ztest: A Fast, User-Friendly, Lightweight C++ Unit Testing Framework

April 27, 2025

Contents

1 Requirement Analysis for System

1.1 The Background and Motivation of System

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 understanding the dynamic collaboration between classes and objects, but also the ability to solve architectural challenges using design patterns and abstract thinking. 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 (with a graphical user interface, GUI) testing tool. The goal is to provide an intuitive and user-friendly environment for developers and testers to write, run, and manage test cases. The tool will support various types of tests (such as unit tests and integration tests) and provide detailed test result reports.

Table 1: Comparison of Mainstream Testing Frameworks

Framework	**GUI Support**	**Concurrency**	**Reporting**	**Extensibility**
Google Test (C++)	None	Limited	Basic	Medium
JUnit (Java)	Eclipse Plugin	Supported	HTML/XML	High
PyTest (Python)	Third-Party Tools	Excellent	Rich	Excellent
Catch2 (C++)	None	Average	Simple	Medium
Ours (C++)	Yes	Excellent	Rich	High

1.2 System Objectives

The design goal of this testing framework is to provide a flexible, efficient, and easy-to-use (with a graphical user interface, GUI) testing tool that supports test case management, assertion validation, test execution, and result reporting. The following sections detail its features in terms of test case management, assertion mechanisms, test executor, and result reporting.

1.2.1 Test Case Management

Test case management is one of the core functions of the testing framework, supporting the definition, registration, and organization of test cases. Test cases can be defined through inheritance from a base class or using a factory pattern, and support for setting up pre-test and post-test hook functions is provided.

Creating Individual Test Cases Three methods for creating test cases have been implemented: chain creation, macro definition, and inheritance definition:

1. **Chain Creation**: Through chain creation, customized test cases can be achieved. In network programming tests, using chain creation aligns more logically with the requirements. An example is provided below:

```
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. **Macro Definition**: Through macro definition, customized test cases can be achieved. Macro definitions greatly simplify the syntax, making test case definitions clearer and more structured. The actual implementation is inheritance definition + automatic registration.

```
ZTEST_F(BasicMath, FailedAdditionTest) {
    EXPECT_EQ(6, add(2, 3));
    ASSERT_TRUE(6==add(2, 3));
    return Zstate::Z_SUCCESS;// If this point is reached, the test is successful
}
```

3. **Inheritance Definition**: By inheriting from the test base class, customized test cases can be achieved with higher extensibility, allowing for the definition of more custom test methods.

```
class MathTests_Addition : public ZtestBase {
    public:
        MathTests_Addition() : ZtestBase("MathTests.Addition", ZType::ZSAFE, "Test
            addition function") {}
        Zstate run() override {
               EXPECT_EQ(5, add(2, 3)); // Expected result is 5
               EXPECT_EQ(0, add(0, 0)); // Expected result is 0
               return Zstate::Z_SUCCESS;
        }
};
```

Test Case Registration In the case of macro definitions, the **ZTestRegistry** test registration center can be used to dynamically register test cases. For inheritance from the test base class, due to the higher degree of freedom given to users, for safety reasons, users are required to register manually. For chain-created test cases, registration can be achieved by calling the registration function after build, or by manual addition. Here is an example of using the test registration center class:

```
// Manual registration example
ZTestRegistry::getInstance().registerTest(std::make_shared<MathTests_Addition>());
```

Building Test Suites Multiple test cases can be organized through the **ZTestSuite** class, supporting batch execution and multi-threaded testing. It also records the time required to run multiple test cases, the number of test cases passed, and other metrics.

```
auto mathSuite = std::make_unique<ZTestSuite>("Math", ZType::Z_SAFE,"math test");
// Add individual test cases to the suite
mathSuite->addTest(TestFactory::createTest("Addition", ZType::Z_SAFE, "", add, 2, 3).
    setExpectedOutput(5).build());
mathSuite->addTest(TestFactory::createTest("Subtraction", ZType::Z_SAFE, "", subtract, 5,
     3).setExpectedOutput(2).build());
```

1.2.2 Assertion Mechanisms

Assertion mechanisms are used to validate 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 validate conditions within tests. The main assertions provided are:

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

Usage examples are as follows:

```
// 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** exception is thrown. Additionally, by inheriting from the **ZTestFailureException** exception handling function, custom exception handling logic can be defined.

1.2.3 Test Executor

The test executor is responsible for managing the execution of test cases, supporting **multi-threaded** parallel execution of test cases, and collecting test results. It implements automatic scheduling of tests and statistical analysis of test results.

The **ZTestContext** class is used for test context management, responsible for maintaining the test case queue and executing tests in parallel using multiple threads.

```
// Define a test context object
ZTestContext context;
// Add test cases to the test queue
for (auto &&test : registeredTests)
    context.addTest(std::move(test));
// Execute test cases in parallel using multiple threads
context.runAll();
```

Two types of test cases can actually be defined: **z_safe** and **z_unsafe**. These two types correspond to two testing modes. **z_safe** test cases are thread-safe, while **z_unsafe** test cases are not thread-safe. The default type is **z_safe**. If **z_unsafe** types appear during testing, the framework will wait for their execution to complete before running **z_safe** test cases.

1.2.4 Result Reporting

After testing is completed, the framework generates a detailed test result report, including test names, execution time, pass/fail status, and error information. We define three states for test results:

- **z_success**: Test passed.
- **z_failed**: Test failed.
- **z_unknown**: Test not completed.

Logging Test results are output through the **ZLogger** class to the console or **a file**. The output rules are as follows: [info] indicates test information, [FAILED] indicates test failure, and [OK] indicates test success. An example output is as follows:

```
[    OK    ] BasicMath.NegativeTest (0 ms)
[  info  ] Preparing environment for addition test...
[    OK    ] Advanced.Multiplication (0 ms)
[ FAILED ] BasicMath.FailedSubtractionTest (1 ms)
Error: Test Failure in BasicMath.FailedSubtractionTest:
  Expected: 3
  Actual : 2
[ FAILED ] BasicMath (1 ms)
```

The **ZTestResult** class also saves information such as test execution time, pass/fail status, and error messages to store test results.

1.2.5 GUI Display

Test results can be displayed through a GUI. Future plans include adding functionality to the GUI interface for adding, deleting, modifying test cases, and exporting test reports. The general interface design is shown in the figure below.

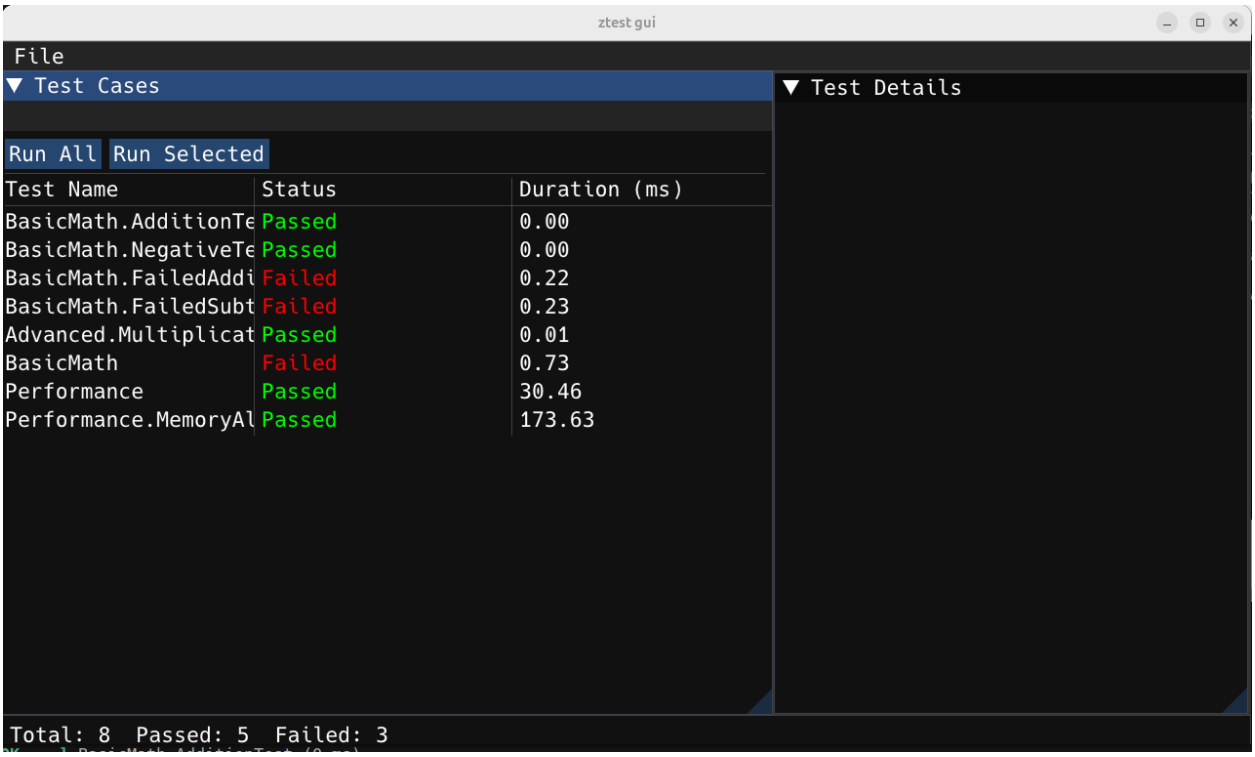

Figure 1: Test Management Interface Layout

2 Overall Architecture

The core module loads test cases and functions to be tested through a plugin mode. The core module is divided into four parts.

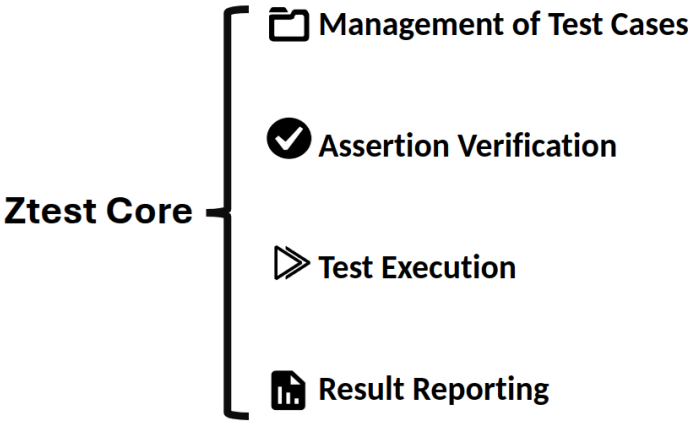

Figure 2: ztest core architecture

The GUI architecture uses the MVC (Model-View-Controller) pattern. The View focuses on the UI that users see and provides interaction. The Controller acts on the model and view. It controls the flow of data to model objects and updates the view when data changes. It separates the view from the model. The Model mainly implements the modeling of underlying files.

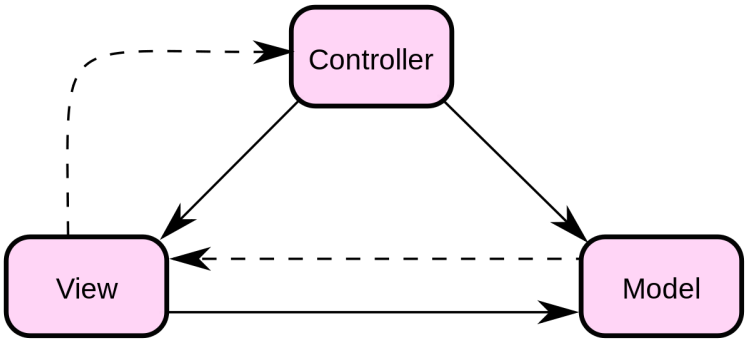

Figure 3: ztest gui architecture

3 Program Analysis

3.1 Technical Details

3.1.1 Testing Framework Design

- **ZtestInterface**: Defines the basic interface for test cases, including methods for obtaining test names, running tests, and getting test types.
- **ZtestBase**: Serves as the base class for test cases, implementing basic attributes and methods for tests, such as test names, test types, description information, pre-test hook functions (beforeAll), and post-test hook functions (afterEach).

- **ZtestSingleCase**: Inherits from **ZtestBase** and is used to define individual test cases. It supports setting test functions, expected results, and description information, and implements the logic for executing tests.
- **ZTestSuite**: Used to organize multiple test cases, supporting batch execution of tests and statistical analysis of test results (passed, failed, total time taken, etc.).
- **ZTestRegistry**: Implemented using the Singleton pattern, this test registration center is responsible for registering and managing test cases, supporting dynamic addition of test cases.
- **ZTestContext**: The test context management class is responsible for maintaining the test case queue and executing test cases in parallel using multiple threads, while also collecting and managing test results.

3.1.2 Test Case Construction and Execution

The construction of test cases is implemented through the **TestBuilder** class, supporting chain calls to set test functions, expected results, pre-test hooks, and post-test hooks. The **TestFactory** class provides convenient methods for creating test cases.

During test execution, the **ZTestContext** class retrieves test cases from the test queue and uses the **ZTimer** class to time the tests, recording the start time, end time, and duration of each test. Test results are encapsulated in the **ZTestResult** class, and detailed information about test results is output to the console.

3.1.3 Assertions and Exception Handling

The framework provides assertion macros **EXPECT_EQ** and **ASSERT_TRUE** to compare expected values with actual values or to verify whether conditions are true. If an assertion fails, a **ZTestFailureException** exception is thrown, and error information is recorded.

3.1.4 Test Suites and Multi-threaded Execution

Test suites (**ZTestSuite**) can contain multiple test cases and support batch execution. The test context (**ZTestContext**) executes test cases in parallel using multiple threads, fully utilizing the computing power of multi-core CPUs to improve testing efficiency.

3.1.5 Logging Management

The framework provides logging functionality through the **ZLogger** class, supporting thread-safe log output. Log information includes test results, error information, and information about pre-test and post-test operations.

3.1.6 UI Implementation

This system uses the MVC architecture to implement the graphical user interface, constructing a visual test management interface using the Dear ImGui framework. The core components of the UI module include model management, view rendering, and controller interaction, implemented as follows:

Model Management Class (ZTestModel) **ZTestModel** maintains the state information of test cases (name, status, duration, error messages), providing thread-safe data update interfaces `updateFromContext()`, tracking the currently selected test case `_selected_test`, and monitoring the test running status `_is_running` and progress `_progress`.

View Rendering Class (ZTestView) **ZTestView** is responsible for rendering the user interface, including the main menu, test list window, and details window. The main menu rendering (renderMainMenu) implements the file menu exit functionality. The test list window (renderTestList) uses a three-column layout to display test names, statuses, and durations, supports test case selection interactions, and provides "Run All" and "Run Selected" operation buttons. The details window (renderDetailsWindow) displays detailed information about the selected test, including multi-line text display of error messages, and uses color coding (green for success, red for failure) to indicate status.

Controller Class (ZTestController) **ZTestController** implements test execution thread management, providing `runAllTests()` and `runSelectedTest()` methods, handling synchronization between multi-threading and model status.

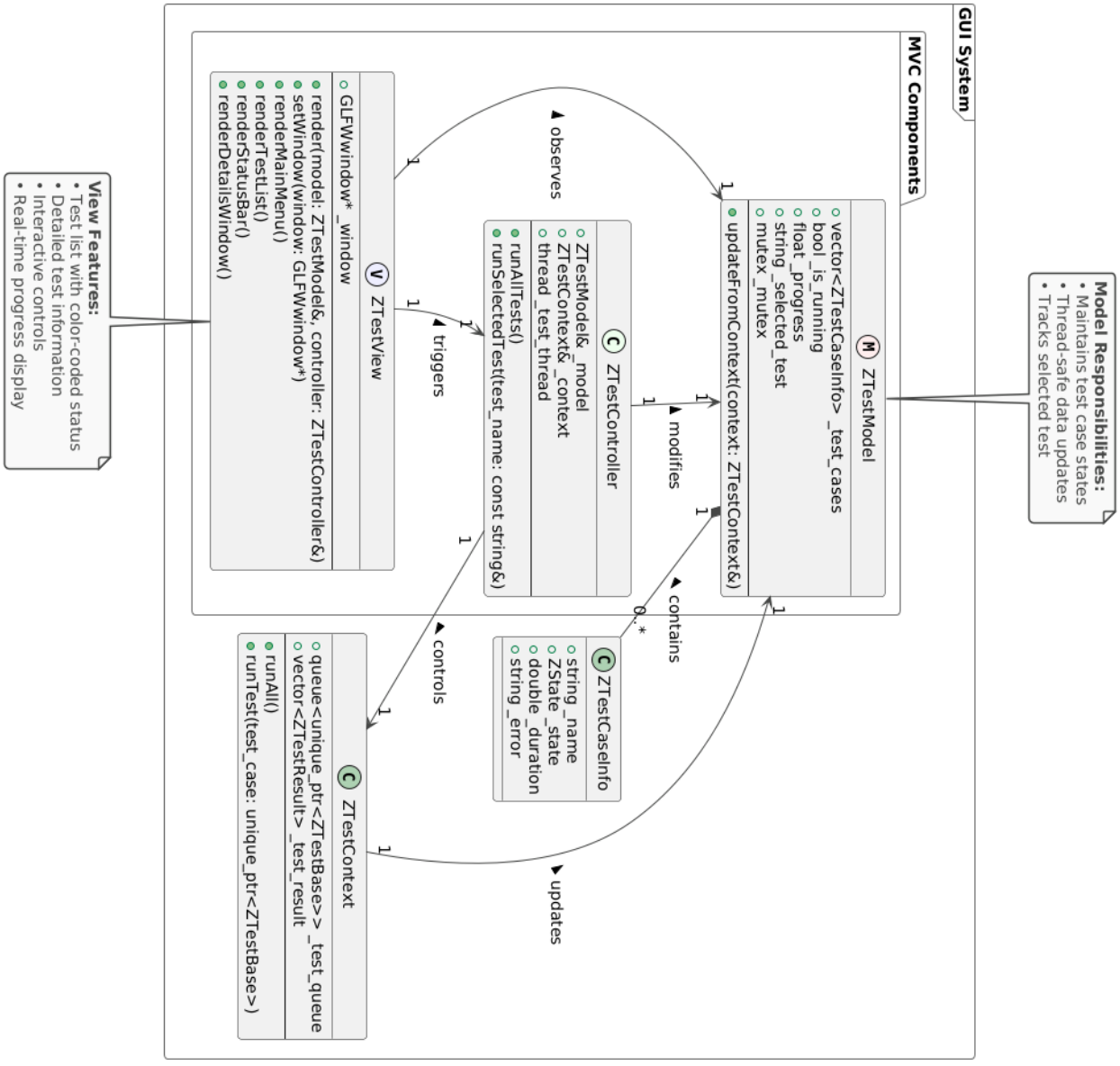

Figure 4: GUI class diagram

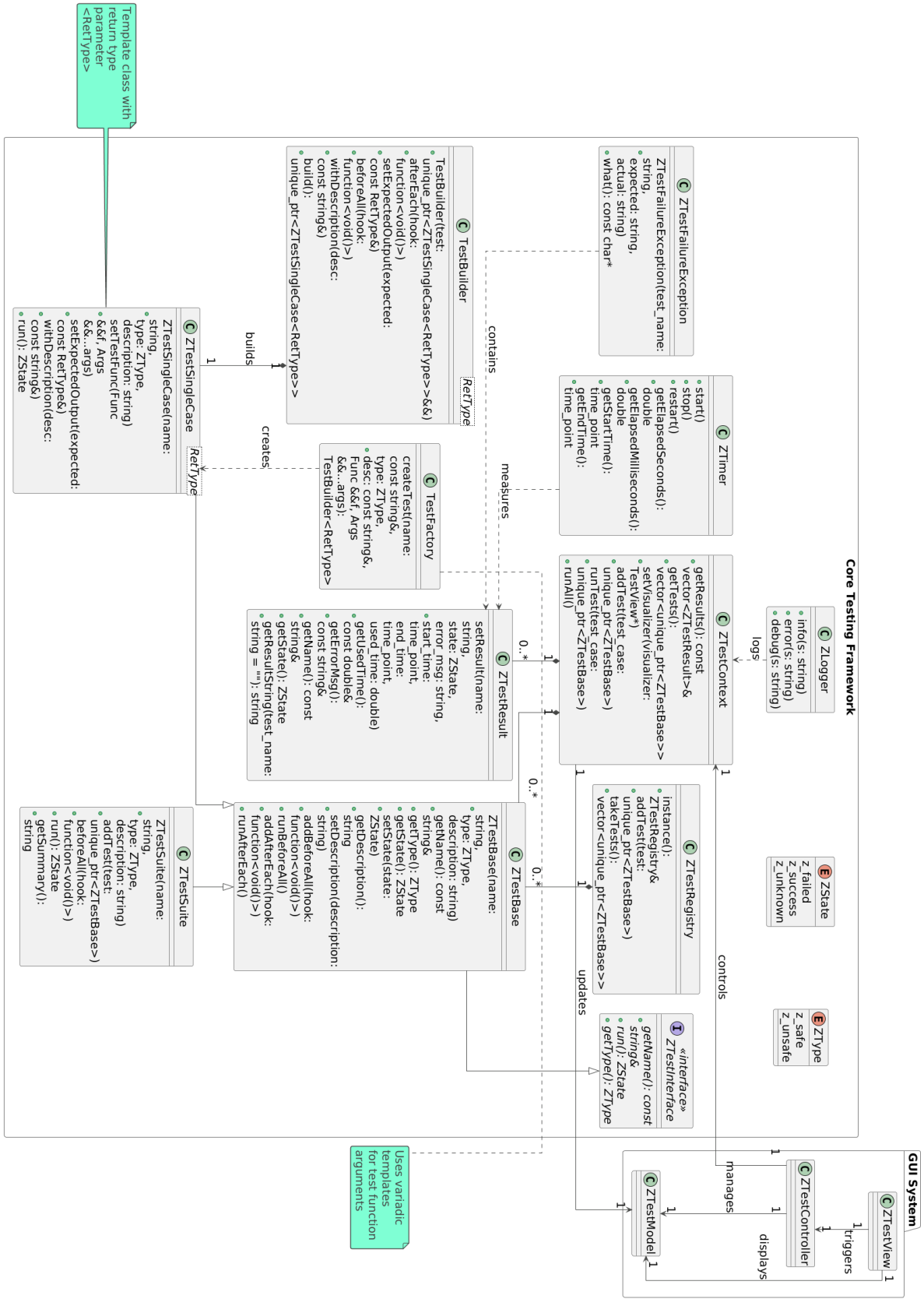

Figure 5: UML class diagram

3.2 Key Issues for System

3.2.1 Automatic Type Inference for Functions to be Tested

When using **ZtestSingleCase**, users only need to provide the function name, parameters, and expected result, and **ZtestSingleCase** will take over the specific logic of running the test and comparing the test results. This requires us to implement automatic inference of the parameter types and return type of the function to be tested when constructing **ZtestSingleCase**, so that the function can be called for testing and result verification. This is mainly achieved using templates and closures.

3.2.2 Separation and Dynamic Loading of Files to be Tested and Test Files

In simple tests, not separating the files to be tested and test files may not seem like a big issue. However, as the size of the project being tested increases and the number of test cases grows, maintaining and modifying the code will become very cumbersome if they are not separated. As a testing framework, we need to provide the ability to separate test files and files to be tested, placing test files in the **test** folder and files to be tested in the **src** folder.

We designed our GUI testing framework to act as the main program, while the user's code to be tested and test cases act as plugins. We need to provide an interface **ZTestAPI** to allow the main program to call functions in the plugins (functions in both test files and files to be tested).

Figure 6: Directory File Structure

```
// Example of exporting a function to be tested
ZTestAPI int subtract(int a, int b) { return a - b; }
```

3.2.3 Implementation of Multiple Construction Methods

Implementing multiple test case construction methods is actually aimed at providing a better user experience. In terms of implementation, chain creation mainly uses the Builder design pattern, individual test case creation mainly uses the Factory pattern, and macro definition mainly expands to inheritance definition. The implementation of multiple syntaxes facilitates users, but how to manage these syntax-constructed classes uniformly is a challenge.

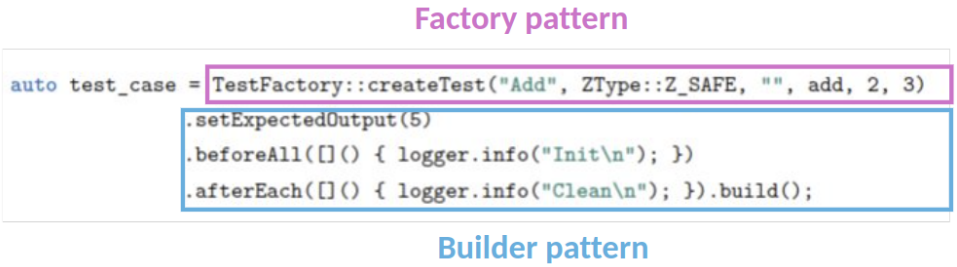

Figure 7: Implementing Chain Creation

3.2.4 Safe Multi-threaded Evaluation

Not all functions are thread-safe. Therefore, we need to categorize the functions to be tested and perform thread-safe evaluation accordingly. There are actually two types of test cases that can be defined: **z_safe** and **z_unsafe**. These two types correspond to two testing modes. **z_safe** test cases are thread-safe, while **z_unsafe** test cases are not thread-safe. The default type is **z_safe**.

If **z_unsafe** types appear during testing, the framework will wait for their execution to complete before running **z_safe** test cases.

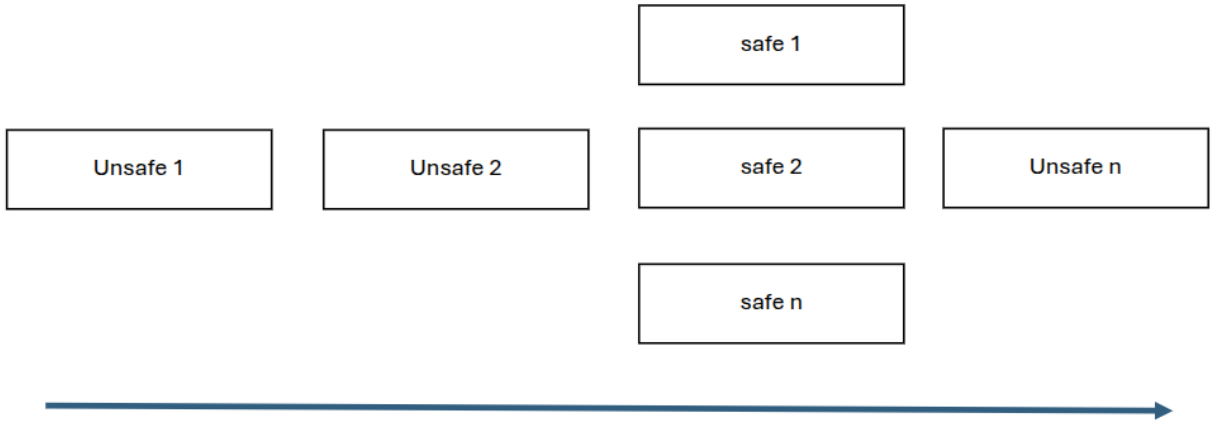

Figure 8: Implementing Safe Multi-threaded Execution

3.2.5 Test Result Data Storage

After using multi-threaded evaluation, the order of each evaluation may not be the same each time. However, we do not want the results of each evaluation to be displayed in a different order on the GUI. Additionally, we need to store historical test results to enhance user experience. From a design perspective, we should add an intermediate layer, more precisely, we may need to add a database-like module to store test results for use by **ZTestModel**.

3.2.6 Pointer Management

Considering that this framework uses a large number of pointers, memory leaks and other issues can easily occur. Therefore, we decided to use smart pointers in C++11 to manage memory and improve the reliability of the system.

3.3 Duty Assignment

Table 2: Assignment Table

Functional Module	**Responsible Person**
GUI Implementation	Wu Hongqing
Assertion Mechanism	Wang Ruiqing
Test Case Management	Zheng Chenyang, Ye Suohua
Test Case Execution	Zheng Chenyang
Report Result Generation	Qi Yansong

GUI Implementation

- Developed a visual interface based on the Dear ImGUI framework.
- Implemented a three-column layout for the test list (name/status/duration).
- Developed progress bar components and real-time status bar updates.
- Implemented error message rendering for the test details window.

Assertion Mechanism

- Designed type-safe templated assertion macros (EXPECT_EQ/ASSERT_TRUE).
- Implemented colored error output.
- Developed an extensible exception handling framework (ZTestFailureException).
- Supported the registration mechanism for custom comparison operators.

Test Case Management

- Built a chained API (TestBuilder) to implement a fluent interface.
- Developed a macro expansion system (ZTEST_F) for automatic registration.
- Designed a tree-shaped organizational structure for test suites (ZTestSuite).
- Implemented dependency injection mechanisms for pre-test and post-test hooks.
- Developed a plugin-based test loading system (dynamic library integration).

Test Case Execution

- Built a thread pool scheduler (ZTestContext).
- Implemented isolated execution strategies for safe and unsafe tests.
- Designed a test priority queue with timeout termination mechanisms.

Report Result Generation

- Designed a structured result storage format (ZTestResult).
- Implemented ANSI escape code-based colored console output.
- Built a version comparison system for historical test results.
- Integrated data visualization components (chart generation).

References

[1] GoogleTest - Google Testing and Mocking Framework. https://github.com/google/googletest. Accessed: 2025-04-27.

[2] JUnit 5 - The 5th major version of the programmer-friendly testing framework for Java and the JVM. https://github.com/junit-team/junit5. Accessed: 2025-04-27.

[3] The pytest framework makes it easy to write small tests, yet scales to support complex functional testing. https://github.com/pytest-dev/pytest. Accessed: 2025-04-27.

[4] Catch2 - A modern, C++-native, test framework for unit-tests, TDD and BDD. https://github.com/catchorg/Catch2. Accessed: 2025-04-27.

[5] GoogleTest User's Guide. https://google.github.io/googletest/. Accessed: 2025-04-27.