for acceptance – a perfectly coded game is no good if users can't figure out how to play it. Usability testing helped us catch issues that no unit test would ever see (like a user not noticing a button). It also provided direct user feedback, validating our design choices. We improved the user interface iteratively through this feedback loop, likely increasing user satisfaction.
  Limits: Usability findings can be subjective. We tested with only a small sample of users (due to time constraints, just one or two people). This may not represent all user types. Also, it's a time-consuming process and not easily repeatable in automation. Every time we change the UI, we'd ideally want to test it again with fresh

eyes, but that isn't always feasible. Additionally, our testers were somewhat technical (our classmates), whereas a broader audience might reveal other usability issues. So there's an inherent limitation in not having professional UX studies or a larger user base.
Future: We propose to incorporate more formal user acceptance testing if the project continues (e.g., a small beta test with multiple users and a feedback survey). We also plan to add accessibility testing – for example, use tools or guidelines (WCAG for color contrast) to ensure our color choices in the GUI are distinguishable by color, for blind users. As mentioned, adding a color; blind mode would be one outcome. Another aspect is considering different platforms/resolutions for the GUI to ensure it's usable in various environments. Usability is an ongoing concern, so we will treat any user feedback from releases as high-priority items to either fix (if it's a clear issue) or consider (if it's a suggestion).

In summary, each testing method provided a unique value but also had drawbacks. The key lesson is that no single testing technique is sufficient. By combining them, we achieved a synergistic effect: automated tests provided reliability and quick checks, while manual and exploratory tests provided depth and user focus. This multi-pronged approach is often recommended in ISTQB and other testing standards, which call for a mix of static analysis, different test levels, and techniques to cover all quality aspects.

Our commitment is to continuous improvement**.** The software industry and tools evolve, and so should our test strategy. We will regularly reflect (for example, in sprint retrospectives) on questions like:

- Were there any production issues or late-found bugs? How did we miss them, and what can we change to catch that type of issue earlier?
- Are our tests becoming a maintenance burden? If so, how can we refactor them (e.g., using more fixtures or factory functions to reduce test code duplication)?
- Is the team fully engaged in the quality process? Perhaps we need additional training or to adopt new practices (for instance, Behavior-Driven Development (BDD) could involve non-technical stakeholders in defining tests in plain language).
- Can we leverage new tools? (E.g., property; based testing for specific algorithms, or static analyzers for code security, etc.)

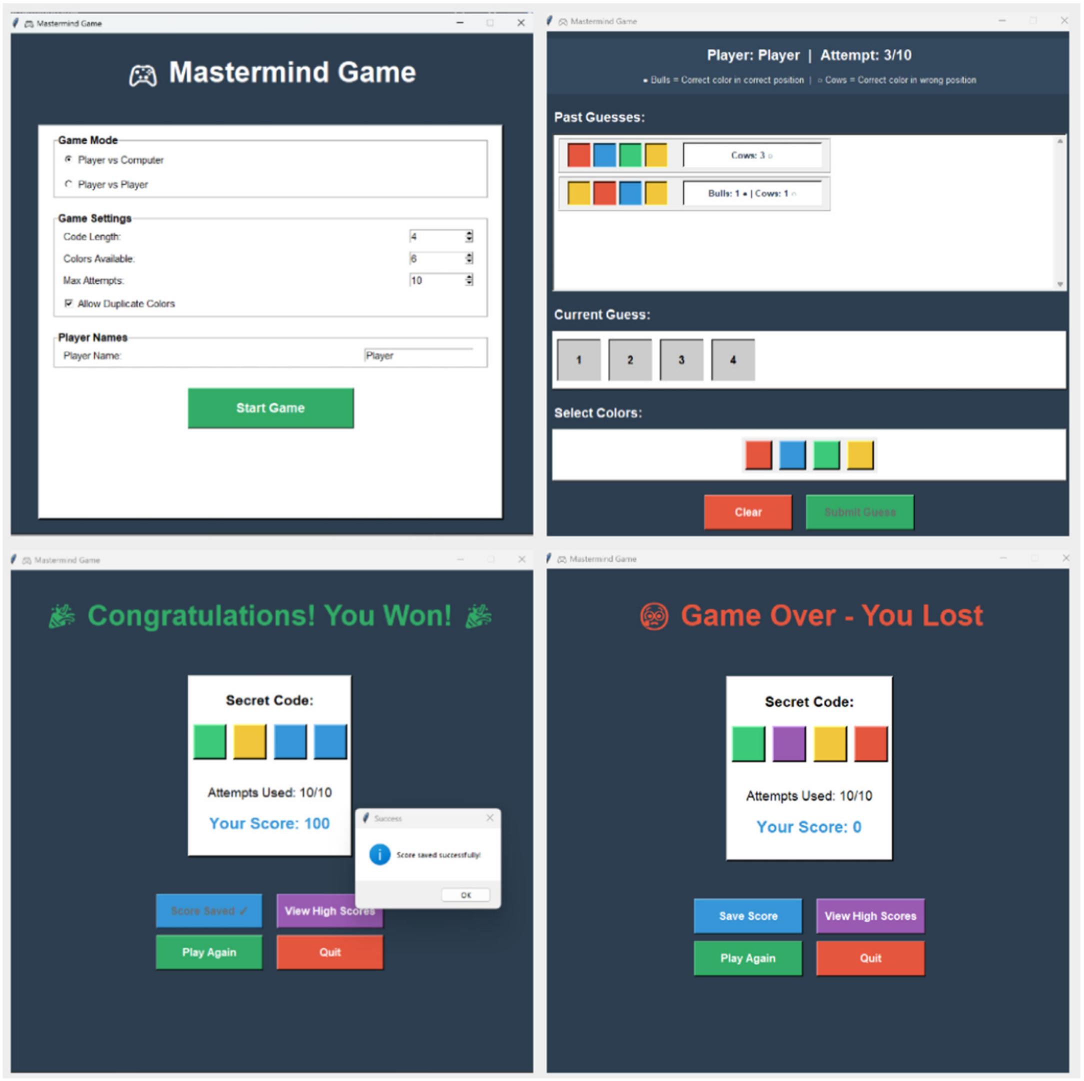
By asking these, we aim to keep our QA process effective and efficient. As an immediate next step, we identified introducing automated GUI tests and possibly integrating a static code quality tool (such as SonarQube) to provide a more in-depth analysis of code metrics and technical debt as areas for improvement.

Quality assurance is a journey, not a destination. This project's testing practices have given us a strong foundation and confidence, and by continuing to adapt and refine these practices, we will maintain high quality in Mastermind and any future projects.

11. Appendix: Artifacts and Templates

This appendix provides supporting materials referenced in the handbook, including example screenshots, code snippets, and test documentation templates that illustrate our testing process and outcomes.

Figure 1. Mastermind Game GUI Screenshot



Mastermind Game

Game Mode
Player vs Computer [✓]
Player vs Player

Game Settings
Code Length: 4
Colors Available: 6
Max Attempts: 10
Allow Duplicate Colors [✓]

Player Names
Player Name: Player

Player: Player | Attempt: 3/10
• Bulls = Correct color in correct position | ○ Cows = Correct color in wrong position

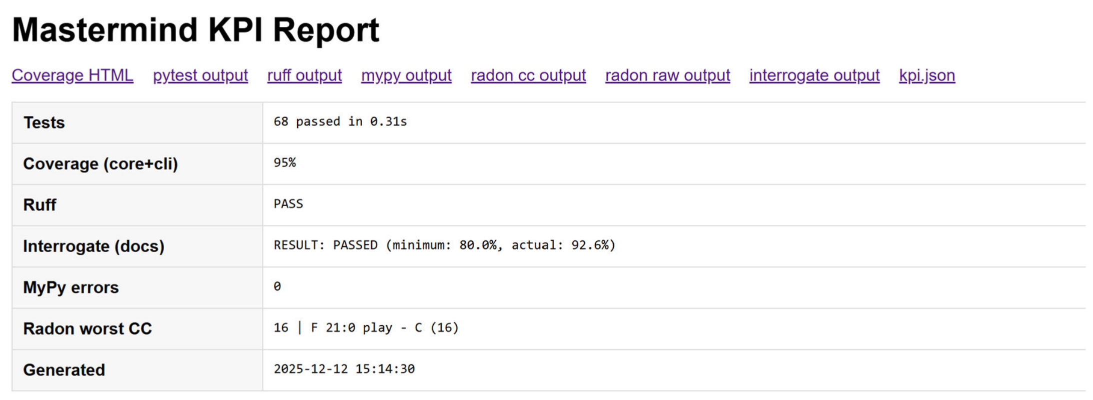
Past Guesses:
🟥🟦🟩🟨 Cows: 3 ○
🟨🟥🟦🟨 Bulls: 1 • | Cows: 1 ○

Congratulations! You Won!
Secret Code: 🟩🟨🟦🟦
Attempts Used: 10/10
Your Score: 100

Success
Score saved successfully!

Game Over - You Lost
Secret Code: 🟩🟪🟨🟥
Attempts Used: 10/10
Your Score: 0

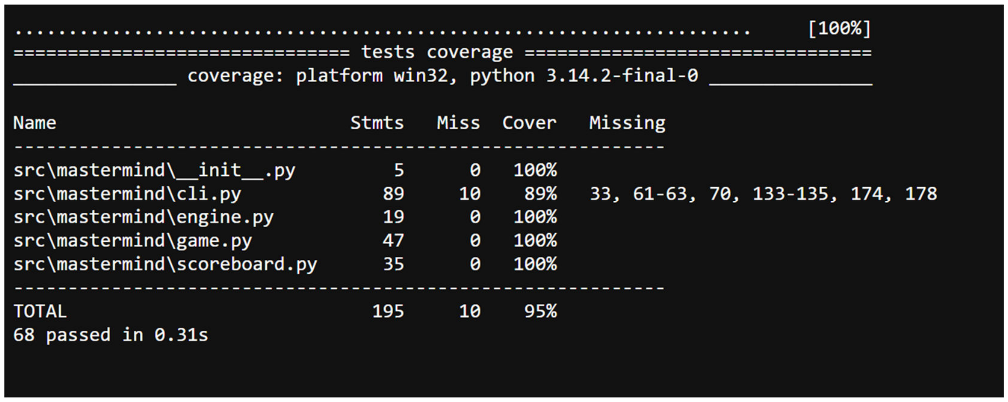
Figure 2. Test Coverage Report Excerpt



Mastermind KPI Report

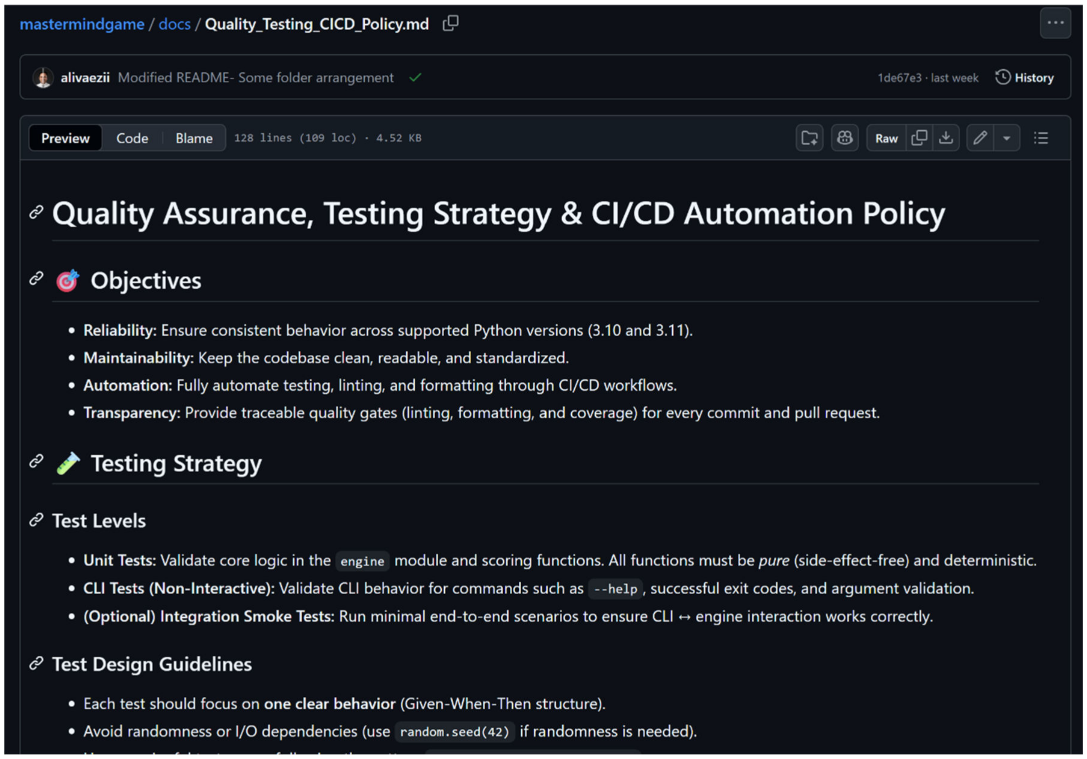
Coverage HTML pytest output ruff output mypy output radon cc output radon raw output interrogate output kpi.json

Tests	`68 passed in 0.31s`
Coverage (core+cli)	`95%`
Ruff	`PASS`
Interrogate (docs)	`RESULT: PASSED (minimum: 80.0%, actual: 92.6%)`
MyPy errors	`0`
Radon worst CC	`16 | F 21:0 play - C (16)`
Generated	`2025-12-12 15:14:30`



```
....................................................................     [100%]
=============================== tests coverage ================================
_______________ coverage: platform win32, python 3.14.2-final-0 _______________

Name                            Stmts   Miss  Cover   Missing
-------------------------------------------------------------
src\mastermind\__init__.py          5      0   100%
src\mastermind\cli.py              89     10    89%   33, 61-63, 70, 133-135, 174, 178
src\mastermind\engine.py           19      0   100%
src\mastermind\game.py             47      0   100%
src\mastermind\scoreboard.py       35      0   100%
-------------------------------------------------------------
TOTAL                             195     10    95%
68 passed in 0.31s
```



mastermindgame / docs / Quality_Testing_CICD_Policy.md

alivaezii Modified README- Some folder arrangement ✓ 1de67e3 · last week History

Preview Code Blame 128 lines (109 loc) · 4.52 KB Raw

Quality Assurance, Testing Strategy & CI/CD Automation Policy

🎯 Objectives

- **Reliability:** Ensure consistent behavior across supported Python versions (3.10 and 3.11).
- **Maintainability:** Keep the codebase clean, readable, and standardized.
- **Automation:** Fully automate testing, linting, and formatting through CI/CD workflows.
- **Transparency:** Provide traceable quality gates (linting, formatting, and coverage) for every commit and pull request.

🧪 Testing Strategy

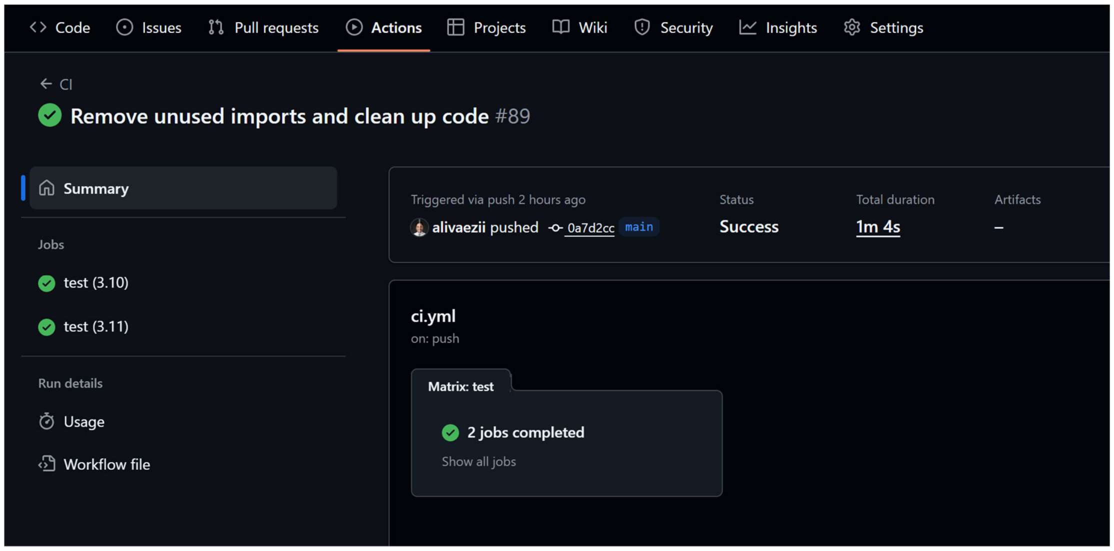
Test Levels

- **Unit Tests:** Validate core logic in the `engine` module and scoring functions. All functions must be *pure* (side-effect-free) and deterministic.
- **CLI Tests (Non-Interactive):** Validate CLI behavior for commands such as `--help`, successful exit codes, and argument validation.
- **(Optional) Integration Smoke Tests:** Run minimal end-to-end scenarios to ensure CLI ↔ engine interaction works correctly.

Test Design Guidelines

- Each test should focus on **one clear behavior** (Given-When-Then structure).
- Avoid randomness or I/O dependencies (use `random.seed(42)` if randomness is needed).

Figure 3. CI Pipeline Test Results (GitHub Actions)



Remove unused imports and clean up code #89

test (3.10)

test (3.11)

Triggered via push 2 hours ago — alivaezii pushed 0a7d2cc main

Status: Success

Total duration: 1m 4s

Artifacts: –

ci.yml
on: push

Matrix: test

2 jobs completed

Sample Test Case Template and Examples

Test Case ID	Title	Description	Steps	Expected Result
TC-ETC-1	Invalid Guess Handling	Verify the game's response to improperly formatted guesses (too short, too long, invalid characters).	1. Launch the game in CLI mode. 2. Enter guess "12" (too short). .3. Enter guess "ABCDEF" (too long, if code length 4). .4. Enter an invalid character "Z999" (if using digits 0-5 for colors).	For each invalid input, the game rejects the guess with an error message **and** does not crash or end the game. The player is re-prompted for a valid guess.
TC-ETC-2	No-Duplicates Rule	Ensure duplicates in a guess are handled when the game is in no-duplicate mode.	1. Start a new game with --no-duplicates setting. 2. If GUI, attempt to select the same color twice for a single guess; If CLI, input a guess like "1123" (duplicate '1'). .3. Observe behavior.	Game prevents the action or flags it: GUI will not allow selecting duplicate colors; CLI will output "Duplicates not allowed" and ignore the guess (not counting it as an attempt).
TC-E2E-1	CLI Full Game Win	Test a complete game flow in CLI, resulting in a win on the first try.	1. Launch CLI with a known secret (seeded random or a debug mode secret "0123"). .2. Input the correct code "0123" as the first guess. 3. Observe game termination.	The CLI should immediately declare victory with a message, display the final score (correctly calculated for nine remaining attempts), and then exit.
TC-E2E-2	GUI Game and Leaderboard	Test playing a game in the GUI and updating the high score.	1. Launch the GUI in PvP mode. 2. Player 1 sets a secret (e.g., Red, Green, Blue, Yellow). .3. Player 2 makes multiple guesses, eventually	The GUI provides feedback after each guess and displays a win message with the score when solved—the High Scores screen updates to include the

			guessing correctly. 4. After a win, go to the High Scores screen.	new score. The scores.json file is updated accordingly.
TC-US-1	GUI Usability - Navigation	Assess first-time user navigation through the GUI.	1. Observe a new user start the game without instructions. 2. Note any confusion or questions the user has. 3. Have the user try to adjust settings or find rules.	The user can start a game and guess without external help. Any points of confusion are minimal and lead to minor improvements (e.g., tooltips). No critical usability blocker is present.
TC-US-2	CLI Usability - Help & Errors	Verify that the CLI provides helpful guidance and error messages.	1. Run mastermind --help. 2. Run mastermind --length -5 (invalid argument). .3. Start a game and input an invalid guess to see the error.	Help text clearly explains usage and options. Invalid arguments trigger error messages like "length must be >=1". An invalid guess during the game yields a friendly error and a reprompt, not a crash.

Quality Gate Criteria (Checklist)

Before merging code or declaring a build release-ready, we verify the following (our “Quality Gate” checklist, as derived from the Test Policy):

- **All unit/integration tests pass** (no failures in test suite).
- **Code coverage ≥ 85%** (no significant drop; review coverage report for any critical untested code).
- **Linting pass** (flake8 reports zero errors/warnings).
- **Static analysis** (no critical security vulnerabilities or code smells reported by tools, if any used).
- **Documentation updated** (any new public methods have docstrings; user documentation reflects new features).
- **DoD acceptance** (Product Owner has accepted the feature as meeting requirements, including any demo or manual acceptance tests).

Only when all boxes are checked can a feature be marked done and merged. Industry standards inspired this checklist and ensure a consistent Definition of Done focused on quality.