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 System Requirements Analysis

1.1 System Background and Motivation

As the need to cultivate students' complex system architecture design abilities grows, mastering object-oriented design principles and pattern-oriented engineering practices has become a key goal in advanced software engineering education. This requires not only an understanding of the dynamic 协作 relationship between classes and objects but also the ability to use design patterns to solve architectural problems through abstract thinking. Existing unit testing frameworks (such as Google Test) have a high learning curve, limited concurrency support, basic 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 environment for developers and testers to write, run, and manage test cases. This tool will support various test types (such as unit testing and integration testing) and offer detailed test result reports.

表 1: Comparison of Mainstream Testing Frameworks

Framework	GUI Support	Concurrent Testing	Reporting System	Data-Driven
gtest (C++)	None	Limited	Basic	Not Supported
JUnit (Java)	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

1.2 System Objectives

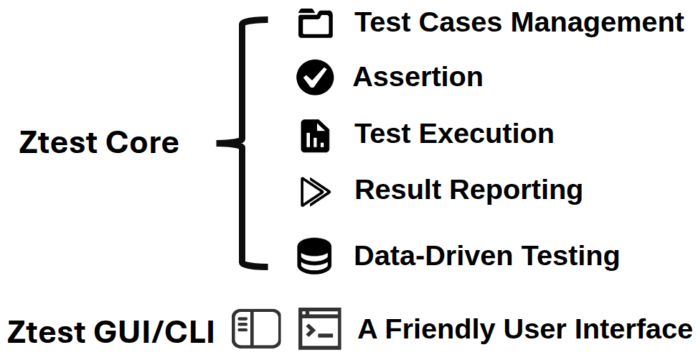


图 1: Ztest Function

1.2.1 Test Management

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

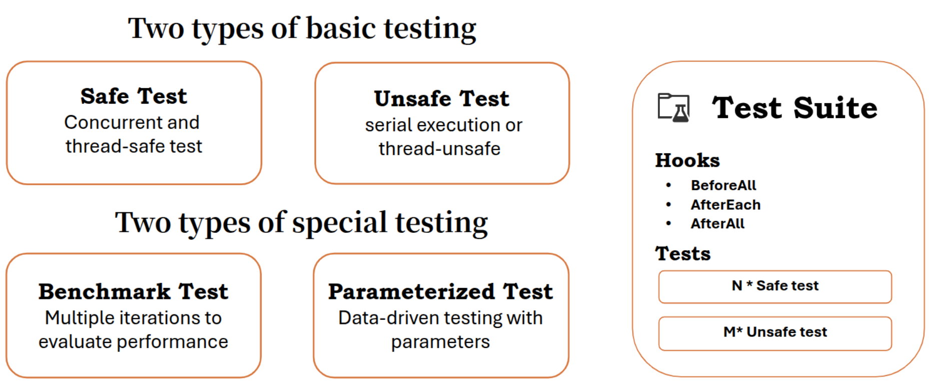

Two types of basic testing
Safe Test
Concurrent and thread-safe test
Unsafe Test
serial execution or thread-unsafe
Two types of special testing
Benchmark Test
Multiple iterations to evaluate performance
Parameterized Test
Data-driven testing with parameters
Test Suite
Hooks
BeforeAll
AfterEach
AfterAll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

图 2: Test Type

We offer 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 define test cases in a chained manner:

```
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();
```

1.2.2 Assertion Mechanism

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

- **EXPECT_EQ**: Verifies whether two values are equal.
- **ASSERT_TRUE**: Verifies whether a condition is true.

Usage example:

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

If an assertion fails, a **ZTestFailureException** is thrown. Custom exception handling logic can also be implemented by inheriting from **ZTestFailureException**.

1.2.3 Data-Driven Testing

Parameterized testing involves replacing certain fixed data in test cases with parameters and generating multiple sets of test data by changing the parameter values to test the software multiple times. It allows testers to cover multiple input scenarios with a single set of test logic instead of writing separate test code for each scenario. ZTest implements parameterized testing and uses CSV import and data parsing to achieve data-driven testing. To accelerate data access, imported data is cached. A lazy loading mechanism is employed, where data is only loaded into the cache when needed, reducing unnecessary resource consumption. Additionally, an LRU cache eviction strategy is used to ensure that the cached data is the most likely to be accessed again, thereby maximizing the effectiveness of the cache.

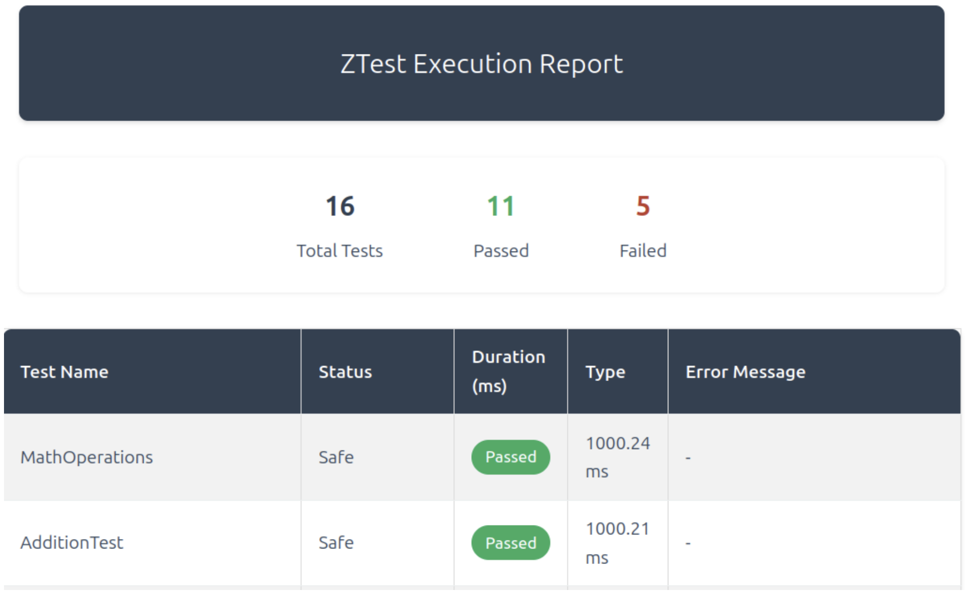
1.2.4 Test Executor

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

- Run safe tests in parallel. Thread creation and destruction are relatively time-consuming operations. Using a thread pool to manage the lifecycle of threads avoids frequent thread creation and destruction, thereby significantly improving system performance.
- Run unsafe tests in sequence. A queue is used to maintain test cases to ensure ordered execution.
- Run Benchmark tests in sequence.
- Run Parameterized tests in sequence.

1.2.5 Report Generation

It can generate HTML, JSON, and XML reports for easy 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	-

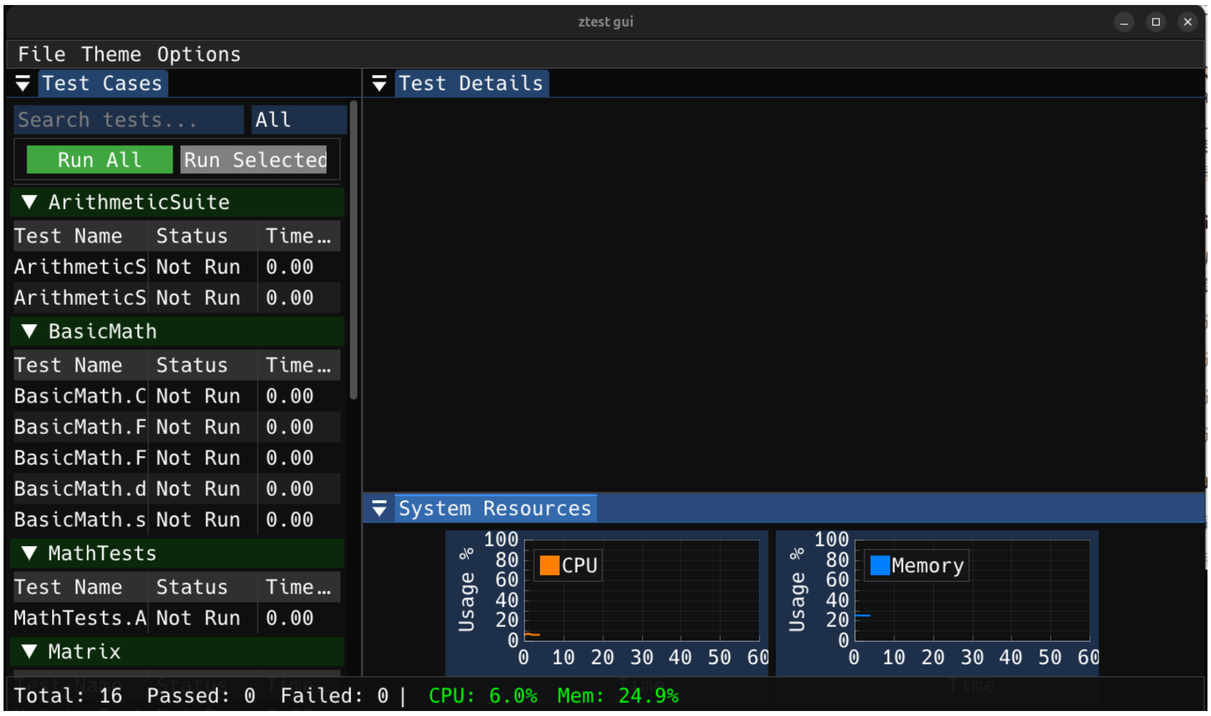
图 3: Test Result Display Interface Layout Diagram

1.2.6 CLI Interface

```
Usage: executor_name [OPTIONS]
Options:
--help                     Show help
--run-all                  Run all tests
--run-safe                 Run safe tests
--run-unsafe               Run unsafe tests
--run-benchmark            Run benchmark tests
--run-parameterized        Run parameterized tests
--run-test-case TEST_CASE  Run a specific test case
```

1.2.7 GUI Display

The GUI can display test results, filter test outcomes, monitor system resource status, and view details of test execution.



▼ 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 Diagram

1.2.8 AI Intelligent Diagnosis Function

By calling the qwen3 API, intelligent diagnosis functionality is achieved. Specific analysis of individual test cases can be obtained on the GUI interface. The report will include

1. Identifying the root cause of failures
2. Providing troubleshooting suggestions
3. Highlighting high-risk test cases
4. Evaluating overall test coverage
5. Proposing suggestions for improving system stability

2 Program Analysis

2.1 Key System Issues

2.1.1 Improving Test Execution Efficiency

Traditional C++ testing frameworks (such as gtest and catch2) often use sequential test execution, leading to long testing times and reduced productivity. Vibro coding has reduced the time required to write test cases, but the time to run them remains largely unchanged. We categorize tasks into four types: short testing time with a focus on result accuracy, such as addition operations, string concatenation, and user login validation; long testing time with a focus on result accuracy, such as merging multiple files, complex string matching and replacement, and large data sorting result validation; short testing time with a focus on process evaluation, such as reading or writing large files, low-complexity algorithm performance testing, and database queries; and long testing time with a focus on process evaluation, such as multi-threaded task processing, stress testing, and high-time-complexity algorithm testing.

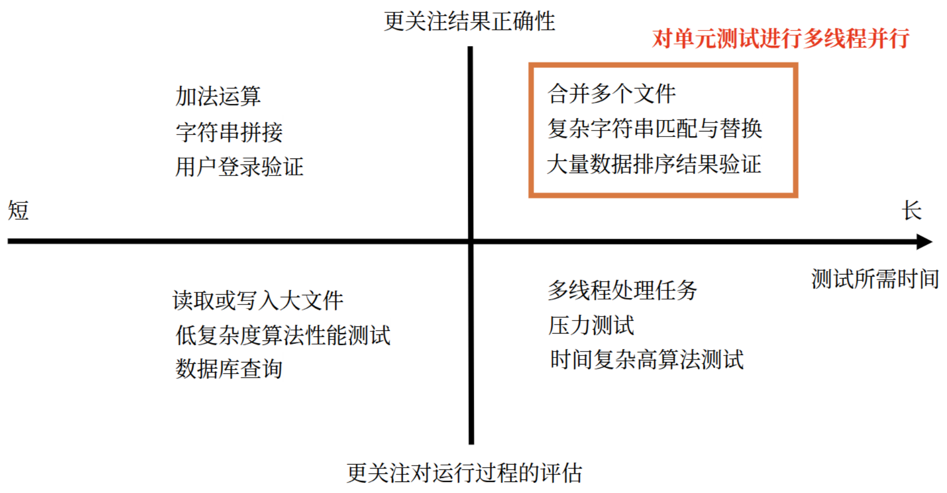


图 5: Task Type

For tasks with long testing times that focus on result accuracy and are thread-safe, ZTest introduces multi-threaded test execution directly into the C++ testing framework, rather than relying on third-party tools like google-parallel, achieving finer-grained control. Additionally, data-driven testing and caching have been introduced to significantly improve the definition and execution speed of large-scale unit tests.

2.1.2 Syntax Sugar Implementation

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

2.1.3 Automatic Type Inference of Functions to be Tested

When defining individual test cases using a chained approach, we aim for users to only need to provide the function name, parameters, and expected results, allowing our framework to handle the test execution logic and result verification. This requires automatic inference of the parameter types of the function under test to facilitate function calls for testing and result validation. We utilize the factory pattern to specify the return type and the builder pattern to construct parameters and set expected results.

2.2 Responsibility Allocation

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

3 Technical Approach

3.1 Runtime 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 cannot be used)
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)

3.2 Overall Design

3.2.1 Ztest Core Architecture Diagram

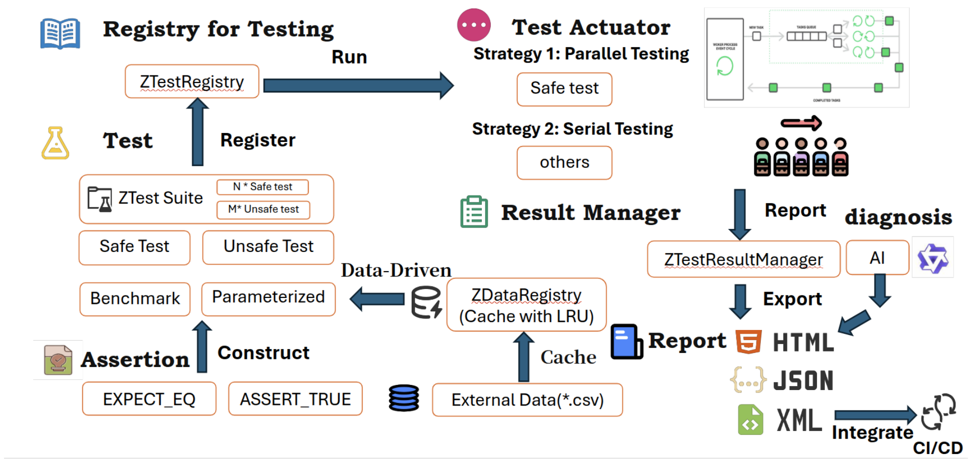


图 6: Ztest Design

The core architecture of ZTest begins with the test definition phase, where tests (including ZTest Suite, safe tests, unsafe tests, benchmark tests, and parameterized tests) use assertions such as EXPECT_EQ and ASSERT_TRUE to validate the correctness of test results. Next, tests are registered with ZTestRegistry for unified management. After registration, tests are sent to the test executor, which adopts different test strategies based on the test type. Safe tests are executed using a parallel strategy, while other test types use a serial strategy. During the execution of data-driven tests, the data-driven module manages external data (e.g., CSV files) through ZDataRegistry (a cache with LRU mechanism) to support test execution. After testing completion, ZTestResultManager collects and processes test results and exports them in HTML, JSON, or XML formats for reporting and further analysis. When exporting to HTML format, AI (qwen turbo) performs test diagnostics and integrates them into the HTML test report. Additionally, the entire testing process supports integration with continuous integration/continuous deployment (CI/CD) systems, enabling automatic feedback of test results into the development workflow to enhance software development efficiency and quality.

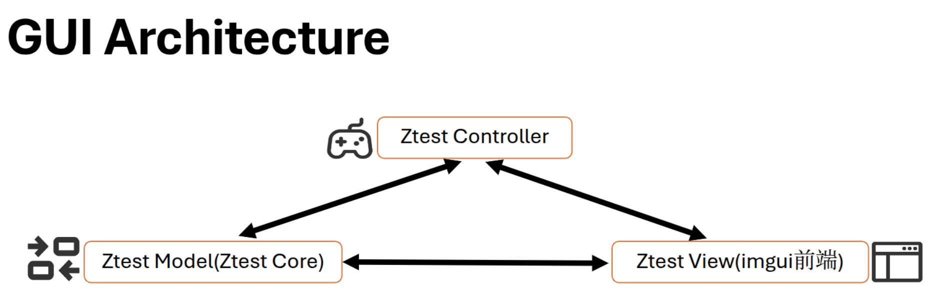


图 7: Ztest GUI Architecture

3.2.2 GUI Framework

The GUI framework is developed using the MVC architecture, where the Model layer handles data processing (further encapsulating Ztest core), the View layer is responsible for interface rendering (primarily using the ImGui framework), and the Controller layer manages user interactions. It translates user operations on the UI (such as clicks and filtering) into calls to the Model and View layers, thereby achieving updates to the user interface and data.

3.2.3 Overall Class Diagram

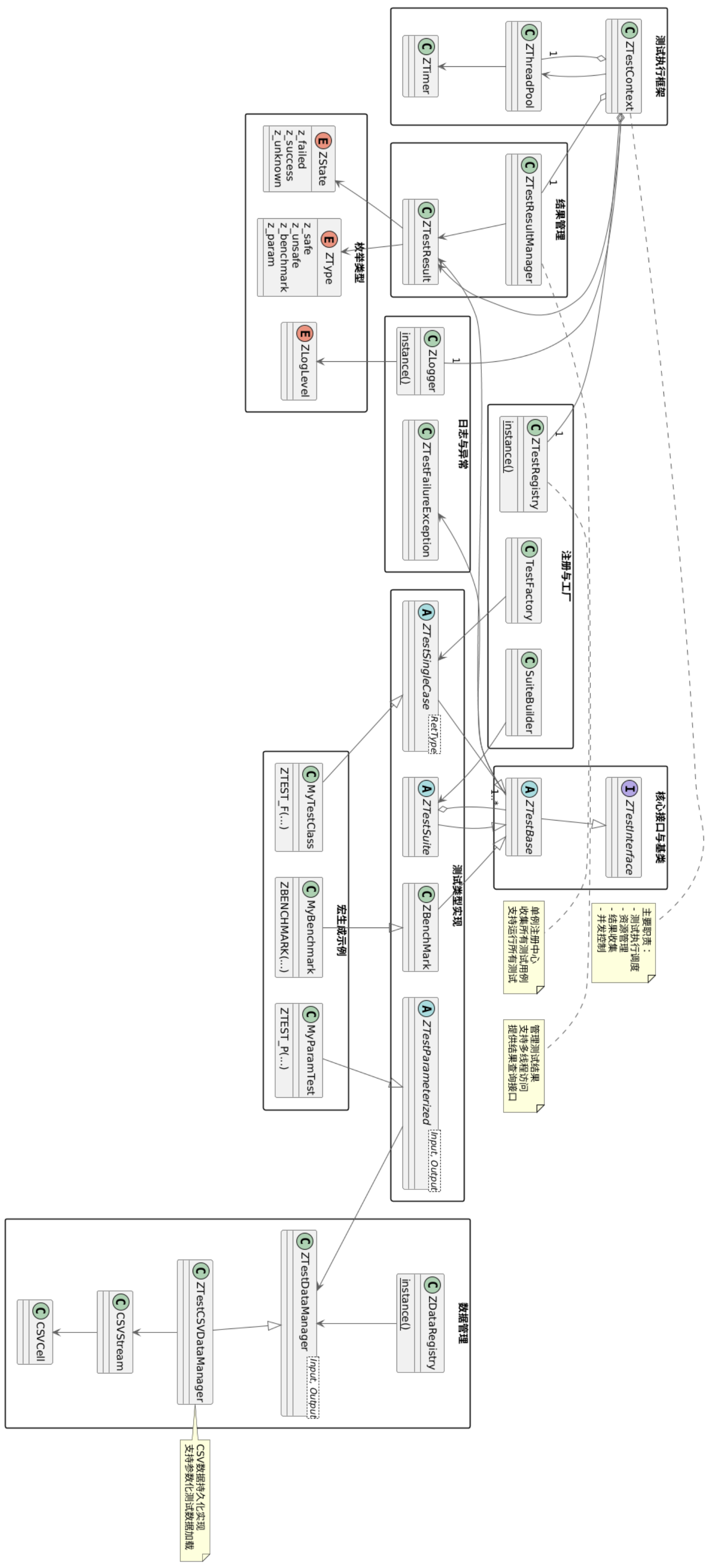


图 8: Ztest Class

3.3 Detailed Design

3.3.1 Test Definition Related Classes

- `ZTestInterface`: Defines the test execution framework interface.
- `ZTestBase` (ztest_base.hpp): An abstract base class for tests, encapsulating general properties and lifecycle hooks. It defines the test execution framework using the template method pattern, implements strategy pattern via virtual function tables, and extends using hook functions with decorator pattern characteristics.
- `ZTestSingleCase` (ztest_singlecase.hpp): Implements single-case tests, executes test logic using the template method pattern, and demonstrates factory pattern usage with TestFactory, supporting chained configuration with builder pattern features.
- `ZTestSuite`(ztest_suite.hpp): A test suite class inheriting from ZTestBase, implementing the template method pattern and supporting extension through hook functions.
- `ZBenchMark` (ztest_benchmark.hpp): A benchmark test class inheriting from ZTestBase and overriding the run() method to implement benchmarking through iterative test function execution.
- `ZTestParameterized` (ztest_parameterized.hpp): A base class for parameterized tests, encapsulating test data using the composite pattern and implementing a data-driven test framework through the run() method.
- `Macro Definition System` (ztest_macros.hpp): Implements syntax sugar via macros, automatically generates test classes using the factory pattern, and achieves automatic registration with name concatenation techniques.

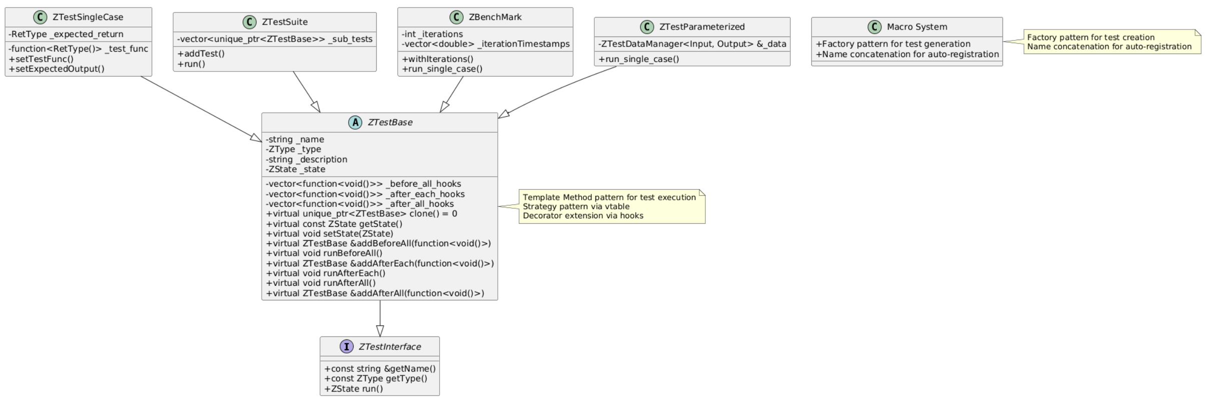

C ZTestSingleCase
-RetType _expected_return
-function<RetType()> _test_func
+setTestFunc()
+setExpectedOutput()
C ZTestSuite
-vector<unique_ptr<ZTestBase>> _sub_tests
+addTest()
+run()
C ZBenchMark
-int _iterations
-vector<double> _iterationTimestamps
+withIterations()
+run_single_case()
C ZTestParameterized
-ZTestDataManager<Input, Output> &_data
+run_single_case()
C Macro System
+Factory pattern for test generation
+Name concatenation for auto-registration
Factory pattern for test creation
Name concatenation for auto-registration
A 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
I ZTestInterface
+const string &getName()
+const ZType getType()
+ZState run()

图 9: Ztest Class

3.3.2 Test Registration Related Classes

- `ZTestRegistry` (ztest_registry.hpp): A globally accessible registry center implemented using the singleton pattern, managing test case collections and ensuring thread-safe registration and retrieval operations.

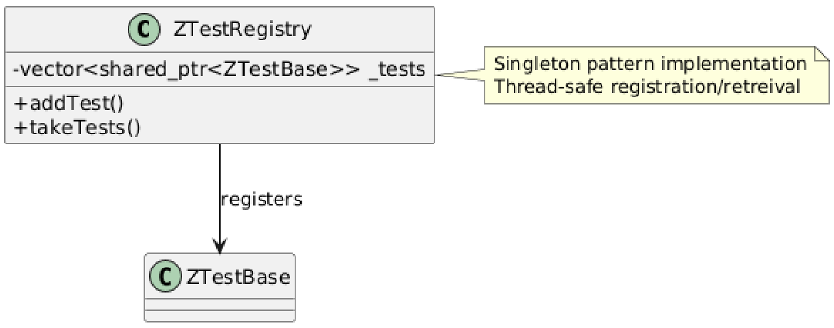

ZTestRegistry
-vector<shared_ptr<ZTestBase>> _tests
+addTest()
+takeTests()
Singleton pattern implementation
Thread-safe registration/retreival
registers
ZTestBase

图 10: Ztest Class

3.3.3 Test Execution Related Classes

- `ZTestContext` (ztest_context.hpp): A test execution context that selects execution strategies based on test types using the strategy pattern and implements parallel test execution via a thread pool.
- `ZThreadPool` (ztest_thread.hpp): A thread pool implementing the object pool pattern, managing tasks using the producer-consumer model, and achieving asynchronous execution monitoring through future/promise mechanisms.

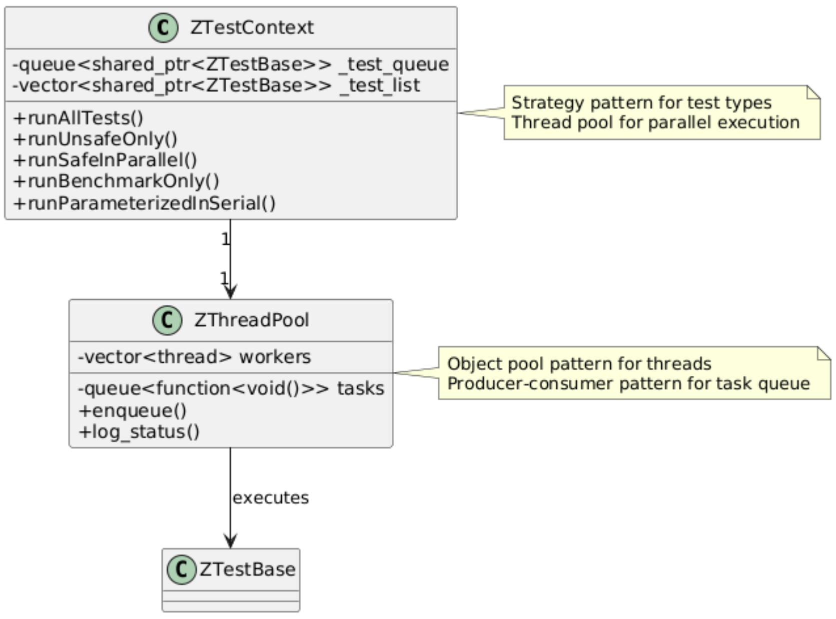

ZTestContext
-queue<shared_ptr<ZTestBase>> _test_queue
-vector<shared_ptr<ZTestBase>> _test_list
+runAllTests()
+runUnsafeOnly()
+runSafeInParallel()
+runBenchmarkOnly()
+runParameterizedInSerial()
Strategy pattern for test types
Thread pool for parallel execution
1
1
ZThreadPool
-vector<thread> workers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

图 11: Ztest Class

3.3.4 Data Management Related Classes

- `ZDataRegistry` (ztest_dataregistry.hpp): A data cache manager implemented using the singleton pattern for global access, with LRU strategy-based 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 implementing test data management for user-specified data types.
- `ZTestCSVDataManager`(ztest_parameterized.hpp): Inherits from `ZTestDataManager` and `ZDataManager`, implementing reading test data from CSV files and processing it into a format suitable for parameterized tests.

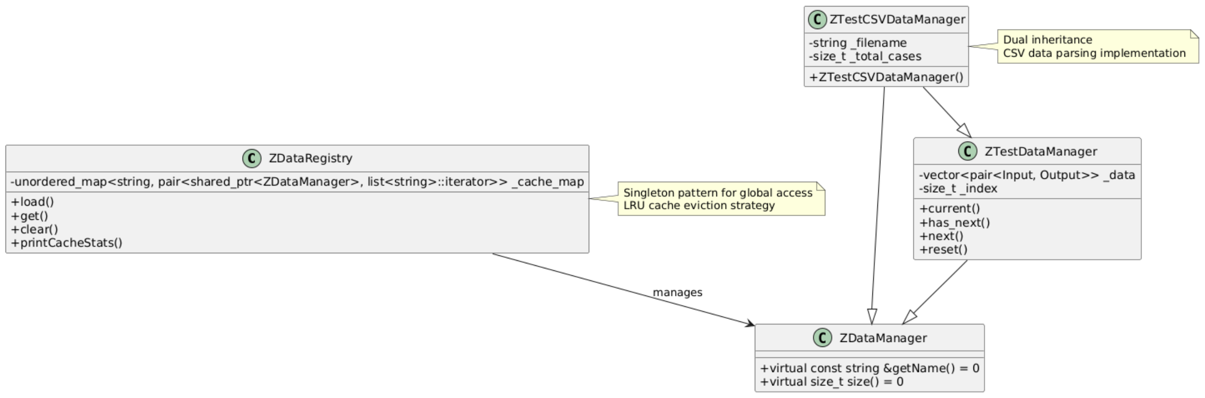

ZTestCSVDataManager
-string _filename
-size_t _total_cases
+ZTestCSVDataManager()
Dual inheritance
CSV data parsing implementation
ZDataRegistry
-unordered_map<string, pair<shared_ptr<ZDataManager>, list<string>::iterator>> _cache_map
+load()
+get()
+clear()
+printCacheStats()
Singleton pattern for global 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

3.3.5 Result Management Related Classes

- `ZTestResult` (ztest_result.hpp): A value object pattern-based test result class encapsulating immutable data such as test status and duration.
- `ZTestResultManager` (ztest_result.hpp): A result manager implemented using the singleton pattern, handling result storage and queries via the chain of responsibility pattern.

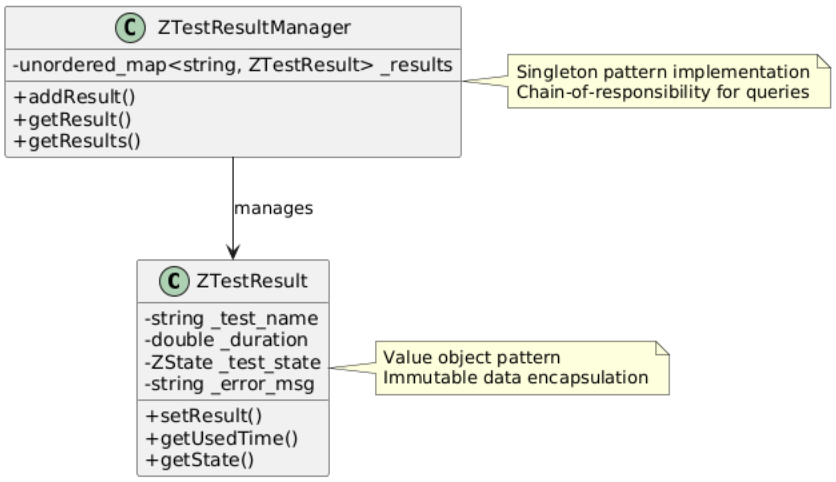


图 13: Ztest Class

3.3.6 Report Generation Related Classes

- `ZLogger` (ztest_logger.hpp): A multi-format report generator applying the template method pattern to define the report generation process and supporting HTML, JSON, and JUnit report formats.

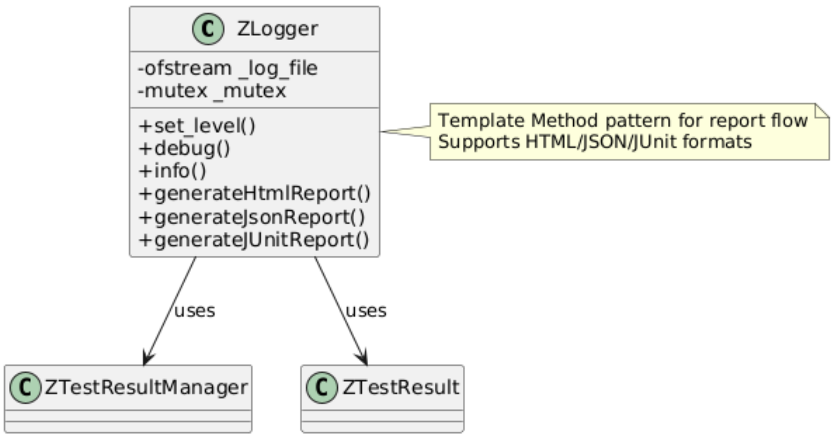


图 14: Ztest Class

3.3.7 Utility Module Related Classes

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

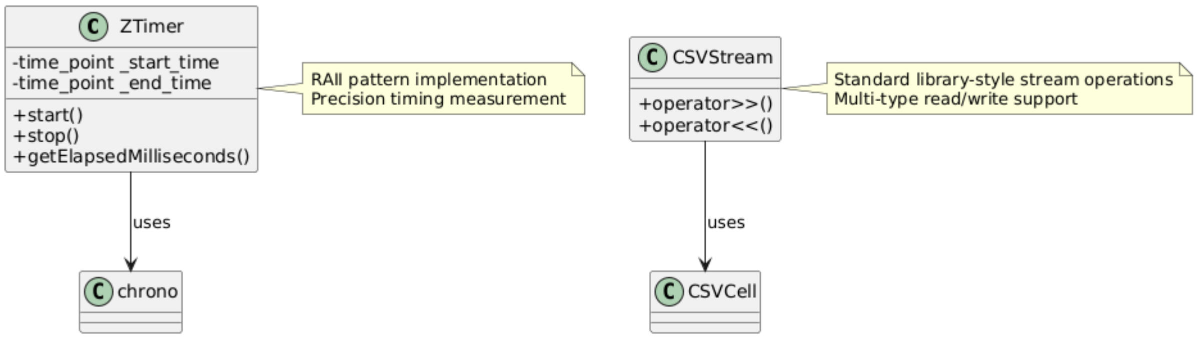

ZTimer
-time_point _start_time
-time_point _end_time
+start()
+stop()
+getElapsedMilliseconds()
RAII pattern implementation
Precision timing measurement
uses
chrono
CSVStream
+operator>>()
+operator<<()
Standard library-style stream operations
Multi-type read/write support
uses
CSVCell

图 15: Ztest Class

3.3.8 GUI Module Related Classes

- `ZTestModel` (gui.hpp): The Model layer in the MVC architecture, implementing the observer pattern to listen for test status changes.
- `ZTestController` (gui.hpp): The Controller layer in the MVC architecture, encapsulating test execution operations using the command pattern.
- `ZTestView` (gui.hpp): The View layer in the MVC architecture, employing the bridge pattern to separate interface elements from their implementation.

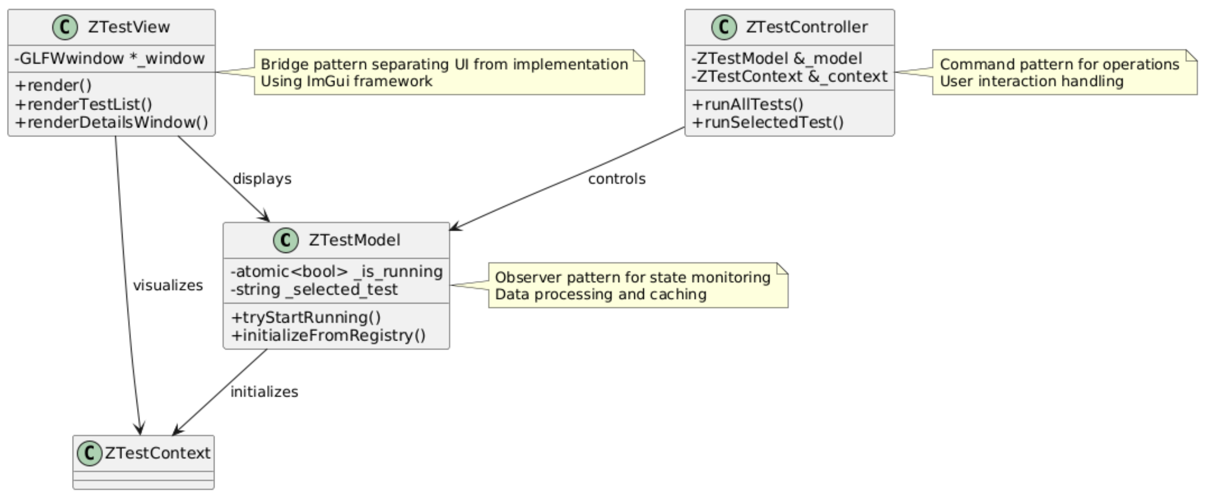


图 16: Ztest Class

4 Programming Progress

Task Phase	**Plan**
Theme Confirmation 2024.04.26-2024.05.03	Investigate themes and submit proposals.
Core Code Implementation 2024.05.03-2024.05.05	Implemented GUI and execution logic for Safe te
Major Performance Optimization 2024.05.05-2024.05.08	Thread pool optimization
Feature Optimization 2024.05.08-2024.05.11	Improved JSON, HTML output, JUnit format ou
Feature Optimization 2024.05.11-2024.05.13	Introduced imgui Docking functionality in GUI
Feature Optimization 2024.05.13-2024.05.15	Added CLI
Major Feature Optimization 2024.05.15-2024.05.24	Added BENCHMARK testing
Feature Optimization 2024.05.24-2024.05.25	Added device status monitoring and visualizatior
Major Feature Optimization 2024.05.25-2024.05.28	Added parameterized testing and data-driven sup
Major Performance Optimization 2024.05.28-2024.06.02	Implemented data caching and LRU cache evictic
Major Feature Optimization 2024.06.02-2024.06.07	Added AI diagnostics
Summary Work 2024.6.02-2024.06.08	Cross-platform migration & report writing

表 3: Programming Progress

5 Test Report

5.1 Function Testing & System Testing

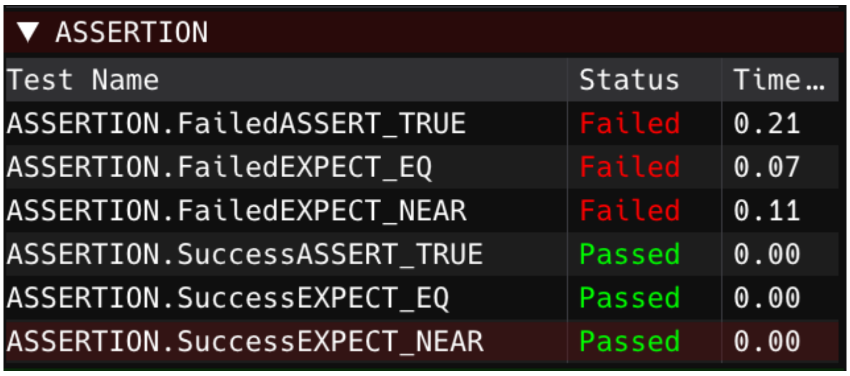
5.1.1 Assertion Function Testing

This test aims to verify the behavior of assertion macros such as EXPECT_EQ, EXPECT_NEAR, and ASSERT_TRUE in both successful and failed scenarios. By designing different test cases, it checks whether these assertions can correctly identify matches between expected and actual results, ensuring the reliability of the testing framework's assertion functionality.

```
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;
}
```



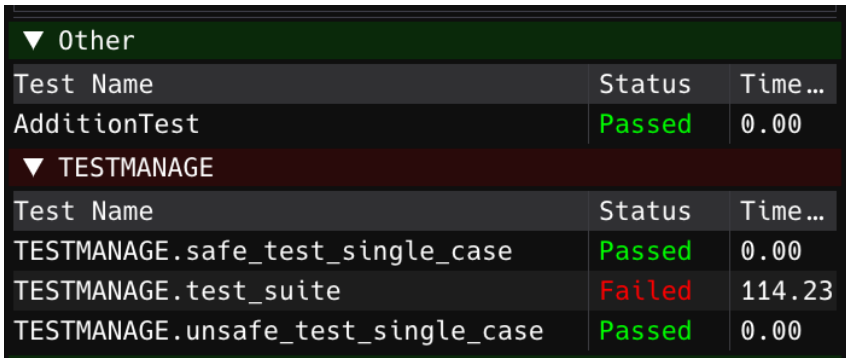
```
▼ ASSERTION
Test Name                         Status   Time…
ASSERTION.FailedASSERT_TRUE       Failed   0.21
ASSERTION.FailedEXPECT_EQ         Failed   0.07
ASSERTION.FailedEXPECT_NEAR       Failed   0.11
ASSERTION.SuccessASSERT_TRUE      Passed   0.00
ASSERTION.SuccessEXPECT_EQ        Passed   0.00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

图 17: Assertion Function Test

5.1.2 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 comprehensively covers different testing scenarios. Additionally, dynamic test case construction allows for flexible creation and registration of new test cases, enhancing the test framework's extensibility. During testing, memory allocation, function calls, and assertion checks ensure the correct execution of test cases and consistency with expected results. The test framework also provides pre and post hooks for necessary setup and cleanup operations before and after tests, ensuring test environment stability and result accuracy.

```
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
```

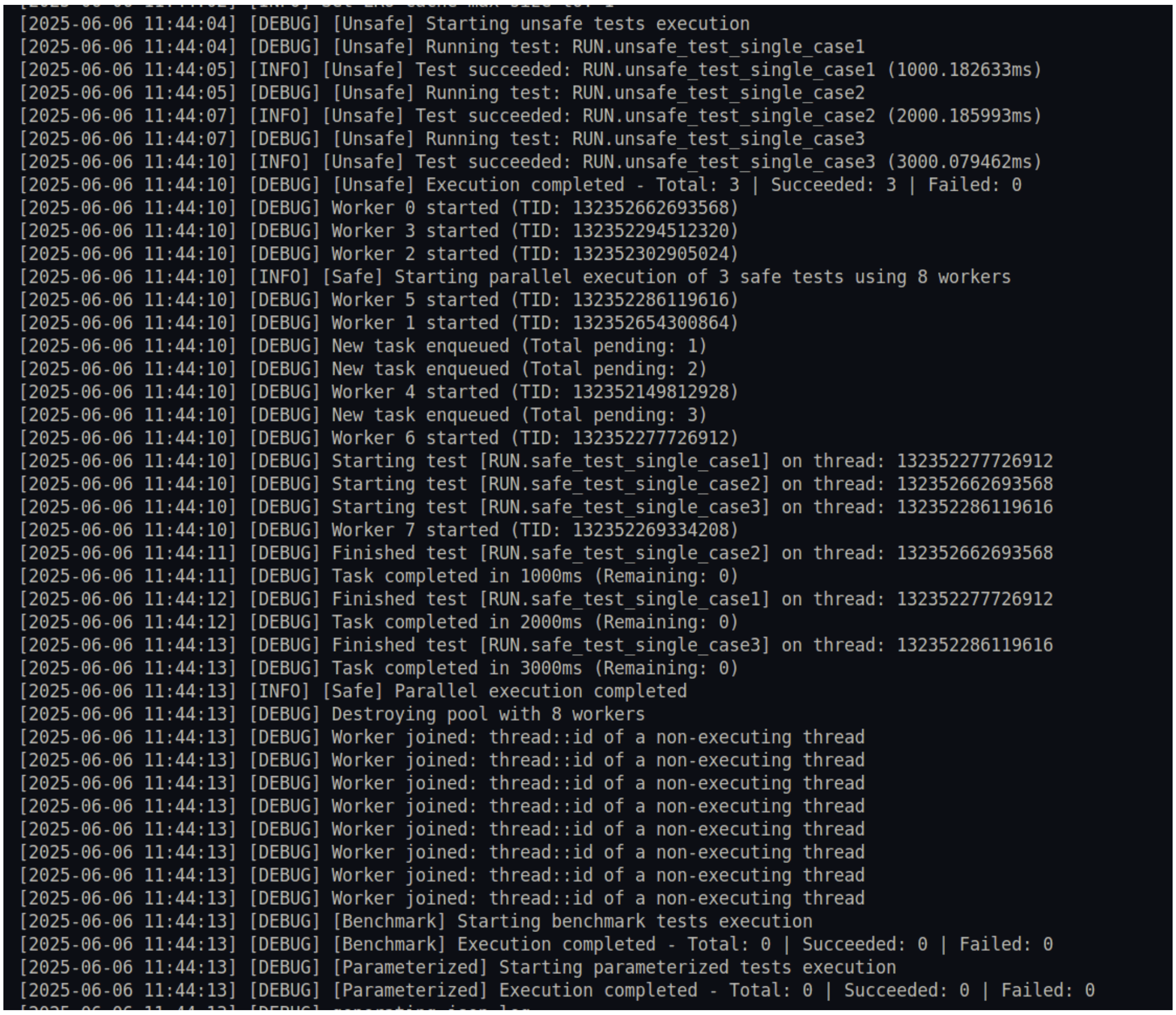
图 18: Test Management Function Test

5.1.3 Test Execution Management

This test module primarily verifies the efficiency and accuracy of the test execution management mechanism. Multiple safe and unsafe test cases are defined to simulate different execution times and task types. Each test case simulates varying execution times using the sleep function to evaluate the performance differences between parallel and sequential execution. Test results are interpreted 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;
}
```



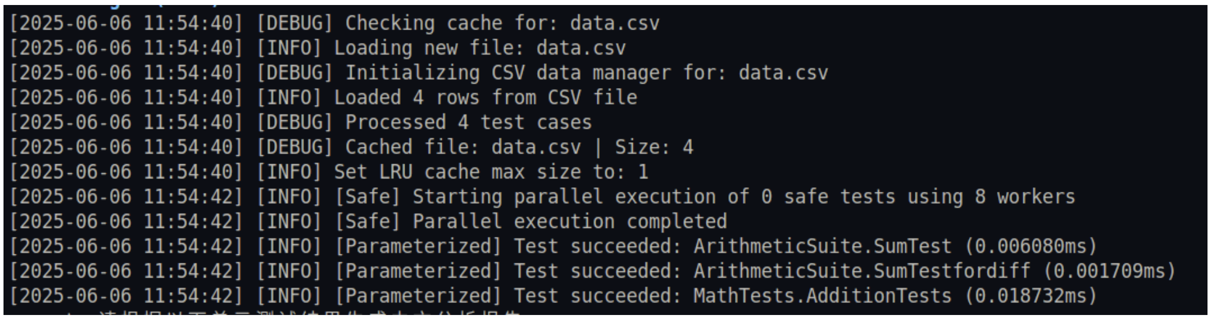
```
[2025-06-06 11:44:04] [DEBUG] [Unsafe] Starting unsafe tests execution
[2025-06-06 11:44:04] [DEBUG] [Unsafe] Running test: RUN.unsafe_test_single_case1
[2025-06-06 11:44:05] [INFO] [Unsafe] Test succeeded: RUN.unsafe_test_single_case1 (1000.182633ms)
[2025-06-06 11:44:05] [DEBUG] [Unsafe] Running 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

5.1.4 Data-Driven Testing

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

```
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;
}
```



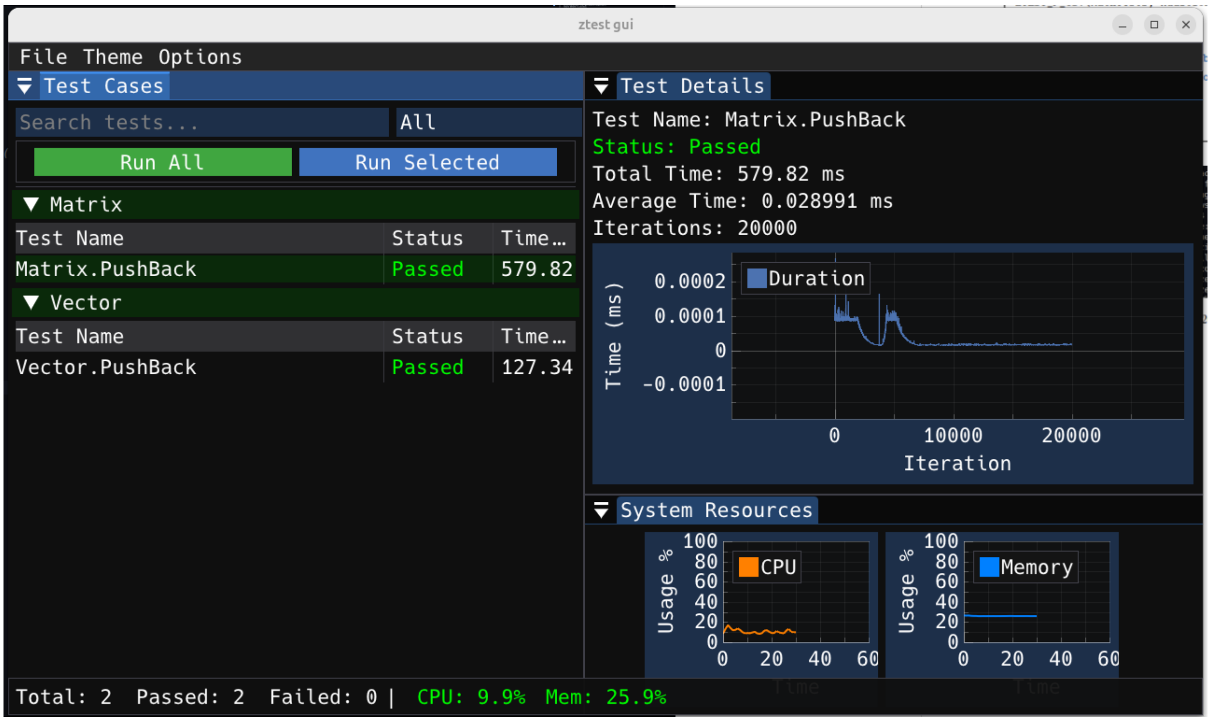
```
[2025-06-06 11:54:40] [DEBUG] Checking cache for: data.csv
[2025-06-06 11:54:40] [INFO] Loading new file: data.csv
[2025-06-06 11:54:40] [DEBUG] Initializing CSV data manager for: data.csv
[2025-06-06 11:54:40] [INFO] Loaded 4 rows from CSV file
[2025-06-06 11:54:40] [DEBUG] Processed 4 test cases
[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

5.1.5 Benchmark Testing

Testing the definition of benchmark-type tests, visualizing test time distribution, and monitoring CPU and memory usage in 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;
}
```


ztest gui

File Theme Options

Test Cases

Search tests... All

Run All Run Selected

▼ 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

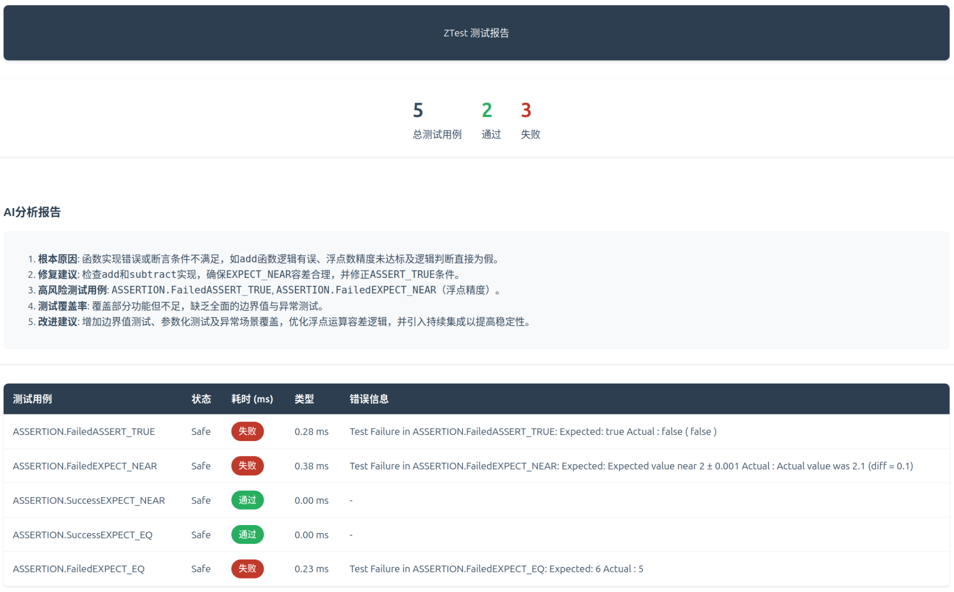
Total: 2 Passed: 2 Failed: 0 | CPU: 9.9% Mem: 25.9%

图 21: Data-Driven Function Test

-

5.1.6 Report Generation Testing

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


ZTest 测试报告

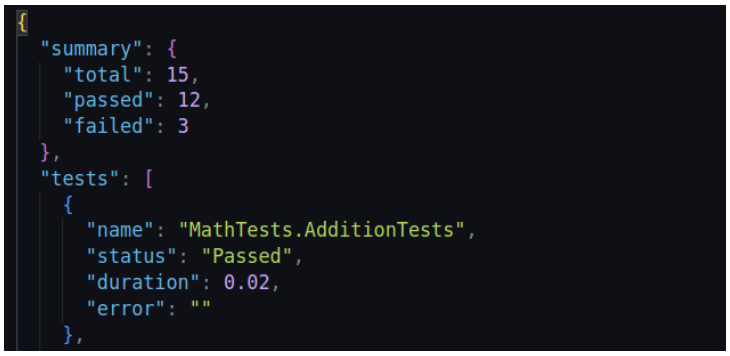
5 总测试用例 2 通过 3 失败

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。

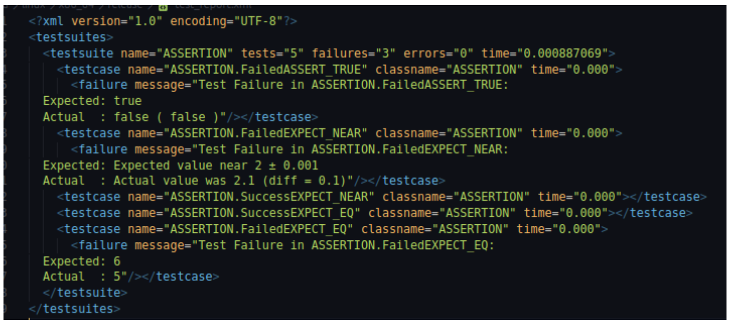
测试用例	状态	耗时 (ms)	类型	错误信息
ASSERTION.FailedASSERT_TRUE	Safe	失败	0.28 ms	Test Failure in ASSERTION.FailedASSERT_TRUE: Expected: true Actual : false (false)
ASSERTION.FailedEXPECT_NEAR	Safe	失败	0.38 ms	Test Failure in ASSERTION.FailedEXPECT_NEAR: Expected: Expected value near 2 ± 0.001 Actual : Actual value 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

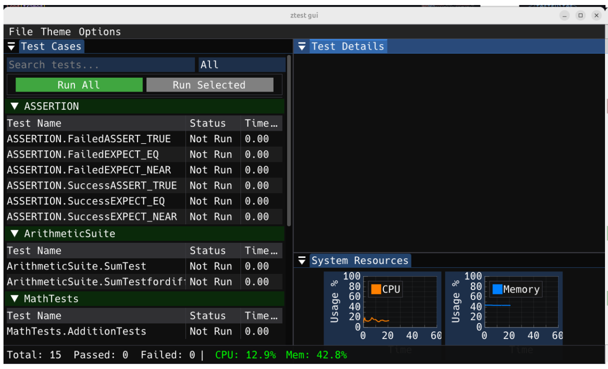


```xml
<?xml version="1.0" encoding="UTF-8"?>
<testsuites>
  <testsuite name="ASSERTION" tests="5" failures="3" errors="0" time="0.000887069">
    <testcase name="ASSERTION.FailedASSERT_TRUE" classname="ASSERTION" time="0.000">
      <failure message="Test Failure in ASSERTION.FailedASSERT_TRUE:
  Expected: 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

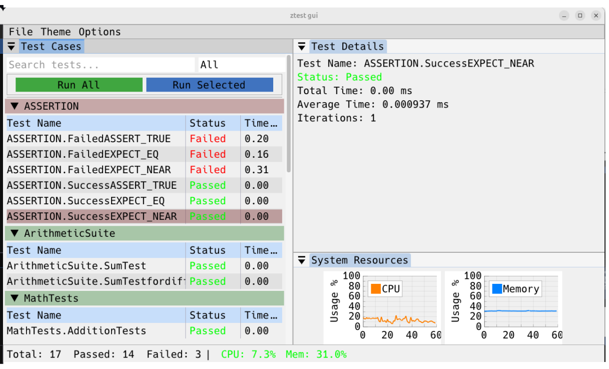
5.1.7 GUI Testing

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



```
▼ 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%
```

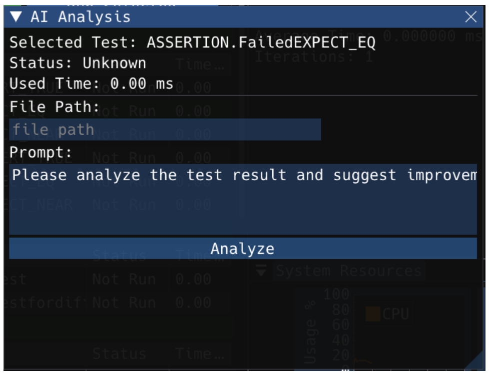
图 25: Dark Theme GUI Display



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

▼ 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%
```

图 26: Light Theme GUI Display

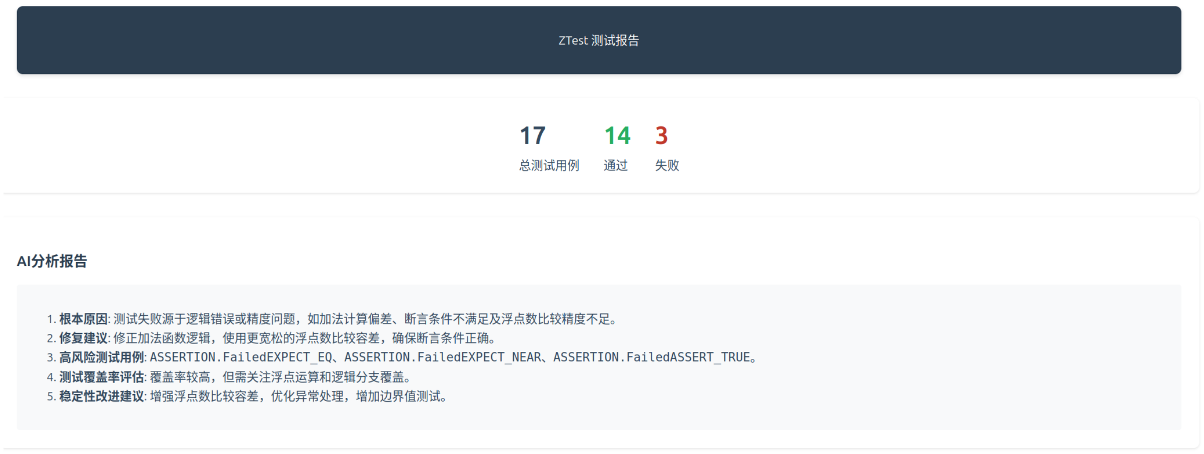


```
▼ AI Analysis
Selected Test: ASSERTION.FailedEXPECT_EQ
Status: Unknown
Used Time: 0.00 ms
File Path:
file path
Prompt:
Please analyze the test result and suggest improvem
Analyze
```

图 27: AI Diagnostic Assistant Display

5.1.8 AI Diagnosis Testing

During report generation, AI (qwen turbo) is invoked for diagnostics and integrated into the HTML report.



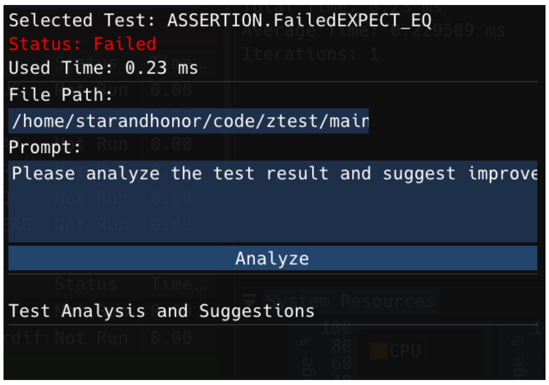
ZTest 测试报告

17	14	3
总测试用例	通过	失败

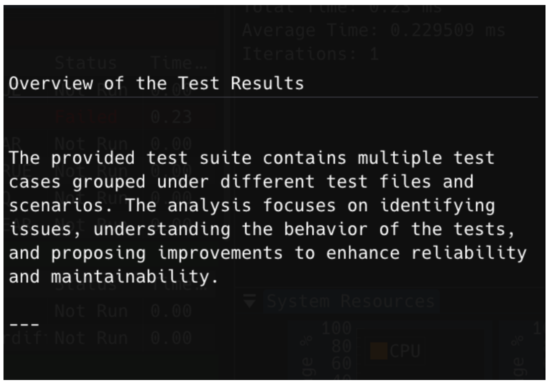
AI分析报告

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 to diagnose individual test cases and display results, allowing users to input custom prompts and file paths.



```
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
Analyze
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.

---
```

6 Personal Summaries

Name	Summary
Zheng Chenyang	During the test framework design and implementation process, I deeply appreciated the importance of modular design and layered architecture. By introducing the MVC pattern, I achieved a clear separation of models, views, and controllers, enhancing code maintainability and extensibility. In a multi-threaded environment, I optimized resource monitoring and task scheduling, ensuring system stability and performance. Additionally, by integrating AI analysis capabilities, I provided intelligent feedback for test results, enhancing the tool's utility. The development process strengthened my technical expertise and made me more mindful of teamwork and communication efficiency. In the future, I will continue to iterate, refine features, and improve user experience.
Ye Suohua	In this program, I was mainly responsible for the frontend part, choosing the imgui library for frontend development and successfully completing the frontend architecture and rendering. I realized that frontend structures need to be designed based on user requirements, considering factors such as cross-platform compatibility and library version issues.
Qi Yansong	Participating in the development of the "ztest" unit testing framework, I felt that the knowledge I learned could be applied not only to game development but also to tool software design. As a lightweight and easy-to-use C++ testing framework, ztest addresses the shortcomings of existing frameworks like Google Test in terms of GUI support, concurrent testing, and reporting systems. Through innovative design, it enhances user experience and extensibility. I particularly appreciated the team's implementation of multiple test case creation methods, which reflect the "user-friendly" design philosophy. The design of multithreaded safety evaluations made me aware of the complexity of concurrency issues and the importance of categorized handling. Through practice, I gained a deeper understanding of modern C++ features and design patterns. This experience not only improved my technical skills but also made me recognize that excellent software products need to balance technical depth with user experience. Efficient team communication and collaboration are key to achieving this, and I look forward to continuing to explore challenges and solutions in engineering practices.
Wang Ruizhen	Through this project, I have grown a lot. It was a large-scale project with a high degree of complexity, posing a significant challenge for me. Due to time constraints, I had to learn on the job. Although it was rushed and I only had a superficial understanding of many aspects, I also gained a wealth of knowledge, mastered new skills, and honed my problem-solving abilities.
Wu Hongqing	In the zTest project, I was responsible for GUI design. On the technical side, I gained practical experience in applying the MVC architecture, utilized the ImGui framework to create a visual interface, designed a simple visualization window, and coordinated the ZTestModel, ZTestView, and ZTestController classes to handle data visualization, thereby enhancing user-friendliness. In terms of teamwork, I improved my collaboration skills by working with the team to define interfaces and advance the project. I also standardized the implementation of different classes within the project to prevent compilation

7 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