South China University of Technology

《High-level Language Programming Project》 Report

Ztest: A C++ Unit Testing Framework

School :	School of Future Technology
Major :	Data Science and Big Data Technology
Student Name :	Zheng Chenyang,Ye Suohua,Wu Hongqing, Qi Yansong,Wang Ruizhen
Teacher :	Zhang Huaidong
Submission Date:	2025.6.6

目录

1 Abstract

Ztest is an innovative C++ unit testing framework designed to address limitations in existing tools like Google Test, such as lack of GUI support, poor concurrency handling, and inadequate reporting systems. It introduces a layered architecture combining object-oriented design, functional programming, and metaprogramming, with core modules including a test executor, data-driven testing system, and AI-powered diagnostics. Key features include:

1. **Parallel Execution**: Safe tests are run concurrently via a thread pool (8-thread default), reducing execution time compared to sequential frameworks.
2. **Data-Driven Testing**: Implements parameterized tests with CSV integration and LRU caching, accelerating large-scale test execution.
3. **Smart Diagnostics**: Integrates Qwen Turbo API for failure root cause analysis, test coverage assessment, and stability recommendations in HTML reports.
4. **Cross-Platform GUI**: Built with ImGui and MVC architecture, supporting real-time resource monitoring, test filtering, and AI-assisted debugging.
5. **Extensible Reports**: Generates HTML/XML/JSON reports compatible with CI/CD pipelines, including detailed benchmark metrics and visualizations.

2 System Requirements Analysis

2.1 System Background and Motivation

With the increasing demand for complex system architecture design capabilities, mastering object-oriented design principles and pattern-based engineering practices has become a core goal of advanced software engineering education. This not only requires an understanding of the dynamic collaboration between classes and objects, but also the ability to solve architectural challenges through design patterns with abstract thinking skills. Existing unit testing frameworks (such as Google Test) have several drawbacks, including a steep learning curve, limited concurrency support, overly simplistic reporting systems, and average extensibility. Our team plans to develop a flexible, efficient, and easy-to-use testing tool (with a graphical user interface, GUI) to provide an intuitive and user-friendly environment for developers and testers to write, run, and manage test cases. This tool will support various types of tests (such as unit tests and integration tests) and provide detailed test result reports.

表 1: Comparison of Mainstream Testing Frameworks

Framework	**GUI Support**	**Concurrent Testing**	**Report System**	**Data-Driven**
Google Test (C++)	Third-Party	Limited	Basic	Not Supported
JUnit (Java)	Eclipse Plugin	Supported	HTML/XML	Supported
PyTest (Python)	Third-Party	Excellent	Rich	Supported
Catch2 (C++)	None	Average	Concise	Not Supported
Ztest* (C++)	Excellent	Excellent	Rich	Supported

2.2 System Objectives

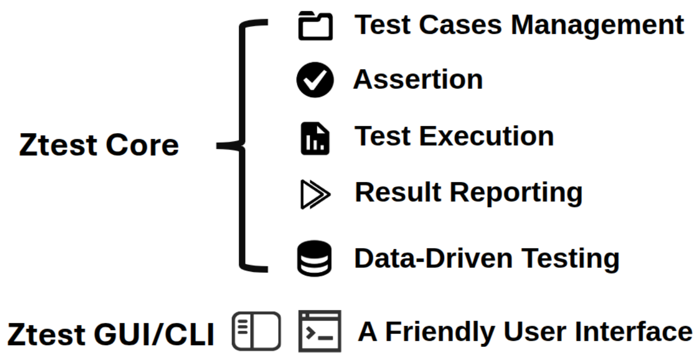


图 1: Ztest Function

2.2.1 Test Management

Compared to traditional unit testing frameworks, Ztest supports various test types, such as parallel and thread-safe tests, serial or thread-unsafe tests, performance evaluation through iteration, and parameterized data-driven tests. We provide test fixtures to manage individual tests.

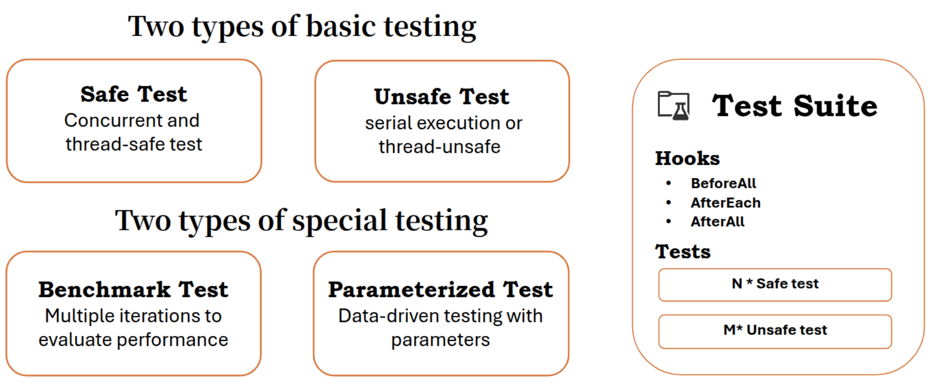

Two types of basic testing
Safe Test
Concurrent and thread-safe test
Unsafe Test
serial execution or thread-unsafe
Two types of special testing
Benchmark Test
Multiple iterations to evaluate performance
Parameterized Test
Data-driven testing with parameters
Test Suite
Hooks
BeforeAll
AfterEach
AfterAll
Tests
N * Safe test
M* Unsafe test

图 2: Test Types

We provide macros similar to Google Test to simplify test definition:

```
ZTEST_F(SuiteName, TestName, safe/unsafe) { ... } // Define a test
ZBENCHMARK(SuiteName, TestName, iterations) { ... } // Define a benchmark
ZTEST_P(SuiteName, TestName, data) { ... } // Define a parameterized test
ZTEST_P_CSV(SuiteName, TestName, "data.csv") { ... } // Define a parameterized
    test with csv data
```

At the same time, we can chain test case definitions:

```
auto test_case = TestFactory::createTest("Add", ZType::Z_SAFE, "", add, 2, 3)
               .setExpectedOutput(5)
               .beforeAll([]() { logger.info("Init\n"); })
               .afterEach([]() { logger.info("Clean\n"); }).build();
```

2.2.2 Assertion Mechanism

The assertion mechanism is used to verify whether the expected results of test cases are correct. The framework provides a variety of assertion macros, such as **EXPECT_EQ** and **ASSERT_TRUE**, to quickly verify conditions in tests. The main assertions provided are:

- **EXPECT_EQ**: Verifies whether two values are equal.
- **ASSERT_TRUE**: Verifies whether a condition is true.
- **EXPECT_NEAR**: Verifies whether two floating-point values are close enough.

Usage example:

```
// If the assertion fails, an exception is thrown
EXPECT_EQ(5, add(2, 3));
ASSERT_TRUE(6 == add(2, 3));
EXPECT_NEAR(5.0, add(2.0, 3.0), 0.1);
```

If an assertion fails, a **ZTestFailureException** exception is thrown. You can also inherit from the **ZTestFailureException** to customize the exception handling logic.

2.2.3 Data-Driven Testing

Parameterized testing refers to replacing certain fixed data in test cases with parameters during the testing process, and then generating multiple sets of test data by changing the values of the parameters to perform multiple tests on the software. It allows testers to cover multiple input scenarios with a single set of test logic, instead of writing separate test code for each scenario. A data-driven testing framework (Data-Driven Testing Framework) is an automated testing framework that separates test data from test scripts and drives test case execution through external data sources (such as Excel, CSV files, databases, etc.).

ZTest implements parameterized testing and data-driven testing by importing and parsing data from CSV files. To accelerate data access, the imported data is cached. A lazy loading mechanism is used, where data is only loaded into the cache when needed, reducing unnecessary resource consumption. At the same time, an LRU cache eviction strategy is used to ensure that the data in the cache is the most likely to be accessed again, thereby maximizing the effectiveness of the cache.

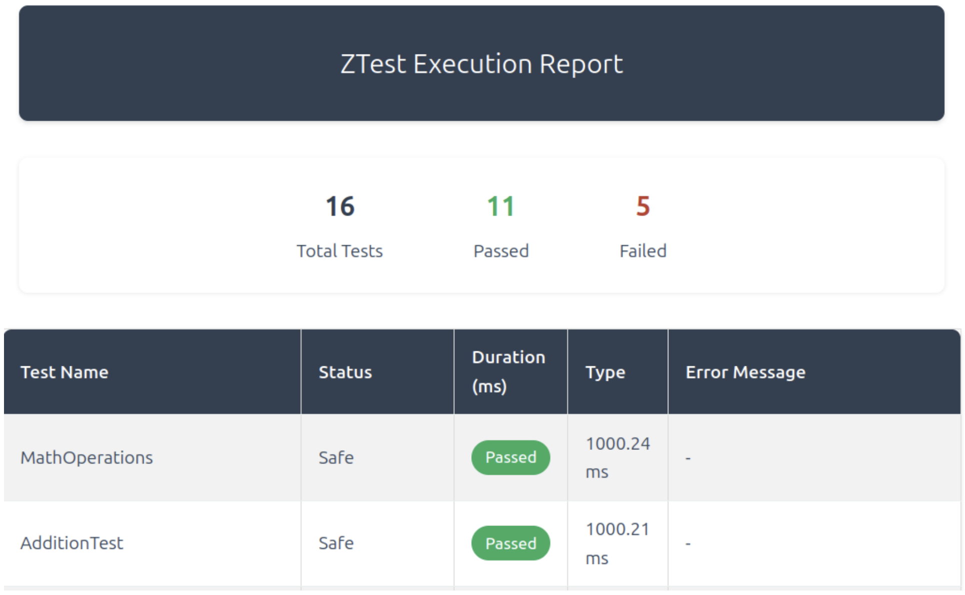
2.2.4 Test Executor

The test executor is responsible for managing the execution of test cases, supporting **concurrent** execution of test cases and collecting test results, implementing automatic scheduling of tests and test result statistics. Its behavior is as follows:

- Concurrently run safe tests. Thread creation and destruction are relatively time-consuming operations. Using a thread pool to manage the lifecycle of threads allows for thread reuse, avoiding frequent thread creation and destruction, thereby significantly improving system performance.
- Serially run unsafe tests. A queue is used to maintain the order of test cases to ensure sequential execution.
- Serially run Benchmark tests.
- Serially run Parameterized tests.

2.2.5 Report Generation

It can generate HTML, JSON, and XML reports for convenient viewing of test results and integration with CI/CD.


ZTest Execution Report

16 Total Tests

11 Passed

5 Failed

Test Name	Status	Duration (ms)	Type	Error Message
MathOperations	Safe	Passed	1000.24 ms	-
AdditionTest	Safe	Passed	1000.21 ms	-

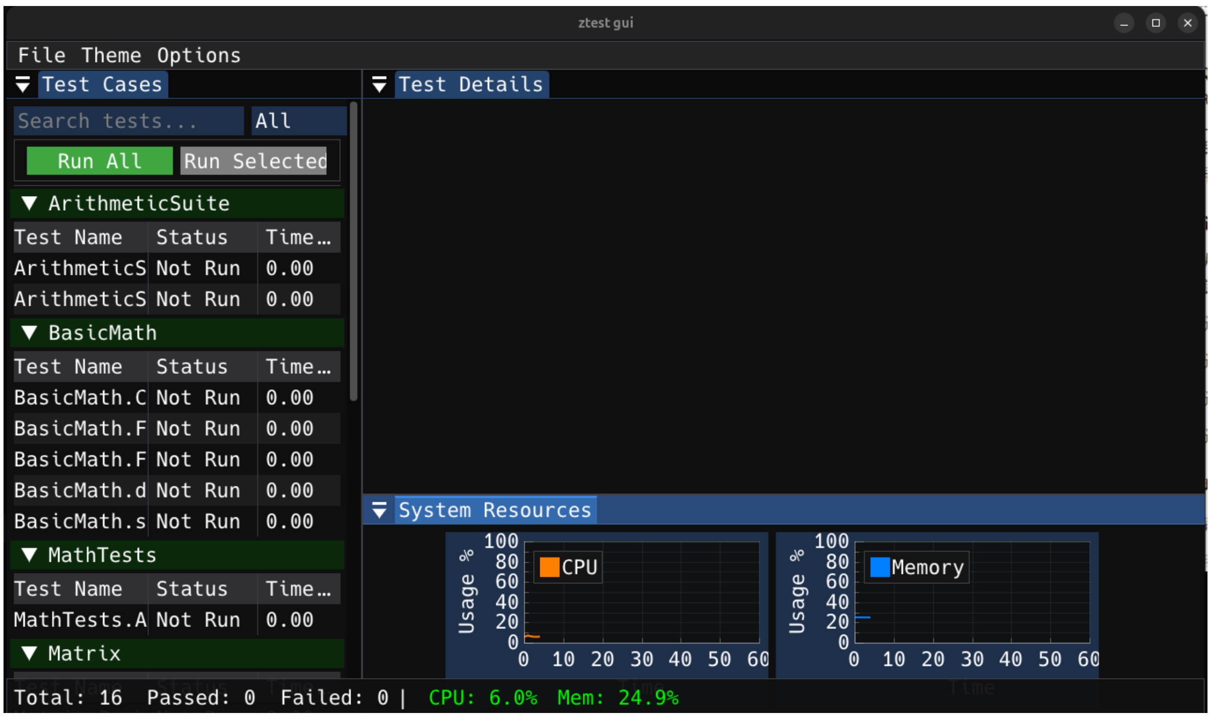
图 3: Test Result Display Interface Layout

2.2.6 CLI Interface

```
Usage: executor_name [OPTIONS]
Options:
--help                      Show help
--run-all                   Run all tests
--list-tests                List all tests
--no-gui                    Run in headless mode
```

2.2.7 GUI Display

The GUI can display test results, filter test results, monitor system resource status, and view details of a specific test run.



▼ ArithmeticSuite

Test Name	Status	Time…
ArithmeticS	Not Run	0.00
ArithmeticS	Not Run	0.00

▼ BasicMath

Test Name	Status	Time…
BasicMath.C	Not Run	0.00
BasicMath.F	Not Run	0.00
BasicMath.F	Not Run	0.00
BasicMath.d	Not Run	0.00
BasicMath.s	Not Run	0.00

▼ MathTests

Test Name	Status	Time…
MathTests.A	Not Run	0.00

▼ Matrix

```
Total: 16  Passed: 0  Failed: 0 |  CPU: 6.0%  Mem: 24.9%
```

图 4: Test Management Interface Layout

2.2.8 AI Intelligent Diagnosis Function

By calling the qwen3 interface, an intelligent diagnosis function is realized. The report can include:

1. Identifying the root cause of failures
2. Providing repair suggestions
3. Identifying high-risk test cases
4. Assessing overall test coverage

5. Offering suggestions for improving system stability

You can also obtain specific analysis for a particular test case on the GUI interface, customize prompts, and upload source files to assist with analysis.

3 Program Analysis

3.1 Key System Issues

3.1.1 Excellent Architectural Design

Ztest uses a layered architecture for the core modules, specifically as follows:

- Basic Layer: Provides basic functions of the test framework, such as test case registration, test execution, test result management, and log output.
- Execution Layer: Manages the execution of test cases, including registration, execution, result management, and log output.
- Data Layer: Implements data-driven parameterized testing and provides data sources for test cases.
- Report Layer: Offers multi-format report output, such as HTML, JSON, XML, etc.
- Concurrency Layer: Encapsulates thread pool management for concurrent execution, improving testing efficiency.

3.1.2 Application of Programming Design Ideas

In the implementation process, Ztest uses five programming paradigms: object-oriented programming, generic programming, functional programming, concurrent programming, and metaprogramming.

3.1.3 Improving Test Execution Efficiency

Traditional C++ testing frameworks (such as gtest, catch2) mostly use sequential test execution methods, resulting in longer test times and reduced production efficiency. Meanwhile, the emergence of vibe coding has significantly reduced the time required to write test cases, but the time required to run test cases has not been significantly reduced. We categorize tasks into four types: short test time and more focused on result correctness, such as addition operations, string concatenation, and user login verification; long test time and more focused on result correctness, such as merging multiple files, complex string matching and replacement, and large data sorting result verification; short test time and more focused on process evaluation, such as reading or writing large files, low-complexity algorithm performance testing, and database queries; and long test time and more focused on process evaluation, such as multi-threaded processing tasks, stress testing, and high-time-complexity algorithm testing.

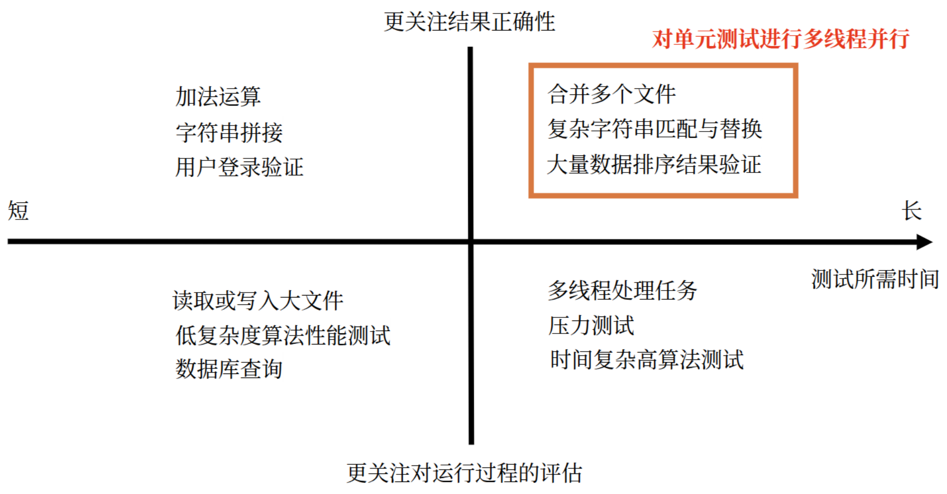


图 5: Task Types

For tasks that take a long time to test, are more focused on result correctness, and are thread-safe, ZTest introduces concurrent test execution into the C++ testing framework system, directly integrating it into the test framework instead of using third-party tools like google-parallel, achieving finer-grained control. At the same time, introducing data-driven testing and using caching to accelerate the definition and execution speed of large-scale unit tests is of great significance.

3.1.4 Implementation of Syntactic Sugar

How to encourage developers to use unit testing frameworks? The key lies in the simplicity of syntax definition. We use a series of complex macro definitions to implement syntactic sugar, thereby simplifying user definitions.

3.1.5 Automatic Type Inference for Functions to Be Tested

When defining a single test case in a chained manner, we hope that users only need to provide the function name, parameters, and expected results, and our framework will take over the specific logic of running and comparing test results. This requires us to automatically infer the parameter types of the function to be tested so that the function can be called for testing and test result verification. We use the factory pattern to specify the return type and the builder pattern to construct parameters and set expected results.

3.2 Responsibilities Allocation

- Zheng Chenyang: Architectural design, main code writing for ztest core and GUI, report writing, and presentation
- Ye Suohua: GUI improvement, partial test logic improvement
- Wu Hongqing: GUI improvement
- Qi Yansong: GUI improvement
- Wang Ruizhen: Attempt to migrate to Windows

4 Technical Approach

4.1 Operating Environment

表 2: Development and Operating Environment

Component	Tool Used for Development
Processor	Intel i9-14900HX (32) @ 5.800GHz
Operating System	Ubuntu 24.04.2 LTS x86_64 (kernel 6.11.0-26-generic)
Compiler	gcc 13.3.0 or clang 18.1.3 (MSVC is not allowed)
Graphics API	glfw 3.4 + glad 4.0.1
GUI Framework	ImGui-1.91.7-docking
Data Visualization Tool	implot v0.16
Build System	XMake v2.9.9+HEAD.40815a0
C++ Standard	C++20 (required)

4.2 Overall Design

4.2.1 Ztest Core Architecture Diagram

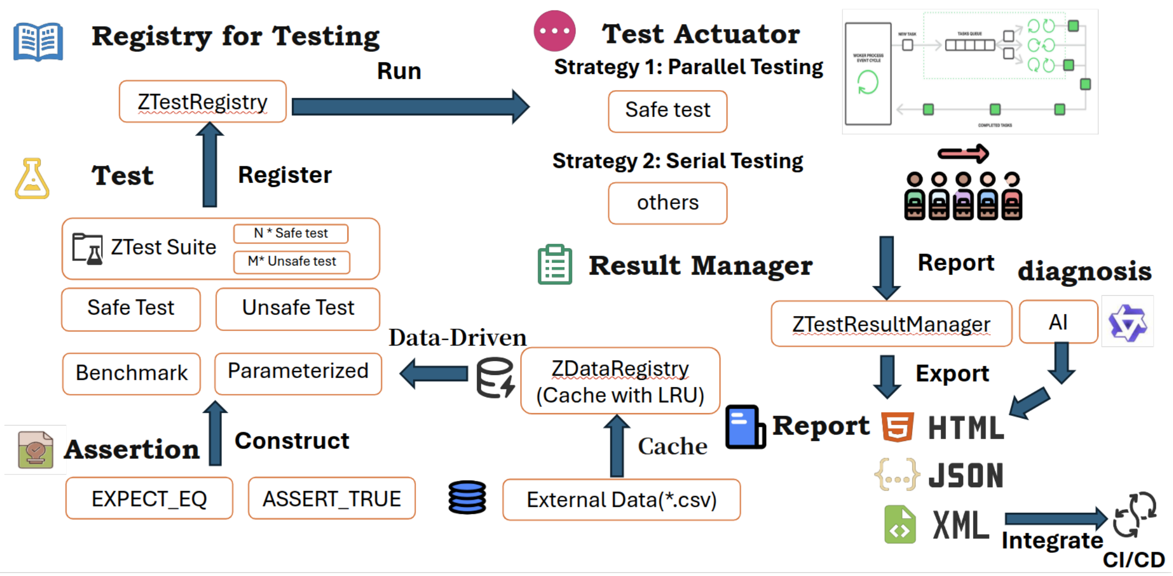


图 6: Ztest Design

The core architecture process of ZTest begins with the test definition stage, where tests (including ZTest Suite, safe tests, unsafe tests, benchmark tests, and parameterized tests) are defined using assertions such as EXPECT_EQ and ASSERT_TRUE to verify the correctness of test results. Next, the tests are registered with the ZTestRegistry for unified management. After registration, the tests are sent to the test executor, which adopts different test strategies based on the type of test. For safe tests, a parallel testing strategy is used, while other types of tests are executed serially. During the execution of data-driven tests, the data-driven module manages external data (such as

CSV files) through ZDataRegistry (with an LRU mechanism cache) to support test execution. After testing is completed, the result manager ZTestResultManager is responsible for collecting and processing test results and exporting them in HTML, JSON, or XML formats for reporting and further analysis. When exporting in HTML format, AI (qwen turbo) is used for test diagnosis and integrated into the HTML test report. In addition, the entire testing process supports integration with Continuous Integration/Continuous Deployment (CI/CD) systems, allowing test results to be automatically fed back into the development process to improve software development efficiency and quality.

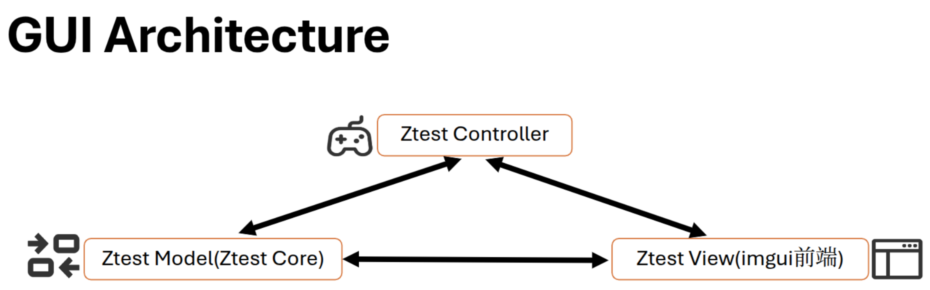


图 7: Ztest GUI Architecture

4.2.2 GUI Framework

The GUI framework is developed using the MVC architecture, where the Model layer is responsible for data processing (further encapsulating Ztest core), the View layer is responsible for interface drawing (mainly using the ImGui framework), and the Controller layer is responsible for user interaction, converting user operations on the UI interface (such as clicks and filtering) into calls to the Model and View layers to update the user interface and data.

4.2.3 Overall Class Diagram

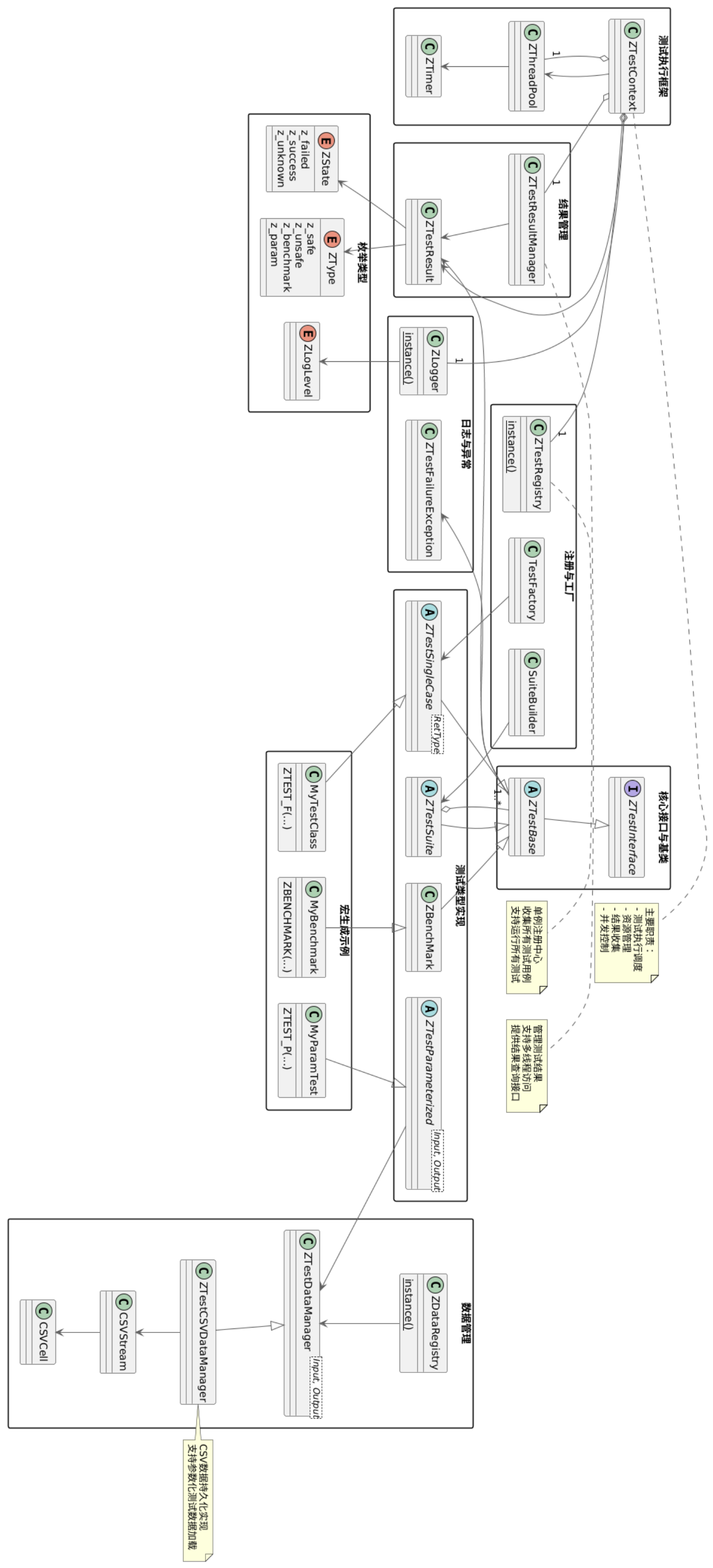


图 8: Ztest Class Diagram

4.3 Detailed Design

4.3.1 Test Definition Related Classes

- `ZTestInterface`: The test interface defines the test execution framework.
- `ZTestBase` (ztest_base.hpp): An abstract test base class that encapsulates common attributes and lifecycle hooks. It uses the Template Method pattern to define the test execution framework, implements the Strategy pattern through a virtual function table, and extends using hook functions with the Decorator pattern.
- `ZTestSingleCase` (ztest_singlecase.hpp): A single test case implementation that uses the Template Method pattern to execute test logic, cooperates with TestFactory to demonstrate the Factory pattern, and supports chain configuration with the Builder pattern characteristics.
- `ZTestSuite` (ztest_suite.hpp): A test suite class that inherits from ZTestBase, used to organize test cases, implements the Template Method pattern, and supports extension with hook functions.
- `ZBenchMark` (ztest_benchmark.hpp): A benchmark test class that inherits from ZTestBase and overrides the run() method to implement benchmark testing through iterative execution of the test function.
- `ZTestParameterized` (ztest_parameterized.hpp): A parameterized test base class that uses the Composite pattern to encapsulate test data and implements the data-driven testing framework through the run() method.
- `Macro Definition System` (ztest_macros.hpp): Implements syntactic sugar through macro definitions, uses the Factory pattern to automatically generate test classes, and implements automatic registration with naming connection technology.

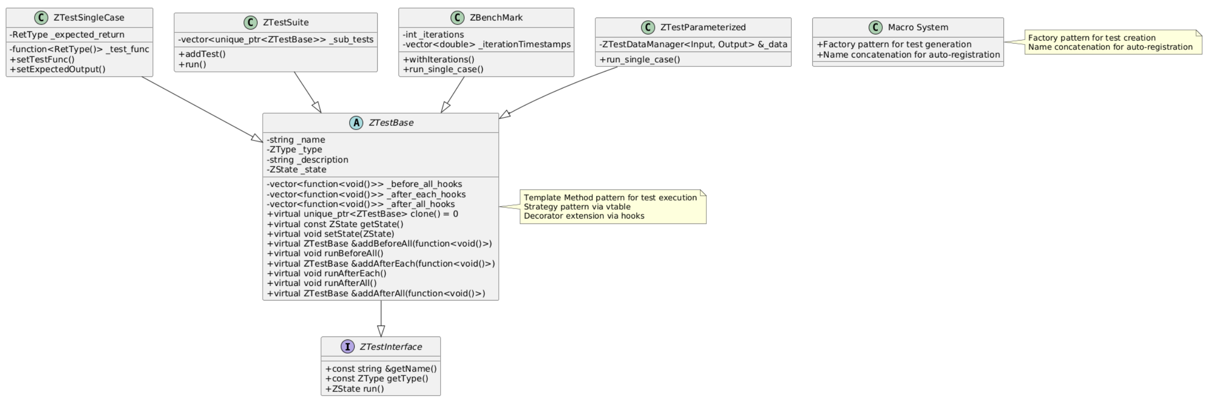

ZTestSingleCase
-RetType _expected_return
-function<RetType()> _test_func
+setTestFunc()
+setExpectedOutput()
ZTestSuite
-vector<unique_ptr<ZTestBase>> _sub_tests
+addTest()
+run()
ZBenchMark
-int _iterations
-vector<double> _iterationTimestamps
+withIterations()
+run_single_case()
ZTestParameterized
-ZTestDataManager<Input, Output> &_data
+run_single_case()
Macro System
+Factory pattern for test generation
+Name concatenation for auto-registration
Factory pattern for test creation
Name concatenation for auto-registration
ZTestBase
-string _name
-ZType _type
-string _description
-ZState _state
-vector<function<void()>> _before_all_hooks
-vector<function<void()>> _after_each_hooks
-vector<function<void()>> _after_all_hooks
+virtual unique_ptr<ZTestBase> clone() = 0
+virtual const ZState getState()
+virtual void setState(ZState)
+virtual ZTestBase &addBeforeAll(function<void()>)
+virtual void runBeforeAll()
+virtual ZTestBase &addAfterEach(function<void()>)
+virtual void runAfterEach()
+virtual void runAfterAll()
+virtual ZTestBase &addAfterAll(function<void()>)
Template Method pattern for test execution
Strategy pattern via vtable
Decorator extension via hooks
ZTestInterface
+const string &getName()
+const ZType getType()
+ZState run()

图 9: Ztest Class Diagram

4.3.2 Test Registration Related Classes

- `ZTestRegistry` (ztest_registry.hpp): A global registration center implemented with the Singleton pattern, managing test case sets and ensuring thread-safe registration and retrieval operations.

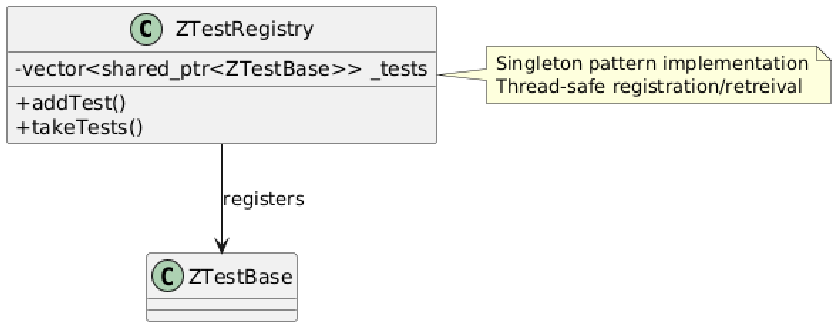

ZTestRegistry
-vector<shared_ptr<ZTestBase>> _tests
+addTest()
+takeTests()
Singleton pattern implementation
Thread-safe registration/retreival
registers
ZTestBase

图 10: Ztest Class Diagram

4.3.3 Test Execution Related Classes

- `ZTestContext` (ztest_context.hpp): The test execution context uses the Strategy pattern to select execution strategies based on test types and implements parallel test execution with the Thread Pool pattern.
- `ZThreadPool` (ztest_thread.hpp): A thread pool implementation that uses the Object Pool pattern, manages task queues with the Producer-Consumer pattern, and implements asynchronous execution monitoring with future/promise mechanisms.

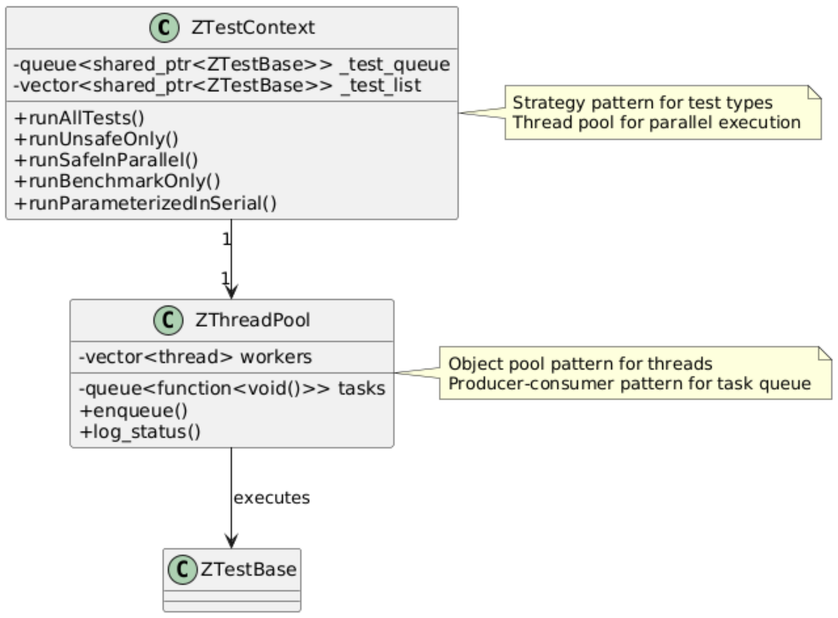

ZTestContext
-queue<shared_ptr<ZTestBase>> _test_queue
-vector<shared_ptr<ZTestBase>> _test_list
+runAllTests()
+runUnsafeOnly()
+runSafeInParallel()
+runBenchmarkOnly()
+runParameterizedInSerial()
Strategy pattern for test types
Thread pool for parallel execution
1
1
ZThreadPool
-vector<thread> workers
-queue<function<void()>> tasks
+enqueue()
+log_status()
Object pool pattern for threads
Producer-consumer pattern for task queue
executes
ZTestBase

图 11: Ztest Class Diagram

4.3.4 Data Management Related Classes

- `ZDataRegistry` (ztest_dataregistry.hpp): A data cache manager implemented with the Singleton pattern for global access, using an LRU strategy for cache eviction.
- `ZDataManager` (ztest_parameterized.hpp): An abstract base class for data management, providing data management interfaces and implementing iterator-like functionality similar to Python.

- `ZTestDataManager` (ztest_parameterized.hpp): A generic class that implements test data management functionality for user-specified data types in the code.
- `ZTestCSVDataManager` (ztest_parameterized.hpp): A class that inherits from `ZTestDataManager` and `ZDataManager`, implementing the functionality of reading test data from CSV files and processing it into a form suitable for parameterized testing.

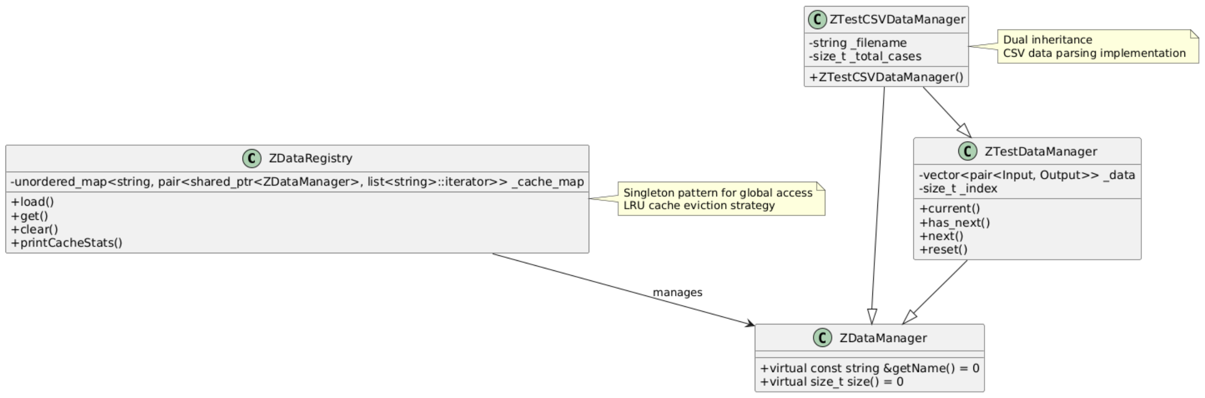

ZTestCSVDataManager
-string _filename
-size_t _total_cases
+ZTestCSVDataManager()
Dual inheritance
CSV data parsing implementation
ZDataRegistry
-unordered_map<string, pair<shared_ptr<ZDataManager>, list<string>::iterator>> _cache_map
+load()
+get()
+clear()
+printCacheStats()
Singleton pattern for global access
LRU cache eviction strategy
ZTestDataManager
-vector<pair<Input, Output>> _data
-size_t _index
+current()
+has_next()
+next()
+reset()
manages
ZDataManager
+virtual const string &getName() = 0
+virtual size_t size() = 0

图 12: Ztest Class Diagram

4.3.5 Result Management Related Classes

- `ZTestResult` (ztest_result.hpp): A test result class implemented with the Value Object pattern, encapsulating immutable data such as test status and duration.
- `ZTestResultManager` (ztest_result.hpp): A result manager implemented with the Singleton pattern, using the Chain of Responsibility pattern to handle result storage and querying.

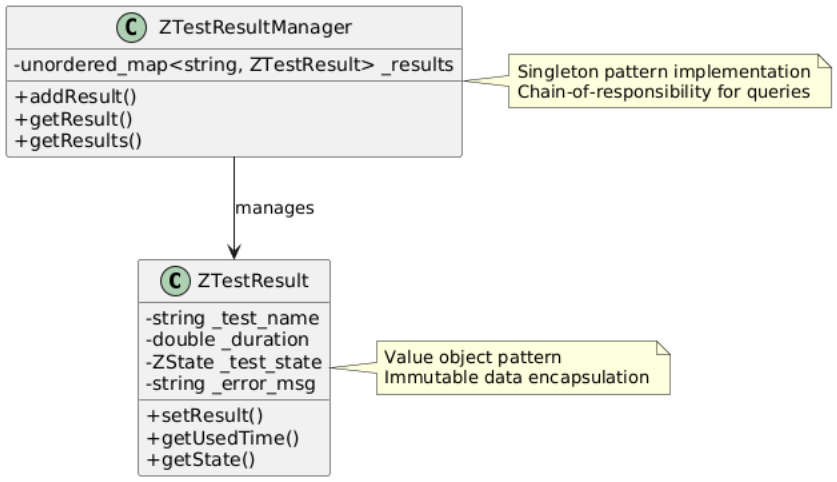


图 13: Ztest Class Diagram

4.3.6 Report Generation Related Classes

- `ZLogger` (ztest_logger.hpp): A multi-format report generator that uses the Template Method pattern to define the report generation process and supports HTML, JSON, and JUnit report formats.

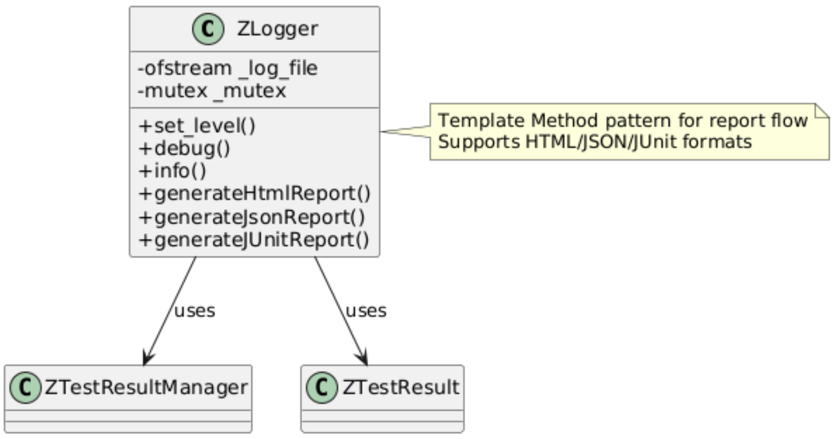


图 14: Ztest Class Diagram

4.3.7 Utility Module Related Classes

- `ZTimer` (ztest_timer.hpp): A timer implemented with the RAII pattern, encapsulating time measurement functionality.
- `CSVStream` (ztest_utils.hpp): Implements CSV stream operations similar to the standard input/output library, supporting read, write, and basic information printing.

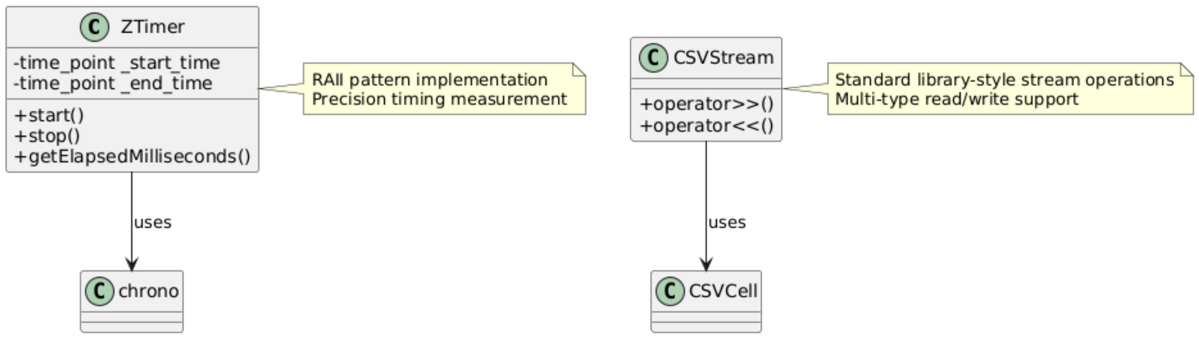

ZTimer
-time_point _start_time
-time_point _end_time
+start()
+stop()
+getElapsedMilliseconds()
RAII pattern implementation
Precision timing measurement
uses
chrono
CSVStream
+operator>>()
+operator<<()
Standard library-style stream operations
Multi-type read/write support
uses
CSVCell

图 15: Ztest Class Diagram

4.3.8 GUI Module Related Classes

- `ZTestModel` (gui.hpp): The Model layer in the MVC architecture, using the Observer pattern to monitor test status changes.
- `ZTestController` (gui.hpp): The Controller layer in the MVC architecture, implementing the Command pattern to encapsulate test execution operations.
- `ZTestView` (gui.hpp): The View layer in the MVC architecture, using the Bridge pattern to separate interface elements from implementation.

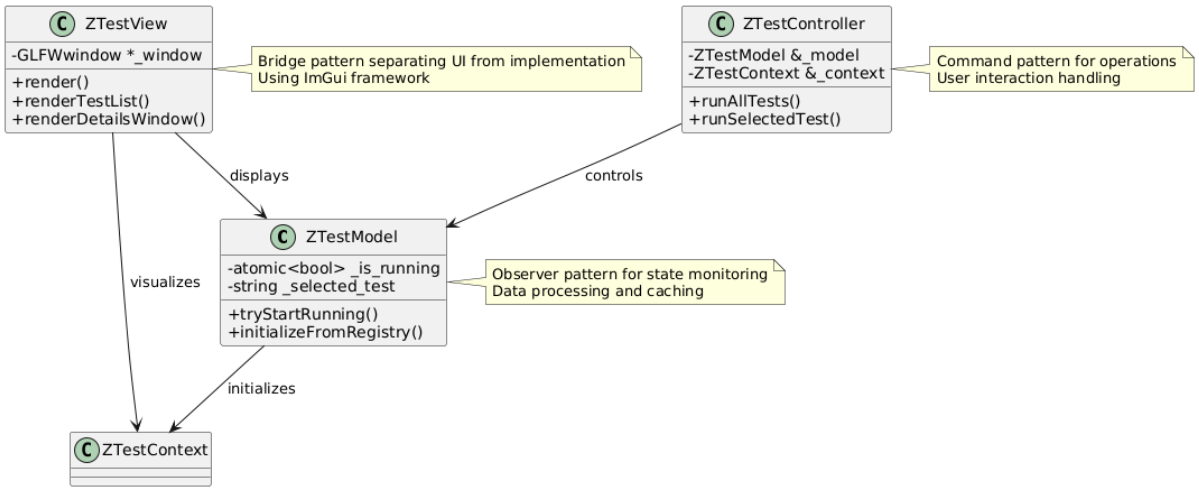


图 16: Ztest Class Diagram

5 Programming Progress

Task Phase	**Time**	**Plan**	**Actual**
Determine Topic	2024.04.26-2024.05.03	Investigate the topic and submit a proposal.	Done
Implement Core Code	2024.05.03-2024.05.05	Implemented the execution logic of GUI and Safe test and Unsafe test	Done
Major Performance Optimization	2024.05.05-2024.05.08	Thread pool optimization	Done
Function Optimization	2024.05.08-2024.05.11	Improved JSON, HTML output, JUnit format output, CI integration	Done
Function Optimization	2024.05.11-2024.05.13	Introduced imgui Docking functionality in GUI	Done
Function Optimization	2024.05.13-2024.05.15	Added CLI	Done
Major Function Optimization	2024.05.15-2024.05.24	Added BENCHMARK testing	Done
Function Optimization	2024.05.24-2024.05.25	Added device status monitoring and visualization	Done
Major Function Optimization	2024.05.25-2024.05.28	Added parameterized testing and data-driven functionality	Done
Major Performance Optimization	2024.05.28-2024.06.02	Implemented caching and LRU cache eviction for data	Done
Major Function Optimization	2024.06.02-2024.06.04	Added AI diagnosis	Done
Summary Work	2024.06.04-2024.06.05	Cross-platform porting & Report writing	Done

表 3: Programming Progress

6 Test Report

6.1 Functional Testing & System Testing

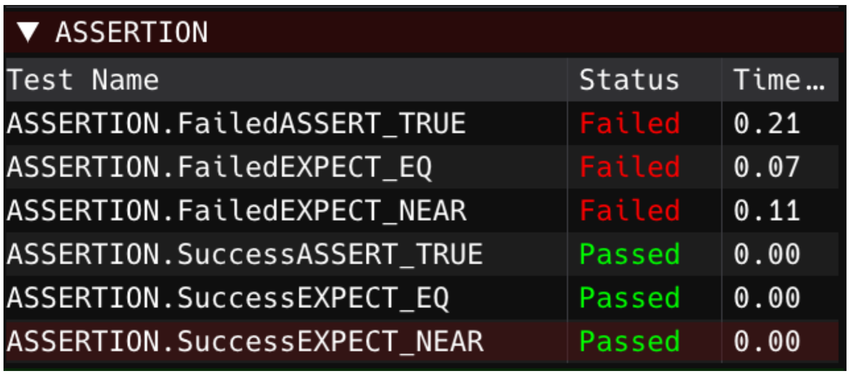
6.1.1 Assertion Function Testing

This test aims to verify the behavior of assertion macros such as EXPECT_EQ, EXPECT_NEAR, and ASSERT_TRUE in both successful and failed cases. By designing different test cases, we can check whether these assertions correctly identify the match between expected and actual results, ensuring that the assertion function of the test framework works reliably.

```
int add(int a, int b) { return a + b; }
double subtract(double a, double b) { return a - b; }
ZTEST_F(ASSERTION, FailedEXPECT_EQ) {
  EXPECT_EQ(6, add(2, 3));
  return ZState::z_success;
}
ZTEST_F(ASSERTION, SuccessEXPECT_EQ) {
  EXPECT_EQ(5, add(2, 3));
  return ZState::z_success;
}
ZTEST_F(ASSERTION, SuccessEXPECT_NEAR) {
  EXPECT_NEAR(2, subtract(5.0, 3.0), 0.001);
  return ZState::z_success;
}

ZTEST_F(ASSERTION, FailedEXPECT_NEAR) {
  EXPECT_NEAR(2, subtract(5.1, 3.0), 0.001);
  return ZState::z_success;
}

ZTEST_F(ASSERTION, FailedASSERT_TRUE) {
  ASSERT_TRUE(false);
  return ZState::z_success;
}
ZTEST_F(ASSERTION, SuccessASSERT_TRUE) {
  ASSERT_TRUE(true);
  return ZState::z_success;
}
```



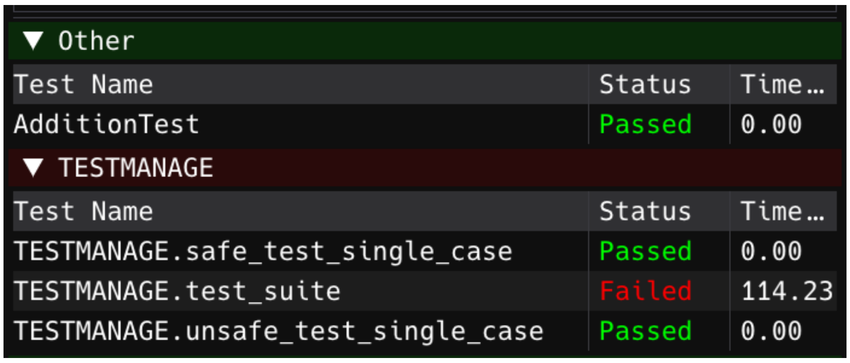
```
▼ ASSERTION
Test Name                           Status   Time…
ASSERTION.FailedASSERT_TRUE         Failed   0.21
ASSERTION.FailedEXPECT_EQ           Failed   0.07
ASSERTION.FailedEXPECT_NEAR         Failed   0.11
ASSERTION.SuccessASSERT_TRUE        Passed   0.00
ASSERTION.SuccessEXPECT_EQ          Passed   0.00
ASSERTION.SuccessEXPECT_NEAR        Passed   0.00
```

图 17: Assertion Function Test

6.1.2 Test Management Testing

This test module aims to verify the completeness and reliability of the test management system. By defining various test cases, including single safe tests, single unsafe tests, and test suites with multiple assertions, the system can comprehensively cover different test scenarios. Additionally, the dynamic test case construction functionality allows flexible creation and registration of new test cases, further enhancing the extensibility of the test framework. During testing, memory allocation, function call verification, and assertion checks ensure the correct execution of test cases and consistency with expected results. Moreover, the test framework provides setup and teardown hooks for necessary configurations and cleanup operations before and after tests, ensuring the stability of the test environment and the accuracy of test results.

```cpp
ZTEST_F(TESTMANAGE, safe_test_single_case, safe) {
  ASSERT_TRUE(true);
  return ZState::z_success;
}
ZTEST_F(TESTMANAGE, unsafe_test_single_case, unsafe) {
  ASSERT_TRUE(true);
  return ZState::z_success;
}
ZTEST_F(TESTMANAGE, test_suite) {
  const size_t MB100 = 100 * 1024 * 1024;
  auto ptr = std::make_unique<char[]>(MB100);
  ASSERT_TRUE(ptr != nullptr);
  EXPECT_EQ(3, subtract(5, 3));
  EXPECT_EQ(6, add(2, 3));
  return ZState::z_success;
}
void createSingleTestCase() {
  // Use TestBuilder to construct test
  auto test =
    TestFactory::createTest("AdditionTest",          // Test name
                            ZType::z_safe,           // Execution
                            "Test addition functionality", // Description
                            add, 2, 3 // Function and arguments
                            )
        .setExpectedOutput(5) // Set expected result
        .beforeAll([]() {  // Setup hook
          std::cout << "Setting up single test..." << std::endl;
        })
        .afterEach([]() { // Teardown hook
          std::cout << "Cleaning up after test..." << std::endl;
        })
        .withDescription("Verify basic addition")
        .registerTest()
        .build(); // Register with test
}
// in main()
createSingleTestCase();
```



▼ Other

Test Name	Status	Time…
AdditionTest	Passed	0.00

▼ TESTMANAGE

Test Name	Status	Time…
TESTMANAGE.safe_test_single_case	Passed	0.00
TESTMANAGE.test_suite	Failed	114.23
TESTMANAGE.unsafe_test_single_case	Passed	0.00

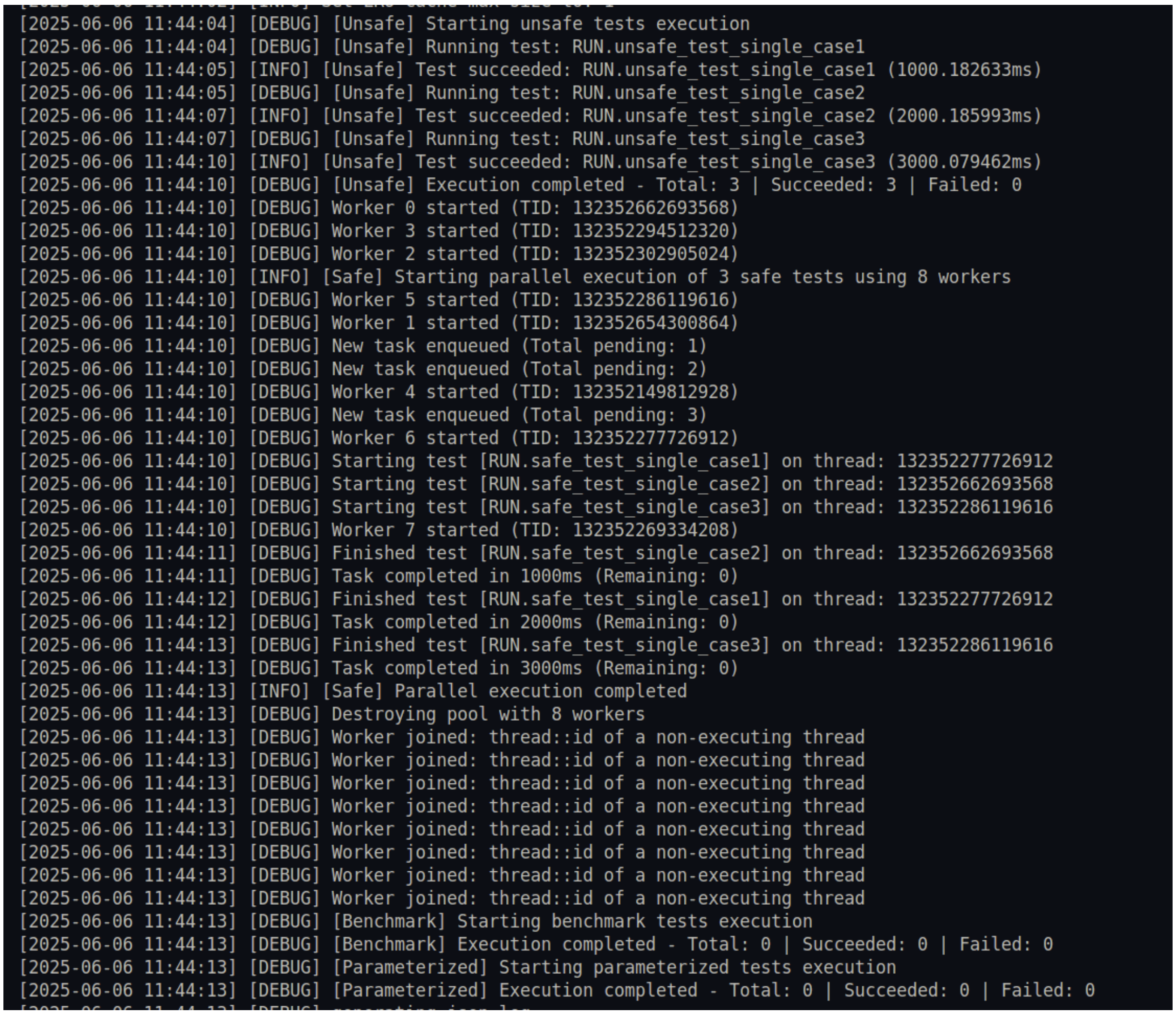
图 18: Test Management Function Test

6.1.3 Test Execution Management

In this test module, the efficiency and accuracy of the test execution management mechanism are primarily verified. Multiple safe and unsafe test cases are defined to simulate different execution times and task types. Each test case uses the sleep function to simulate different execution times to test the performance differences between parallel and sequential execution. The test results are explained as follows:

- A thread pool with eight threads is used to execute three test cases in parallel.
- Three test cases are executed sequentially.

```
ZTEST_F(RUN, safe_test_single_case1, safe) {
  sleep(2);
  ASSERT_TRUE(true);
  return ZState::z_success;
}
ZTEST_F(RUN, safe_test_single_case2, safe) {
  sleep(1);
  ASSERT_TRUE(true);
  return ZState::z_success;
}
ZTEST_F(RUN, safe_test_single_case3, safe) {
  sleep(3);
  ASSERT_TRUE(true);
  return ZState::z_success;
}
ZTEST_F(RUN, unsafe_test_single_case1, unsafe) {
  sleep(1);
  EXPECT_EQ(false, false);
  return ZState::z_success;
}
ZTEST_F(RUN, unsafe_test_single_case2, unsafe) {
  sleep(2);
  EXPECT_EQ(false, false);
  return ZState::z_success;
}
ZTEST_F(RUN, unsafe_test_single_case3, unsafe) {
  sleep(3);
  EXPECT_EQ(false, false);
  return ZState::z_success;
}
```



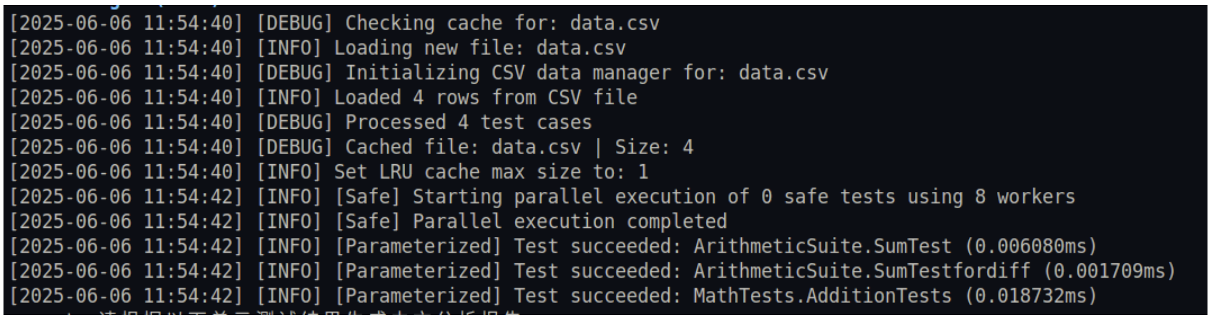
```
[2025-06-06 11:44:04] [DEBUG] [Unsafe] Starting unsafe tests execution
[2025-06-06 11:44:04] [DEBUG] [Unsafe] Running test: RUN.unsafe_test_single_case1
[2025-06-06 11:44:05] [INFO] [Unsafe] Test succeeded: RUN.unsafe_test_single_case1 (1000.182633ms)
[2025-06-06 11:44:05] [DEBUG] [Unsafe] Running test: RUN.unsafe_test_single_case2
[2025-06-06 11:44:07] [INFO] [Unsafe] Test succeeded: RUN.unsafe_test_single_case2 (2000.185993ms)
[2025-06-06 11:44:07] [DEBUG] [Unsafe] Running test: RUN.unsafe_test_single_case3
[2025-06-06 11:44:10] [INFO] [Unsafe] Test succeeded: RUN.unsafe_test_single_case3 (3000.079462ms)
[2025-06-06 11:44:10] [DEBUG] [Unsafe] Execution completed - Total: 3 | Succeeded: 3 | Failed: 0
[2025-06-06 11:44:10] [DEBUG] Worker 0 started (TID: 132352662693568)
[2025-06-06 11:44:10] [DEBUG] Worker 3 started (TID: 132352294512320)
[2025-06-06 11:44:10] [DEBUG] Worker 2 started (TID: 132352302905024)
[2025-06-06 11:44:10] [INFO] [Safe] Starting parallel execution of 3 safe tests using 8 workers
[2025-06-06 11:44:10] [DEBUG] Worker 5 started (TID: 132352286119616)
[2025-06-06 11:44:10] [DEBUG] Worker 1 started (TID: 132352654300864)
[2025-06-06 11:44:10] [DEBUG] New task enqueued (Total pending: 1)
[2025-06-06 11:44:10] [DEBUG] New task enqueued (Total pending: 2)
[2025-06-06 11:44:10] [DEBUG] Worker 4 started (TID: 132352149812928)
[2025-06-06 11:44:10] [DEBUG] New task enqueued (Total pending: 3)
[2025-06-06 11:44:10] [DEBUG] Worker 6 started (TID: 132352277726912)
[2025-06-06 11:44:10] [DEBUG] Starting test [RUN.safe_test_single_case1] on thread: 132352277726912
[2025-06-06 11:44:10] [DEBUG] Starting test [RUN.safe_test_single_case2] on thread: 132352662693568
[2025-06-06 11:44:10] [DEBUG] Starting test [RUN.safe_test_single_case3] on thread: 132352286119616
[2025-06-06 11:44:10] [DEBUG] Worker 7 started (TID: 132352269334208)
[2025-06-06 11:44:11] [DEBUG] Finished test [RUN.safe_test_single_case2] on thread: 132352662693568
[2025-06-06 11:44:11] [DEBUG] Task completed in 1000ms (Remaining: 0)
[2025-06-06 11:44:12] [DEBUG] Finished test [RUN.safe_test_single_case1] on thread: 132352277726912
[2025-06-06 11:44:12] [DEBUG] Task completed in 2000ms (Remaining: 0)
[2025-06-06 11:44:13] [DEBUG] Finished test [RUN.safe_test_single_case3] on thread: 132352286119616
[2025-06-06 11:44:13] [DEBUG] Task completed in 3000ms (Remaining: 0)
[2025-06-06 11:44:13] [INFO] [Safe] Parallel execution completed
[2025-06-06 11:44:13] [DEBUG] Destroying pool with 8 workers
[2025-06-06 11:44:13] [DEBUG] Worker joined: thread::id of a non-executing thread
[2025-06-06 11:44:13] [DEBUG] Worker joined: thread::id of a non-executing thread
[2025-06-06 11:44:13] [DEBUG] Worker joined: thread::id of a non-executing thread
[2025-06-06 11:44:13] [DEBUG] Worker joined: thread::id of a non-executing thread
[2025-06-06 11:44:13] [DEBUG] Worker joined: thread::id of a non-executing thread
[2025-06-06 11:44:13] [DEBUG] Worker joined: thread::id of a non-executing thread
[2025-06-06 11:44:13] [DEBUG] Worker joined: thread::id of a non-executing thread
[2025-06-06 11:44:13] [DEBUG] Worker joined: thread::id of a non-executing thread
[2025-06-06 11:44:13] [DEBUG] [Benchmark] Starting benchmark tests execution
[2025-06-06 11:44:13] [DEBUG] [Benchmark] Execution completed - Total: 0 | Succeeded: 0 | Failed: 0
[2025-06-06 11:44:13] [DEBUG] [Parameterized] Starting parameterized tests execution
[2025-06-06 11:44:13] [DEBUG] [Parameterized] Execution completed - Total: 0 | Succeeded: 0 | Failed: 0
```

图 19: Test Executor Function Test

6.1.4 Data-Driven Testing

Two types of data-driven test cases are demonstrated, using in-memory datasets and external CSV files as data sources for testing.

```
ZTestDataManager<vector<int>, int> sum_test_data = {
    {{1, 2}, 3}, {{-1, 1}, 0}, {{10, 20}, 30}};
ZTEST_P(ArithmeticSuite, SumTest, sum_test_data) {
  auto &&[inputs, expected] = _data.current();
  int actual = inputs[0] + inputs[1];
  EXPECT_EQ_FOREACH(expected, actual);
  return ZState::z_success;
}
ZTestDataManager<tuple<float, int>, float> sum_test_data2 = {
    {{1.2, 2}, 3.2}, {{-1.0, 1}, 0.0}, {{10.1, 20}, 30.2}};
ZTEST_P(ArithmeticSuite, SumTestfordiff, sum_test_data2) {
  auto &&[inputs, expected] = _data.current();
  float actual = std::get<0>(inputs) + std::get<1>(inputs);
  EXPECT_EQ_FOREACH(expected, actual);
  return ZState::z_success;
}
ZTEST_P_CSV(MathTests, AdditionTests, "data.csv") {
  auto inputs = getInput();
  auto expected = getOutput();
  double actual = std::get<double>(inputs[0]) + std::get<double>(inputs[1]);
  EXPECT_EQ(actual, std::get<double>(expected));
  return ZState::z_success;
}
```



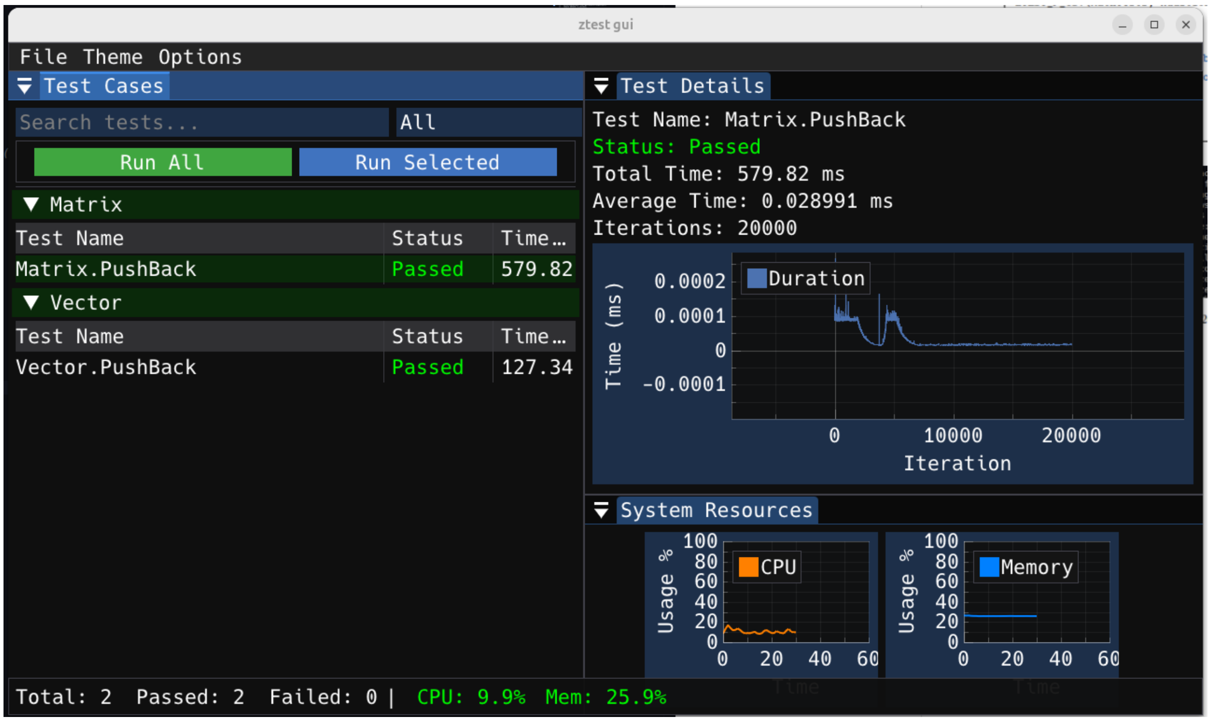
```
[2025-06-06 11:54:40] [DEBUG] Checking cache for: data.csv
[2025-06-06 11:54:40] [INFO] Loading new file: data.csv
[2025-06-06 11:54:40] [DEBUG] Initializing CSV data manager for: data.csv
[2025-06-06 11:54:40] [INFO] Loaded 4 rows from CSV file
[2025-06-06 11:54:40] [DEBUG] Processed 4 test cases
[2025-06-06 11:54:40] [DEBUG] Cached file: data.csv | Size: 4
[2025-06-06 11:54:40] [INFO] Set LRU cache max size to: 1
[2025-06-06 11:54:42] [INFO] [Safe] Starting parallel execution of 0 safe tests using 8 workers
[2025-06-06 11:54:42] [INFO] [Safe] Parallel execution completed
[2025-06-06 11:54:42] [INFO] [Parameterized] Test succeeded: ArithmeticSuite.SumTest (0.006080ms)
[2025-06-06 11:54:42] [INFO] [Parameterized] Test succeeded: ArithmeticSuite.SumTestfordiff (0.001709ms)
[2025-06-06 11:54:42] [INFO] [Parameterized] Test succeeded: MathTests.AdditionTests (0.018732ms)
```

图 20: Data-Driven Function Test

6.1.5 Benchmark Testing

Testing the definition of benchmark-type tests, visualization of test time distribution, and monitoring of CPU and memory usage on the GUI.

```
ZBENCHMARK(Vector, PushBack) {
  std::vector<int> v;
  for (int i = 0; i < 10000; ++i)
    v.push_back(i);
  return ZState::z_success;
}
ZBENCHMARK(Matrix, PushBack, 20000) {
  std::vector<int> v;
  for (int i = 0; i < 1000; ++i)
    v.push_back(random());
  return ZState::z_success;
}
```



Matrix

Test Name	Status	Time…
Matrix.PushBack	Passed	579.82

Vector

Test Name	Status	Time…
Vector.PushBack	Passed	127.34

Test Details

Test Name: Matrix.PushBack
Status: Passed
Total Time: 579.82 ms
Average Time: 0.028991 ms
Iterations: 20000

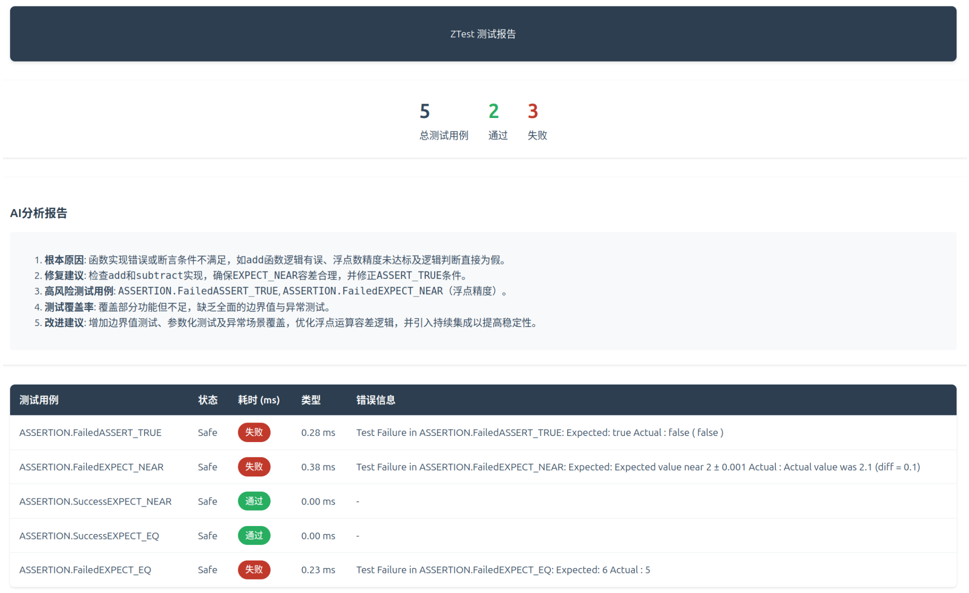
Total: 2 Passed: 2 Failed: 0 | CPU: 9.9% Mem: 25.9%

图 21: Benchmark Function Test

•

6.1.6 Report Generation Testing

Testing the generation of HTML, JSON, and XML format reports.



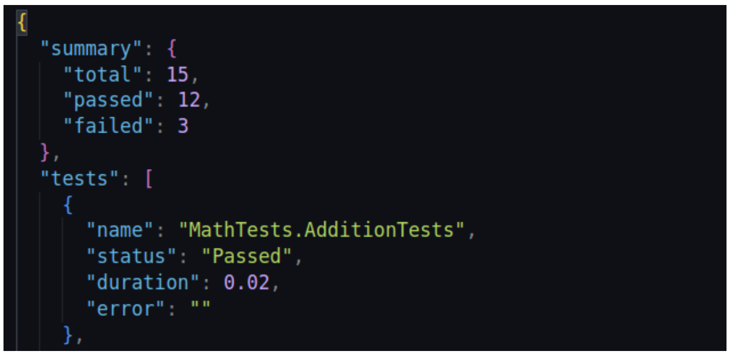
5 总测试用例 2 通过 3 失败

AI分析报告

1. **根本原因**: 函数实现错误或断言条件不满足，如add函数逻辑有误、浮点数精度未达标及逻辑判断直接为假。
2. **修复建议**: 检查add和subtract实现，确保EXPECT_NEAR容差合理，并修正ASSERT_TRUE条件。
3. **高风险测试用例**: ASSERTION.FailedASSERT_TRUE, ASSERTION.FailedEXPECT_NEAR（浮点精度）。
4. **测试覆盖率**: 覆盖部分功能但不足，缺乏全面的边界值与异常测试。
5. **改进建议**: 增加边界值测试、参数化测试及异常场景覆盖，优化浮点运算容差逻辑，并引入持续集成以提高稳定性。

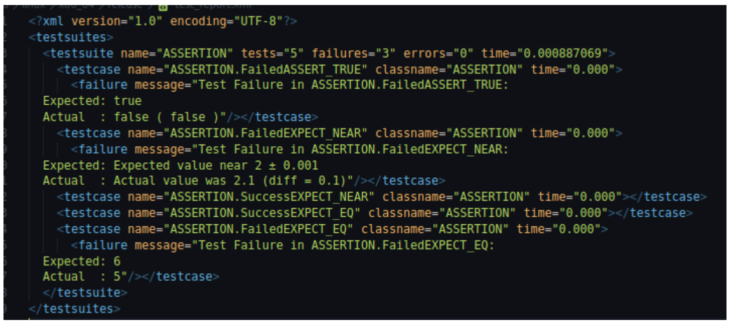
测试用例	状态	耗时 (ms)	类型	错误信息
ASSERTION.FailedASSERT_TRUE	Safe	失败	0.28 ms	Test Failure in ASSERTION.FailedASSERT_TRUE: Expected: true Actual : false (false)
ASSERTION.FailedEXPECT_NEAR	Safe	失败	0.38 ms	Test Failure in ASSERTION.FailedEXPECT_NEAR: Expected: Expected value near 2 ± 0.001 Actual : Actual value was 2.1 (diff = 0.1)
ASSERTION.SuccessEXPECT_NEAR	Safe	通过	0.00 ms	-
ASSERTION.SuccessEXPECT_EQ	Safe	通过	0.00 ms	-
ASSERTION.FailedEXPECT_EQ	Safe	失败	0.23 ms	Test Failure in ASSERTION.FailedEXPECT_EQ: Expected: 6 Actual : 5

图 22: HTML Test Report



```json
{
  "summary": {
    "total": 15,
    "passed": 12,
    "failed": 3
  },
  "tests": [
    {
      "name": "MathTests.AdditionTests",
      "status": "Passed",
      "duration": 0.02,
      "error": ""
    },
```

图 23: Partial JSON Report

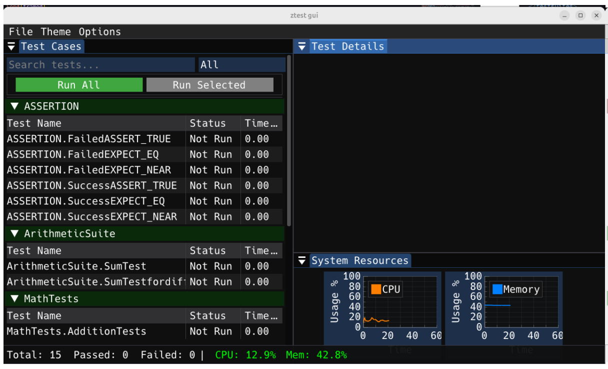


```xml
<?xml version="1.0" encoding="UTF-8"?>
<testsuites>
  <testsuite name="ASSERTION" tests="5" failures="3" errors="0" time="0.000887069">
    <testcase name="ASSERTION.FailedASSERT_TRUE" classname="ASSERTION" time="0.000">
      <failure message="Test Failure in ASSERTION.FailedASSERT_TRUE:
  Expected: true
  Actual  : false ( false )"/></testcase>
    <testcase name="ASSERTION.FailedEXPECT_NEAR" classname="ASSERTION" time="0.000">
      <failure message="Test Failure in ASSERTION.FailedEXPECT_NEAR:
  Expected: Expected value near 2 ± 0.001
  Actual  : Actual value was 2.1 (diff = 0.1)"/></testcase>
    <testcase name="ASSERTION.SuccessEXPECT_NEAR" classname="ASSERTION" time="0.000"></testcase>
    <testcase name="ASSERTION.SuccessEXPECT_EQ" classname="ASSERTION" time="0.000"></testcase>
    <testcase name="ASSERTION.FailedEXPECT_EQ" classname="ASSERTION" time="0.000">
      <failure message="Test Failure in ASSERTION.FailedEXPECT_EQ:
  Expected: 6
  Actual  : 5"/></testcase>
  </testsuite>
</testsuites>
```

图 24: XML (JUnit Format) Report

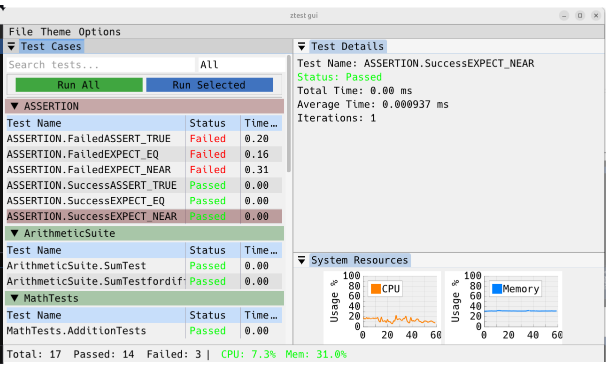
6.1.7 GUI Testing

Testing the display of the GUI, theme switching, AI assistant display, and other functions.



```
▼ ASSERTION
Test Name                      Status   Time…
ASSERTION.FailedASSERT_TRUE    Not Run  0.00
ASSERTION.FailedEXPECT_EQ      Not Run  0.00
ASSERTION.FailedEXPECT_NEAR    Not Run  0.00
ASSERTION.SuccessASSERT_TRUE   Not Run  0.00
ASSERTION.SuccessEXPECT_EQ     Not Run  0.00
ASSERTION.SuccessEXPECT_NEAR   Not Run  0.00
▼ ArithmeticSuite
Test Name                      Status   Time…
ArithmeticSuite.SumTest        Not Run  0.00
ArithmeticSuite.SumTestfordif  Not Run  0.00
▼ MathTests
Test Name                      Status   Time…
MathTests.AdditionTests        Not Run  0.00
Total: 15  Passed: 0  Failed: 0 |  CPU: 12.9%  Mem: 42.8%
```

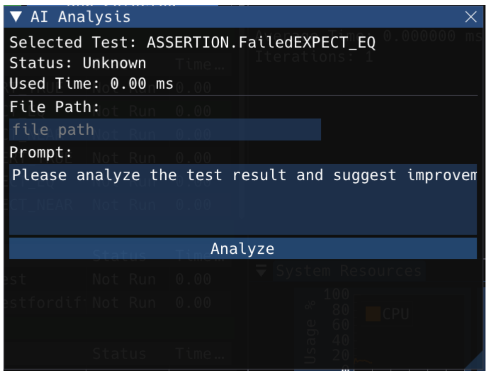
图 25: Dark Theme GUI Display



```
▼ ASSERTION
Test Name                      Status   Time…
ASSERTION.FailedASSERT_TRUE    Failed   0.20
ASSERTION.FailedEXPECT_EQ      Failed   0.16
ASSERTION.FailedEXPECT_NEAR    Failed   0.31
ASSERTION.SuccessASSERT_TRUE   Passed   0.00
ASSERTION.SuccessEXPECT_EQ     Passed   0.00
ASSERTION.SuccessEXPECT_NEAR   Passed   0.00
▼ ArithmeticSuite
Test Name                      Status   Time…
ArithmeticSuite.SumTest        Passed   0.00
ArithmeticSuite.SumTestfordif  Passed   0.00
▼ MathTests
Test Name                      Status   Time…
MathTests.AdditionTests        Passed   0.00
Total: 17  Passed: 14  Failed: 3 |  CPU: 7.3%  Mem: 31.0%
```

```
Test Name: ASSERTION.SuccessEXPECT_NEAR
Status: Passed
Total Time: 0.00 ms
Average Time: 0.000937 ms
Iterations: 1
```

图 26: Light Theme GUI Display

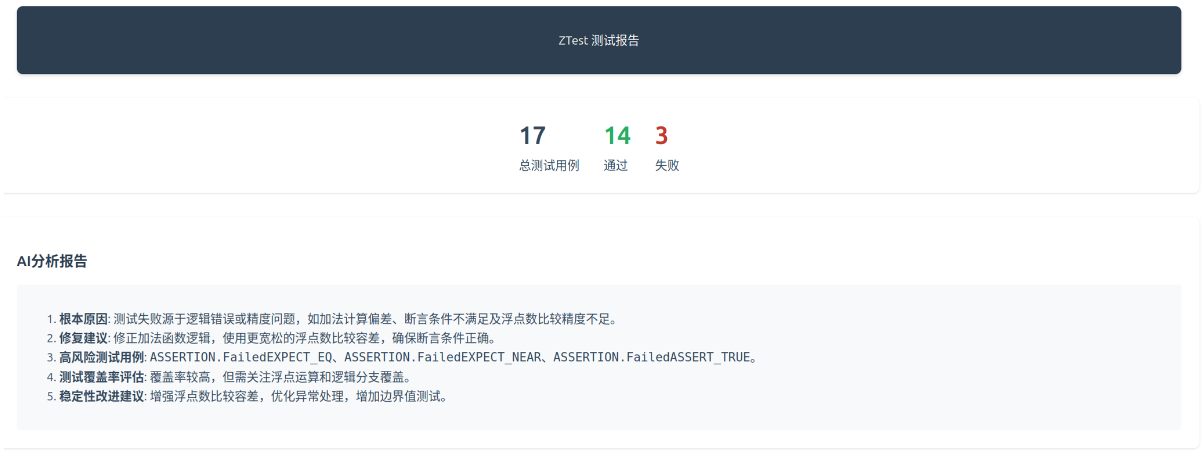


```
▼ AI Analysis
Selected Test: ASSERTION.FailedEXPECT_EQ
Status: Unknown
Used Time: 0.00 ms
File Path:
file path
Prompt:
Please analyze the test result and suggest improvem
```

图 27: AI Diagnosis Assistant Display

6.1.8 AI Diagnosis Testing

When generating reports, AI (qwen turbo) is called for diagnosis and written into the HTML report.



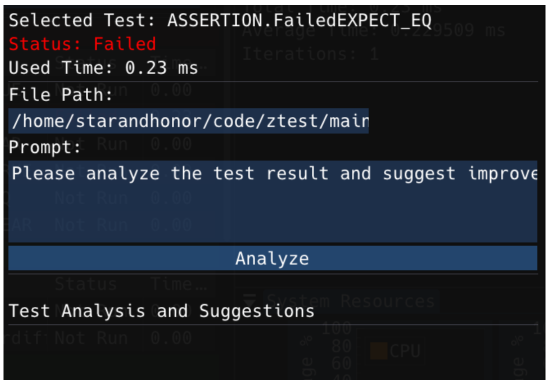
ZTest 测试报告

17 总测试用例 14 通过 3 失败

AI分析报告

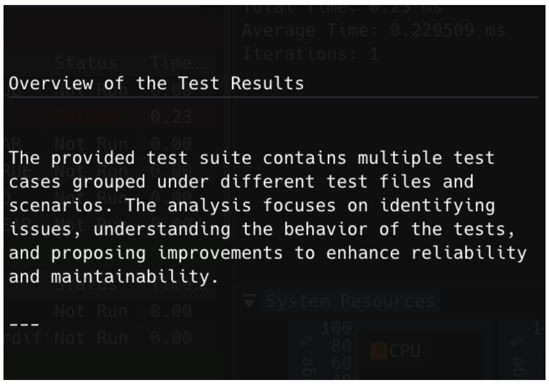
1. **根本原因**: 测试失败源于逻辑错误或精度问题，如加法计算偏差、断言条件不满足及浮点数比较精度不足。
2. **修复建议**: 修正加法函数逻辑，使用更宽松的浮点数比较容差，确保断言条件正确。
3. **高风险测试用例**: ASSERTION.FailedEXPECT_EQ、ASSERTION.FailedEXPECT_NEAR、ASSERTION.FailedASSERT_TRUE。
4. **测试覆盖率评估**: 覆盖率较高，但需关注浮点运算和逻辑分支覆盖。
5. **稳定性改进建议**: 增强浮点数比较容差，优化异常处理，增加边界值测试。

In the GUI interface, AI (qwen turbo) is called for diagnosis of individual test cases and displayed, allowing users to input custom prompts and file paths for the test.



```
Selected Test: ASSERTION.FailedEXPECT_EQ
Status: Failed
Used Time: 0.23 ms
File Path:
/home/starandhonor/code/ztest/main
Prompt:
Please analyze the test result and suggest improve
```

```
Test Analysis and Suggestions
```



```
Overview of the Test Results

The provided test suite contains multiple test
cases grouped under different test files and
scenarios. The analysis focuses on identifying
issues, understanding the behavior of the tests,
and proposing improvements to enhance reliability
and maintainability.

---
```

7 Personal Summary

Name	Insights
Zheng Chenyang	Emphasized the importance of modular design and layered architecture. By introducing the MVC pattern, the maintainability and scalability of the code were enhanced. Optimized resource monitoring and task scheduling in a multi-threaded environment to ensure system stability and performance. Combined AI analysis capabilities to provide intelligent feedback on test results, enhancing the tool's practicality. Future work will continue to iterate and improve functionality and user experience.
Ye Suohua	Mainly responsible for interface development, using the imgui library to complete the front-end architecture and rendering. Realized that front-end design needs to be based on user requirements, considering cross-platform compatibility and library version issues.
Qi Yansong	Participated in the development of the ”ztest” unit testing framework, applying it to game development and tool software design. Through innovative design, addressed the shortcomings of existing frameworks (such as Google Test) in GUI support, concurrent testing, and reporting systems, improving user experience and scalability. Through practice, gained a deeper understanding of modern C++ features and design patterns, recognizing that excellent software products need to balance technical depth and user experience, and that efficient team communication and collaboration are key.
Wang Ruizhen	Faced challenges in this large-scale complex project, practiced and learned new knowledge and skills, and enhanced problem-solving abilities.
Wu Hongqing	Responsible for the GUI design of the ztest project, using the ImGui framework to implement a visual interface. Designed simple and intuitive windows, coordinated the ZTestModel, ZTestView, and ZTestController classes, and improved user-friendliness. Through team collaboration, enhanced team cooperation skills and reduced compilation errors and understanding difficulties.

8 References

[1] Google Test. *Google Test Documentation.* [Online]. Available: https://github.com/google/googletest

[2] JUnit. *JUnit Documentation.* [Online]. Available: https://junit.org/junit5/

[3] Catch2. *Catch2 Documentation.* [Online]. Available: https://github.com/catchorg/Catch2

[4] Google Test Parallel. *Google Test Parallel Documentation.* [Online]. Available: https://github.com/google/gtest-parallel

[5] Pytest. *Pytest Documentation.* [Online]. Available: https://docs.pytest.org/en/latest/

[6] MiniUnit. *MiniUnit Documentation*. [Online]. Available: https://github.com/urin/miniunit