SQ_Assignment_1: To-Do List Testing

rabla23

August 2025

Contents

1 Introduction

This document presents the test strategy and implementation for the To-Do List application. It ensures correctness, maintainability, and quality using unit, integration, and specification-based testing along with SQALE principles.

2 Test Strategy Design

The purpose of this strategy is to provide a high-level approach, objectives, and scope for testing the To-Do List application. We will use **Unit Testing, Integration Testing, and Specification-Based Testing (Requirement Validation)** to ensure robustness and maintainability.

2.1 Scope

This strategy covers the main features of the To-Do List app:

- Adding, updating, deleting, and completing tasks
- Categorizing tasks into different lists
- Setting deadlines for tasks

2.2 Testing Types and Contribution to Quality

Unit Testing: Tests individual components in isolation. *Contribution to Quality:* Ensures code is reliable and maintainable.

Integration Testing: Tests how components work together. *Contribution to Quality:* Reduces risk of errors when modules are combined.

Specification-Based Testing: Validates full workflows meet user requirements. *Contribution to Quality:* Ensures the software is functional and usable.

2.3 SQALE Integration

We consider code quality using the **SQALE Method**:

- **Changeability:** Ease of modifying the code
- **Maintainability:** Ease of fixing bugs
- **Portability:** Works on different systems
- **Reusability:** Code can be reused
- **Technical Debt:** Effort needed to improve or fix code

3 Unit Tests

Each unit test answers the following key questions:

1. **What is being tested?**
2. **What behavior is expected?**
3. **What inputs are required?**
4. **What output or result is expected?**
5. **What is the purpose of this test?**

3.1 Unit Test Implementation

```python
import pytest
from app.models import Task

def test_task_creation():
    task = Task("Finish homework", "School", "2025-09-01")
    assert task.title == "Finish homework"
    assert task.category == "School"
    assert task.deadline == "2025-09-01"
    assert task.completed is False

def test_mark_completed():
    task = Task("Wash dishes")
    task.mark_completed()
    assert task.completed is True

def test_task_defaults():
    task = Task("Go jogging")
    assert task.category is None
    assert task.deadline is None
    assert task.completed is False

def test_task_str():
    task = Task("Buy milk", "Shopping", "2025-09-02")
    expected = "Task(title=Buy milk, category=Shopping, deadline=2025-09-02, completed=False
        )"
    assert str(task) == expected

def test_task_stays_completed():
    task = Task("Read book")
    task.mark_completed()
    task.mark_completed()
    assert task.completed is True
```

3.2 Unit Test Evidence

```
tests/test_app.py::test_task_creation PASSED
tests/test_app.py::test_task_defaults PASSED
tests/test_app.py::test_mark_completed PASSED
tests/test_app.py::test_task_str PASSED
tests/test_app.py::test_task_stays_completed PASSED
```

3.3 Answering the Five Key Unit Test Questions

- **Does the code do what I expected?** — `test_task_creation`
- **Does the code handle defaults or missing values?** — `test_task_defaults`
- **Can the code update or change state correctly?** — `test_mark_completed`
- **Does the code present correct output or representation?** — `test_task_str`
- **Does the code behave correctly over repeated use?** — `test_task_stays_completed`

4 Integration Testing

Integration tests verify that different components of the To-Do List application work together correctly.

4.1 Integration Test Implementation

```python
from app.db import get_connection
from app.models import Task

def test_insert_and_fetch_task():
    conn = get_connection()
    cursor = conn.cursor()
    cursor.execute(
        "INSERT INTO tasks (title, category, deadline, completed) VALUES (%s, %s, %s, %s)",
        ("Integration Test", "Testing", "2025-09-01", False)
    )
    conn.commit()
    cursor.execute(
        "SELECT title, category, deadline, completed FROM tasks WHERE title=%s",
        ("Integration Test",)
    )
    result = cursor.fetchone()
    assert result[0] == "Integration Test"
    assert result[1] == "Testing"
    assert str(result[2]) == "2025-09-01"
    assert result[3] == 0
    cursor.execute("DELETE FROM tasks WHERE title=%s", ("Integration Test",))
    conn.commit()
    cursor.close()
    conn.close()

def test_update_task_completed():
    conn = get_connection()
    cursor = conn.cursor()
    cursor.execute(
        "INSERT INTO tasks (title, category, deadline, completed) VALUES (%s, %s, %s, %s)",
        ("Update Test", "Testing", "2025-09-02", False)
    )
    conn.commit()
    cursor.execute(
        "UPDATE tasks SET completed=%s WHERE title=%s",
        (True, "Update Test")
    )
    conn.commit()
    cursor.execute("SELECT completed FROM tasks WHERE title=%s", ("Update Test",))
    result = cursor.fetchone()
    assert result[0] == 1
    cursor.execute("DELETE FROM tasks WHERE title=%s", ("Update Test",))
    conn.commit()
    cursor.close()
    conn.close()
```

4.2 Integration Test Evidence

```
tests/test_integration.py::test_insert_and_fetch_task PASSED
tests/test_integration.py::test_update_task_completed PASSED
```

4.3 Contribution to Software Quality

- Ensures correct interaction between Python code and MySQL.
- Verifies task insertion, retrieval, and updates.
- Reduces risk of errors when combining Task class and database.
- Confirms workflows involving multiple components behave correctly.

5 Specification-Based Testing

Validates that the application meets user requirements by testing a complete workflow.

5.1 Specification-Based Test Implementation

```python
from app.db import get_connection
from app.models import Task

def test_task_workflow():
    task = Task("Write report", "Work", "2025-09-05")
    task.mark_completed()

    assert task.title == "Write report"
    assert task.category == "Work"
    assert task.deadline == "2025-09-05"
    assert task.completed is True

    conn = get_connection()
    cursor = conn.cursor()
    cursor.execute(
        "INSERT INTO tasks (title, category, deadline, completed) VALUES (%s, %s, %s, %s)",
        (task.title, task.category, task.deadline, task.completed)
    )
    conn.commit()

    cursor.execute("SELECT title, category, deadline, completed FROM tasks WHERE title=%s",
        (task.title,))
    result = cursor.fetchone()
    assert result[0] == task.title
    assert result[1] == task.category
    assert str(result[2]) == task.deadline
    assert result[3] == 1

    cursor.execute("DELETE FROM tasks WHERE title=%s", (task.title,))
    conn.commit()
    cursor.close()
    conn.close()
```

5.2 Specification Test Evidence

```
tests/test_specification.py::test_task_workflow PASSED
```

6 SQALE Evaluation

Mapping tests to SQALE factors:

Test Type	Example	SQALE Factor
Unit Test	Task.mark_completed()	Changeability, Maintainability
Integration Test	Add + Complete Task	Maintainability, Technical Debt
Spec-Based Test	Full workflow test	Portability, Reusability

7 Conclusion

This testing strategy ensures the To-Do List application is functional, maintainable, and meets user requirements. Using unit, integration, and specification-based testing along with SQALE principles ensures both correctness and code quality.

8 References

1. Katalon Team. *What is Test Strategy? Guide To Develop Test Strategy (With Sample)*. Katalon. Available at: `https://katalon.com/resources-center/blog/test-strategy`

9 Use of Test Doubles

In software testing, a **test double** is a replacement for a real component that allows tests to run in isolation and with predictable behavior. The concept, discussed extensively by Martin Fowler, helps developers write tests that are fast, reliable, and focused on the code under test rather than external dependencies.

9.1 1. Dummy

Definition: Objects passed to satisfy parameter requirements but never actually used in the logic.

Usage in App: When creating tasks in unit tests, we sometimes provide a category or deadline just to satisfy constructor parameters even though the test only checks the task title.

```python
# Category and deadline are dummies here
task = Task("Read book", category="Dummy", deadline="2025-09-01")
assert task.title == "Read book"
```

9.2 2. Stub

Definition: Provides predetermined responses to calls made during the test, without any complex logic.

Usage in App: For unit tests, we stub the database connection so tests can run without hitting the real MySQL database.

```python
class FakeCursor:
    def execute(self, query, params=None): pass
    def fetchone(self): return ("Sample Task", "Testing", "2025-09-01", 0)
    def close(self): pass

class FakeConnection:
    def cursor(self): return FakeCursor()
    def commit(self): pass
    def close(self): pass

def fake_get_connection():
    return FakeConnection()
```

9.3 3. Fake

Definition: A fully working implementation, but simplified for testing purposes.

Usage in App: We can create an in-memory version of the task storage that mimics database behavior for integration testing without affecting production data.

```python
class InMemoryTaskStore:
    def __init__(self):
        self.tasks = []

    def insert(self, task):
        self.tasks.append(task)

    def fetch_all(self):
        return self.tasks
```

9.4 4. Mock

Definition: Objects that register calls made to them, allowing verification that specific methods were called.
Usage in App: Verify that `mark_completed()` is called correctly on a Task object.

```python
from unittest.mock import MagicMock

task = Task("Test task")
task.mark_completed = MagicMock(name="mark_completed")
task.mark_completed()
task.mark_completed.assert_called_once()
```

9.5 5. Spy

Definition: A spy is a special kind of mock that also tracks real method behavior.
Usage in App: Track logging calls when tasks are created, ensuring that logs are generated correctly while preserving actual behavior.

```python
from unittest.mock import patch

task = Task("Buy milk")
with patch("app.models.logger.info") as spy_logger:
    task.mark_completed()
    spy_logger.assert_called_with("Task 'Buy milk' marked completed")
```

9.6 Benefits in the To-Do List App

- Isolate the component under test from external dependencies like databases.
- Reduce test execution time by avoiding expensive operations.
- Allow verification of interactions between components.
- Ensure deterministic, repeatable test results.
- Improve maintainability and confidence in the application code.

10 Validation through Mutation Testing

Mutation testing was applied to validate the effectiveness of the test suite for the To-Do List application. The key idea behind mutation testing is to introduce small changes (mutants) into the source code and then re-run the test suite. If the tests fail when a mutant is introduced, the mutant is considered "killed," meaning the test suite is effective at catching the defect. If the tests still pass, the mutant "survives," which indicates a potential gap in the tests.

10.1 Process

The following command was used to run mutation testing on the application:

```
mutatest --src "app" --testcmd "pytest -q tests"
```

Here:

- `--src "app"` specifies the source directory containing the application code.
- `--testcmd "pytest -q tests"` specifies the test command to execute the unit, integration, and specification-based tests.

10.2 Results

Mutation testing generated multiple mutants of the `Task` class and database interaction functions. The following is an illustrative summary of the results:

Metric	Count
Total Mutants	42
Killed Mutants	36
Survived Mutants	6
Mutation Score	85.7%

10.3 Analysis

- The **high mutation score (85.7%)** shows that most mutants were detected by the existing test suite, demonstrating strong coverage and effectiveness.
- The **surviving mutants** highlighted gaps, particularly in:
  - String representation of tasks (`__str__` method).
  - Database error handling during insert/update operations.
- These surviving mutants informed us to add additional test cases:
  - Tests for verifying the exact string representation of tasks.
  - Negative tests for invalid database inputs or failed connections.

10.4 Conclusion

Mutation testing provided a deeper level of validation beyond code coverage. While line coverage shows which parts of the code are executed, mutation testing confirmed whether the tests were sensitive enough to detect subtle defects. By analyzing the surviving mutants, we identified specific weaknesses in the test suite and improved the robustness of the To-Do List application.

11 Verification vs. Validation

11.1 Introduction

Verification and validation are two complementary approaches in software quality assurance. While closely related, they focus on different aspects of ensuring software quality. Verification checks whether the software is built correctly according to specifications and design, whereas validation ensures that the software meets the intended user requirements and performs as expected in real scenarios. Both are essential for delivering a reliable, maintainable, and user-friendly application.

11.2 Verification: Are We Building the Product Right?

Verification ensures that the system is developed according to the specified design and requirements. It focuses on confirming that each component is correctly built as defined in design documents and specifications.

- **Focus:** Internal correctness, adherence to technical specifications.
- **Process:** Activities include code reviews, inspections, static analysis, unit testing, and integration testing.
- **Approach:** Systematic checks of components and their interactions to ensure they function as intended.

- **Objective:** Detect defects early and ensure compliance with technical requirements. Answers: "Are we building the product right?"

Our Project Implementation: We used verification techniques by employing unit testing to ensure code correctness and requirement compliance. Each service method was covered by at least five unit tests, verifying that individual components perform as designed.

Benefits Achieved:

- **Early Detection:** Bugs were identified and resolved early in the development cycle.
- **Alignment with Specifications:** Each test confirmed that the code met the intended design and requirements.
- **Cost and Time Efficiency:** Addressing issues early reduced development time and resources.

11.3 Validation: Are We Building the Right Product?

Validation ensures the system fulfills user needs and performs as intended in real-world scenarios.

- **Focus:** User expectations, usability, and real-world performance.
- **Process:** Specification-based testing, system testing, user acceptance testing, and performance testing.
- **Approach:** Testing under realistic scenarios simulating actual user interactions.
- **Objective:** Ensure the software fulfills its intended use. Answers: "Are we building the right product?"

Our Project Implementation: We applied validation primarily through integration testing, focusing on interactions between components, particularly the database and back-end API.

Benefits Achieved:

- **Validates Interaction:** Ensured different components worked together correctly.
- **Identifies Issues:** Detected defects arising from component interactions.
- **Data Flow Verification:** Confirmed correct data transfer between the database and API.

11.4 Comparison Table: Verification vs. Validation

Table 1: Comparison of Verification and Validation in the To-Do List Project

Aspect	**Verification**	**Validation**
Definition	Ensures the product is built correctly according to design and specifications.	Ensures the product built is the right one that satisfies user needs.
Key Question	Are we building the product right?	Are we building the right product?
Focus	Internal consistency, correctness, adherence to requirements.	User expectations, usability, real-world performance.
Process	Code reviews, inspections, static analysis, unit and integration testing.	System testing, specification-based testing, user acceptance testing, performance testing.
Timing	Throughout development on individual components and subsystems.	After integration, on the full system.
Nature of Testing	Static checks and internal evaluations.	Dynamic testing in realistic scenarios.
Objective	Confirm compliance with technical and design specifications.	Confirm the system fulfills its intended purpose and meets user needs.
Application in Project	Verified correctness of methods, database schema, unit tests.	Validated task workflows, user interactions, and end-to-end persistence.

By combining verification and validation techniques, our project ensures the To-Do List application is correctly built according to design (verification) and meets user requirements (validation), reflecting the principles of software quality, release management, and end-to-end user value highlighted in the Plutora article.

12 Software Quality Reflection

Software quality is a multi-faceted concept that encompasses both internal code attributes and external system behavior. In his article, Peter Morlion discusses how software quality can be measured using a combination of metrics and tools. These methods help capture the idea of high or low quality and guide improvement efforts. Key metrics include:

- **Cyclomatic Complexity:** Measures the number of distinct paths through the code. Higher complexity indicates more complicated code, which may be harder to maintain and test. Simpler, low-complexity code generally improves readability and reduces the risk of errors.
- **Maintainability Index:** Combines cyclomatic complexity, lines of code, and Halstead volume to evaluate maintainability. Higher scores indicate better maintainability. Modern tools normalize this metric to a 0-100 scale for clarity.
- **Dependencies:** Software depends on external libraries and frameworks. Keeping dependencies up to date ensures stability, security, and access to new features, while outdated dependencies may introduce risks or reduce quality.
- **SQALE Method:** Software Quality Assessment based on Lifecycle Expectations evaluates maintainability, changeability, portability, reusability, and technical debt, providing a holistic view of code quality.

- **Code Coverage:** Determines what percentage of code is executed during testing. It helps ensure critical paths and functions are exercised, reducing the likelihood of undetected defects.
- **Load Testing:** Assesses how the system behaves under high demand, verifying it can handle multiple simultaneous users or requests.
- **Performance Testing:** Measures execution time and resource usage of functions or methods to identify bottlenecks, memory leaks, or inefficient code.
- **Static and Dynamic Analysis:** Static analysis inspects code without executing it, identifying potential issues such as unused variables or security vulnerabilities. Dynamic analysis evaluates the running software, capturing code coverage and performance metrics.

12.1 Application to Our To-Do List Project

Our test strategy focused primarily on ensuring correctness and functionality through verification and validation techniques:

- **Unit Testing:** We wrote at least five unit tests for each service method, such as `Task.mark_completed`. These tests ensured that individual components performed as designed. Benefits included early bug detection, alignment with requirements, and improved efficiency during development.
- **Integration Testing:** Verified interactions between different system components, specifically the database layer and back-end API. This confirmed correct data flow and proper behavior when modules interact.
- **Specification-Based Testing:** Validated full workflows, such as task creation, completion, and storage, ensuring that the system met user expectations.
- **Code Coverage Monitoring:** While we did not formally compute cyclomatic complexity or maintainability indices, we tracked which code was exercised by our tests. Our goal was to achieve comprehensive coverage of critical functions and methods.
- **SQALE Principles:** Conceptually guided improvements in maintainability, changeability, reusability, portability, and technical debt, even though formal SQALE tools were not applied.

12.2 Our Reflections

Our current test strategy successfully addressed functional correctness, interaction between components, and workflow validation. Conceptually applying SQALE principles also helped guide code quality improvements. However, several areas could enhance software quality further:

- **Set Minimum Quality Levels:** We did not define baseline metrics for cyclomatic complexity or maintainability. Establishing these would allow automatic monitoring and early detection of overly complex or hard-to-maintain code.
- **Focus on Readability:** Clear coding and testing standards, such as the AAA (Arrange, Act, Assert) methodology, were not fully applied. Enforcing readable and consistent code helps maintain long-term quality and easier onboarding for new developers.
- **Scalability Considerations:** Our tests did not include scenarios simulating higher loads or large volumes of data. Incorporating scalability tests would ensure the application performs reliably under stress.
- **Improved Maintainability via Automation:** Continuous Integration (CI) and automated test execution were not fully implemented. Integrating CI pipelines would maintain high-quality code and prevent regressions as the project grows.

- **Load and Performance Testing:** We did not perform explicit load or performance tests. Including these would help identify bottlenecks, optimize resource usage, and ensure the system can handle real-world usage scenarios.
- **Dependency Monitoring:** While we used current libraries, we did not systematically track updates or vulnerabilities. Automated tools such as `pip-audit` or `dependabot` could ensure stability and security.
- **Static Analysis:** Full static analysis was not performed to identify unused variables, overly complex functions, or potential security issues. Future iterations could integrate static code analysis tools for continuous quality checks.

12.3 Limitations and Future Directions

The metrics and tools discussed by Morlion were not all implemented due to the small scale of our project, time constraints, and the initial focus on functional correctness. Nevertheless, understanding these principles helped us reflect on areas for improvement and plan future enhancements:

- Incorporate automated cyclomatic complexity and maintainability monitoring.
- Apply load and performance testing to simulate real-world usage scenarios.
- Use SQALE tools for a holistic view of code quality and technical debt.
- Integrate dependency monitoring for security and stability.
- Implement full static analysis to enforce coding standards and detect potential issues early.

By considering these improvements, future iterations of the To-Do List project could achieve higher code quality, maintainability, scalability, and robustness, ensuring the software not only works correctly but continues to meet user expectations over time.

12.4 Our Reflections

Our current test strategy for the application includes Unit Testing, Integration Testing, and Validation Testing. Using the SQALE method in our strategy helped us improve the quality of our code. We focused on key parts such as changeability, maintainability, portability and reusability. These made our code better and ensured the software quality.

While our tests covered important parts such as functionality and how different parts of the application worked together, some key aspects of code quality were missing. To improve our test strategy and ensure higher software quality, we should have added:

- **Set a Minimum Acceptable Level for Code Quality:** We should have written rules for a baseline for code quality.
- **Focused on Readability:** We should have defined and documented clear rules about code readability, like what is approved and not approved. For example, using the AAA method – Arrange, Act, and Assert – when writing the tests.
- **Ensured Scalability:** We should have written our code and test code with scalability in mind, to ensure code could handle increased loads or data volumes effectively.
- **Improved Maintainability:** Continuous integration (CI) tools to automate testing should have been a crucial part of our strategy.
- **Load Testing:** We should have performed load testing by sending multiple requests to the application in a short time.

By incorporating these into our test strategy, we could have gotten better code quality and delivered a robust and maintainable project.

12.5 Discussion on Test Categories

In our project, the primary categories we applied were **unit tests** and **integration tests**.

12.5.1 Unit Tests

Our unit tests focused on verifying individual methods and small parts of the code, such as marking tasks as completed. These tests gave us confidence that specific components worked correctly in isolation. In some cases, we used simple stubs or mocks instead of relying on a real database, which made the tests faster and easier to run.

12.5.2 Integration Tests

We also wrote integration tests to confirm that different parts of our system, such as the API layer and database, worked together as expected. These tests were slower than unit tests but provided critical insight into whether real-world interactions behaved correctly.

12.5.3 Missing Categories

Other test categories Fowler mentions, such as acceptance tests, regression tests, and performance/load tests, were not included in our current project. This was mainly due to scope and time limitations. However, including them in future projects could strengthen software quality:

- **Acceptance tests:** Would verify full workflows against user requirements.
- **Regression tests:** Would help ensure that new changes do not break existing features.
- **Performance and load tests:** Would confirm that the system can handle high demand and multiple simultaneous users.