ztest: 一个快捷，易用，轻量化的 C++ 单元测试框架

2025 年 6 月 4 日

目录

1 需求分析

随着复杂系统架构设计能力的培养需求日益凸显，掌握面向对象设计原则与模式化工程实践已成为高级软件工程教育的核心目标。这不仅需要理解类与对象间的动态协作关系，更要具备通过设计模式解决架构难题的抽象思维能力。现有的单元测试框架 (如 Google Test) 具有上手难度较高，并发支持有限，报告系统过于简洁，可拓展性一般等缺点，我们团队计划开发一个提供一个灵活、高效且易于使用 (带图形用户界面 GUI) 的测试工具，旨在提供一个直观、易用的环境，方便开发人员和测试人员编写、运行和管理测试用例。该工具将支持多种测试类型（如单元测试、集成测试等），并提供详细的测试结果报告。

表 1: 主流测试框架对比

框架	**GUI 支持**	**并发测试**	**报告系统**	**扩展性**
Google Test(C++)	无	有限	基础	中等
JUnit (Java)	Eclipse 插件	支持	HTML/XML	高
PyTest (Python)	第三方工具	优秀	丰富	优秀
Catch2 (C++)	无	一般	简洁	中等
Ours (C++)	有	优秀	丰富	高

2 系统功能

本测试框架的设计目标是提供一个灵活、高效且易于使用 (带图形用户界面 GUI) 的测试工具，支持测试用例的管理、断言验证、测试执行以及结果报告。以下将从测试用例管理、断言机制、测试执行器和结果报告四个方面详细介绍其功能。

2.1 测试用例管理

测试用例管理是测试框架的核心功能之一，支持测试用例的定义、注册和组织。测试用例可以通过继承基类或使用工厂模式创建，并支持设置前置和后置钩子函数。

2.1.1 单个测试用例的创建

实现了三种方式创建测试用例, 分别为链式创建, 宏定义, 继承定义:

1. **链式创建**: 通过链式创建，实现测试用例的自定义, 在网络编程的测试中，使用链式创建的方式创建测试用例会更加符合逻辑。例子如下

```
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. **宏定义**：通过宏定义，实现测试用例的自定义。宏定义极大地简化了语法，使得测试用例的定义更加清晰和结构化。展开之后实际上是继承定义 + 自动注册。

```
ZTEST_F(BasicMath, FailedAdditionTest) {
    EXPECT_EQ(6, add(2, 3));
    ASSERT_TRUE(6==add(2, 3));
    return Zstate::Z_SUCCESS;//如果能运行要此处，说明测试成功
}
```

3. **继承定义**：通过继承测试基类，实现测试用例的自定义，可拓展性更高，可以定义更多自定义的测试方法。

```
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)); // 预期结果为5
               EXPECT_EQ(0, add(0, 0)); // 预期结果为0
               return Zstate::Z_SUCCESS;
        }
};
```

2.1.2 测试用例的注册

在宏定义的情况下可以使用测试注册中心 **ZTestRegistry** 动态注册测试用例，对于继承测试基类，由于其给予用户的自由度较高，安全起见要求用户手动添加。而链式创建的测试用例可以调用在 build 后调用注册函数实现注册或者手动添加。下面是测试注册中心类的使用示例：

```
// 手动注册示例
ZTestRegistry::getInstance().registerTest(std::make_shared<MathTests_Addition>());
```

2.1.3 测试套件的构建

通过 **ZTestSuite** 类组织多个测试用例，支持批量运行，支持多线程测试，会统计测试多个测试用例所需的时间，测试通过的结果，通过测试数量等值。

```
auto mathSuite = std::make_unique<ZTestSuite>("Math", ZType::Z_SAFE,"math test");
// 添加单测用例到套件
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());
```

2.2 断言机制

断言机制用于验证测试用例的预期结果是否正确。框架提供了多种断言宏，如 **EXPECT_EQ** 和 **ASSERT_TRUE**，支持在测试中快速验证条件。主要提供以下断言：

- **EXPECT_EQ**：验证两个值是否相等。
- **ASSERT_TRUE**：验证条件是否为真。

使用样例如下，

```
// 如果断言失败，抛出异常
EXPECT_EQ(5, add(2, 3));
ASSERT_TRUE(6==add(2, 3));
```

如果断言失败，抛出 **ZTestFailureException** 异常。同时可以通过继承 **ZTestFailureException** 异常处理函数，自定义异常处理逻辑。

2.3 测试执行器

测试执行器负责管理测试用例的运行，支持**多线程**并行执行测试用例，并收集测试结果，实现了自动调度测试和测试结果统计。

ZTestContext 用于测试上下文管理类，负责维护测试用例队列，并通过多线程并行执行测试。

```
//定义一个测试上下文对象
ZTestContext context;
// 将测试用例添加到测试队列
for (auto &&test : registeredTests)
    context.addTest(std::move(test));
// 多线程执行测试用例
context.runAll();
```

实际上一共可以定义两种类型的测试用例，一种是 **z__safe** 类型的测试用例，一种是 **z__unsafe** 型，两种类型分别对应两种测试模式，**z__safe** 类型的测试用例是线程安全的，**z__unsafe** 类型的测试器是线程不安全的，默认为 **z__safe** 类型。如果测试时候有 **z__unsafe** 类型出现，则会等待其执行完成，再执行 **z__safe** 类型的测试用例。

2.4 结果报告

2.4.1 报告文件保存

测试完成后，框架会生成详细的测试结果报告，包括测试名称、运行时间、通过/失败状态以及错误信息。我们定义评测结果状态有三种状态：

- **z__success**：测试通过。
- **z__failed**：测试失败。
- **z__unknown**：还没有完成测试。

日志输出测试结果通过 **ZLogger** 类输出到控制台或文件中。输出规则如下，[info] 表示测试信息，[FAILED] 表示测试失败，[OK] 表示测试成功。输出示例如下:

```
[   OK   ] BasicMath.NegativeTest (0 ms)
[  info  ]准备加法测试环境...
[   OK   ] Advanced.Multiplication (0 ms)
[ FAILED ] BasicMath.FailedSubtractionTest (1 ms)
Error: Test Failure in BasicMath.FailedSubtractionTest:
  Expected: 3
  Actual : 2
[ FAILED ] BasicMath (1 ms)
```

同时在 **ZTestResult** 类中，保存了测试的运行时间，测试通过/失败状态，错误信息等信息，用来保存测试结果。

2.4.2 GUI 展示

可以通过 GUI 展示测试结果，同时未来考虑在 GUI 界面加入添加测试用例，删除测试用例，修改测试用例，导出测试报告等功能，界面设计大体如下图所示。

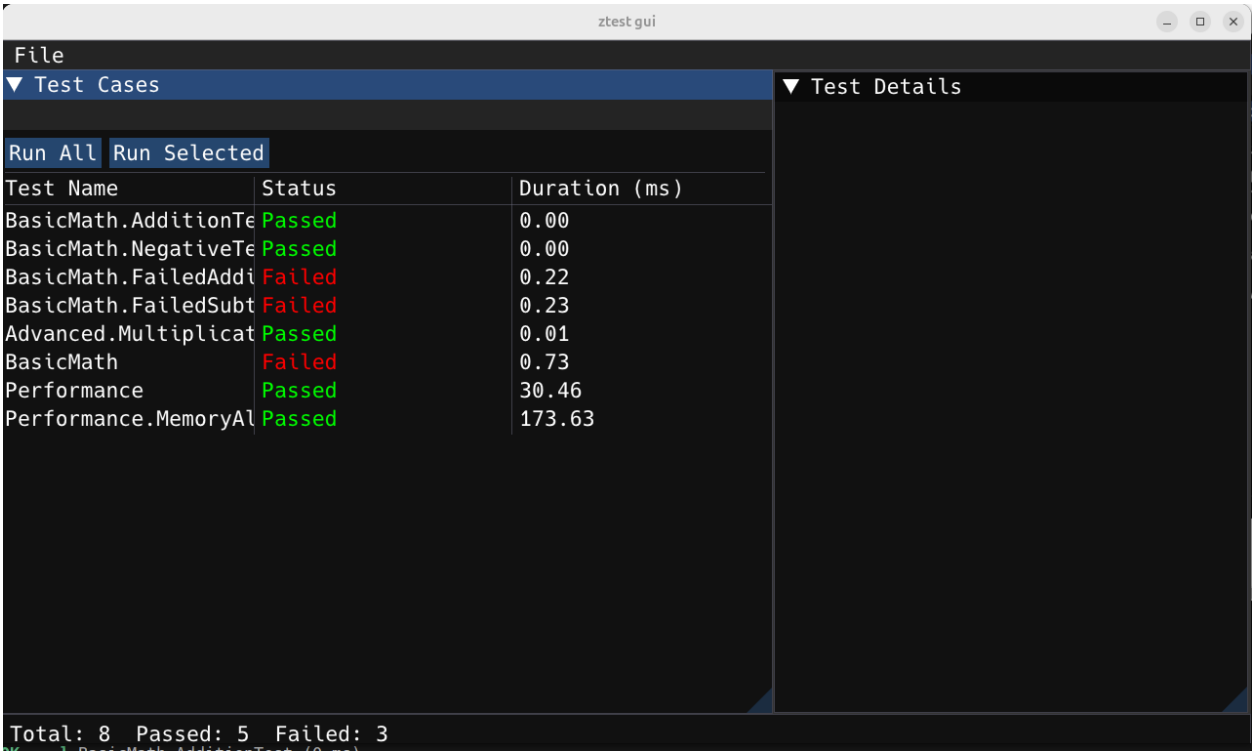

图 1: 测试管理界面布局示意图

3 整体架构

核心模块上我们通过插件模式加载测试用例和待测函数。核心模块分为四个部分。

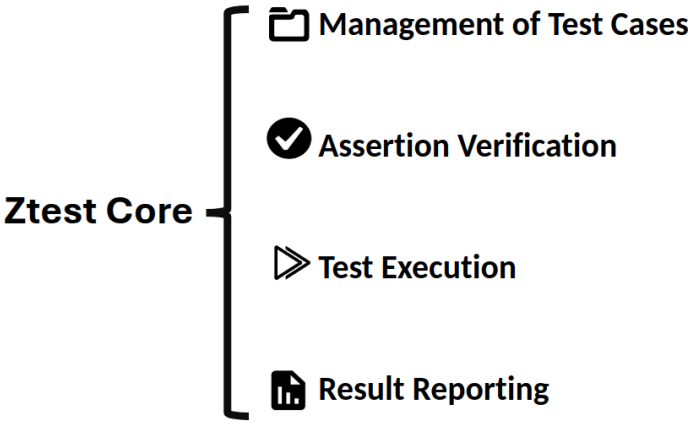

图 2: ztest core architecture

GUI 整体架构上使用的是 MVC(Model-View-Controller) 模式架构。View 关注用户所看见的 UI 并且提供交互,Controller 负责控制器作用于模型和视图上。它控制数据流向模型对象，并在数据变化时更新视图。它使视图与模型分离开。Model 主要实现对于底层文件的建模。

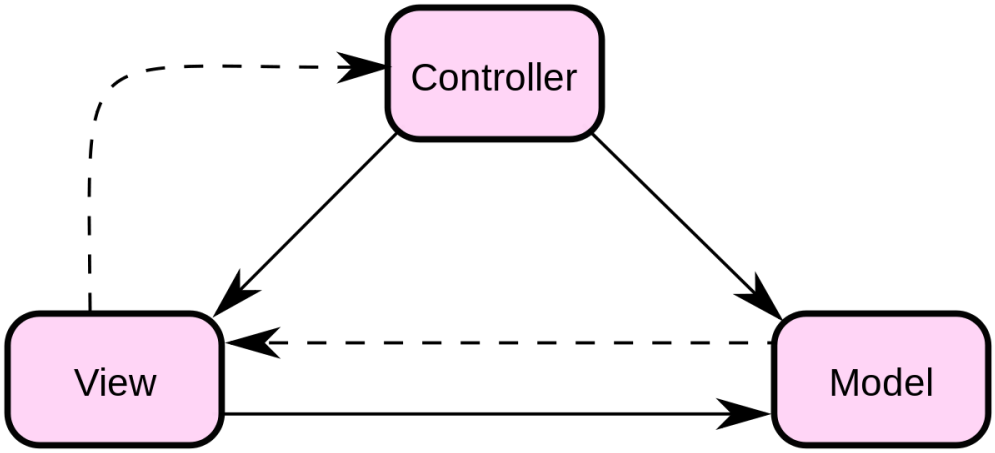

图 3: ztest gui architecture

4 技术细节

4.1 测试框架设计

- **ZtestInterface**：定义了测试用例的基本接口，包括获取测试名称、运行测试以及获取测试类型等方法。

- **ZtestBase**：作为测试用例的基类，实现了测试的基本属性和方法，如测试名称、测试类型、描述信息、前置钩子函数（beforeAll）和后置钩子函数（afterEach）等。
- **ZtestSingleCase**：继承自 **ZtestBase**，用于定义单个测试用例。支持设置测试函数、预期结果以及描述信息，并实现了测试的执行逻辑。
- **ZTestSuite**：用于组织多个测试用例，支持批量运行测试，并统计测试结果（通过、失败、总耗时等）。
- **ZTestRegistry**：采用单例模式实现的测试注册中心，负责注册和管理测试用例，支持动态添加测试用例。
- **ZTestContext**：测试上下文管理类，负责维护测试用例队列，并通过多线程并行执行测试用例，同时收集和管理测试结果。

4.2 测试用例的构建与执行

测试用例的构建通过 **TestBuilder** 类实现，支持链式调用设置测试函数、预期结果、前置钩子和后置钩子等。测试工厂类 **TestFactory** 提供了创建测试用例的便捷方法。

测试执行时，**ZTestContext** 类会从测试队列中取出测试用例，并通过 **ZTimer** 类计时，记录测试的开始时间、结束时间和耗时。测试结果通过 **ZTestResult** 类封装，并在控制台输出测试结果的详细信息。

4.3 测试断言与异常处理

框架提供了断言宏 **EXPECT_EQ** 和 **ASSERT_TRUE**，用于比较预期值和实际值，或验证条件是否为真。如果断言失败，会抛出 **ZTestFailureException** 异常，并记录错误信息。

4.4 测试套件与多线程执行

测试套件（**ZTestSuite**）可以包含多个测试用例，并支持批量运行。测试上下文（**ZTestContext**）通过多线程并行执行测试用例，充分利用多核 CPU 的计算能力，提高测试效率。

4.5 日志管理

框架通过 **ZLogger** 类提供日志功能，支持线程安全的日志输出。日志信息包括测试结果、错误信息以及测试用例的前置和后置操作信息。

4.6 UI 实现

本系统采用 MVC 架构实现图形用户界面，通过 Dear ImGui 框架构建可视化测试管理界面。UI 模块的核心组件包括模型管理、视图渲染和控制器交互三部分，具体实现如下：

4.6.1 模型管理类 (ZTestModel)

ZTestModel 维护测试用例的状态信息（名称、状态、耗时、错误信息），提供线程安全的数据更新接口 `updateFromContext()`，记录当前选中测试用例 `_selected_test`，并跟踪测试运行状态 `_is_running` 和进度 `_progress`。

4.6.2 视图渲染类 (ZTestView)

ZTestView 负责渲染用户界面，包括主菜单、测试列表窗口和详情窗口。主菜单渲染 (renderMainMenu) 实现文件菜单退出功能。测试列表窗口 (renderTestList) 采用三栏式布局显示测试名称、状态和耗时，支持测试用例选择交互，并提供「全部运行」和「运行选中」操作按钮。详情窗口 (renderDetailsWindow) 显示选中测试的详细信息，包含错误消息的多行文本展示，并使用颜色编码（绿色表示成功，红色表示失败）标识状态。

4.6.3 控制器类 (ZTestController)

ZTestController 实现测试执行的线程管理，提供 `runAllTests()` 和 `runSelectedTest()` 方法，处理多线程与模型状态的同步。

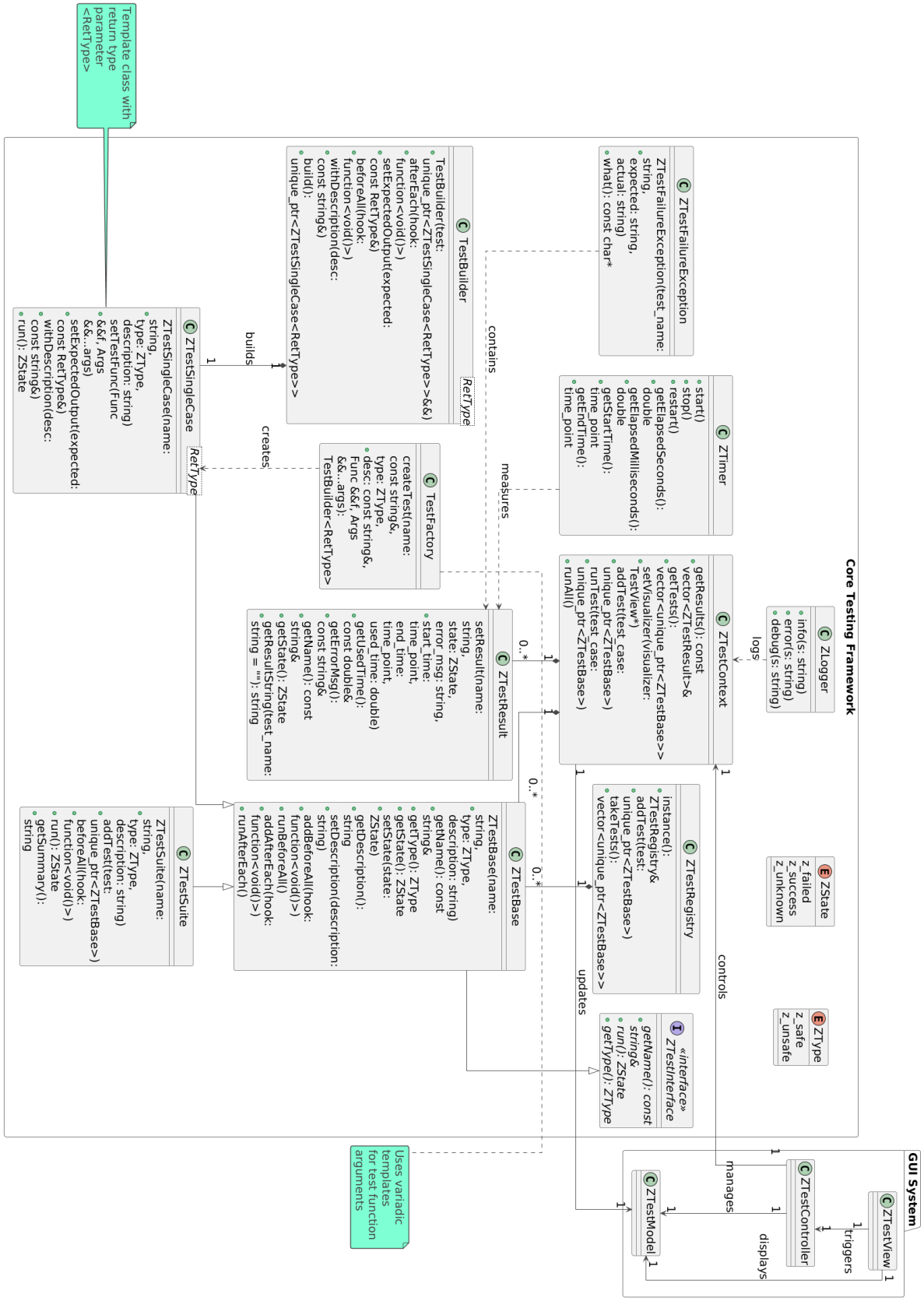

图 4: uml class diagram

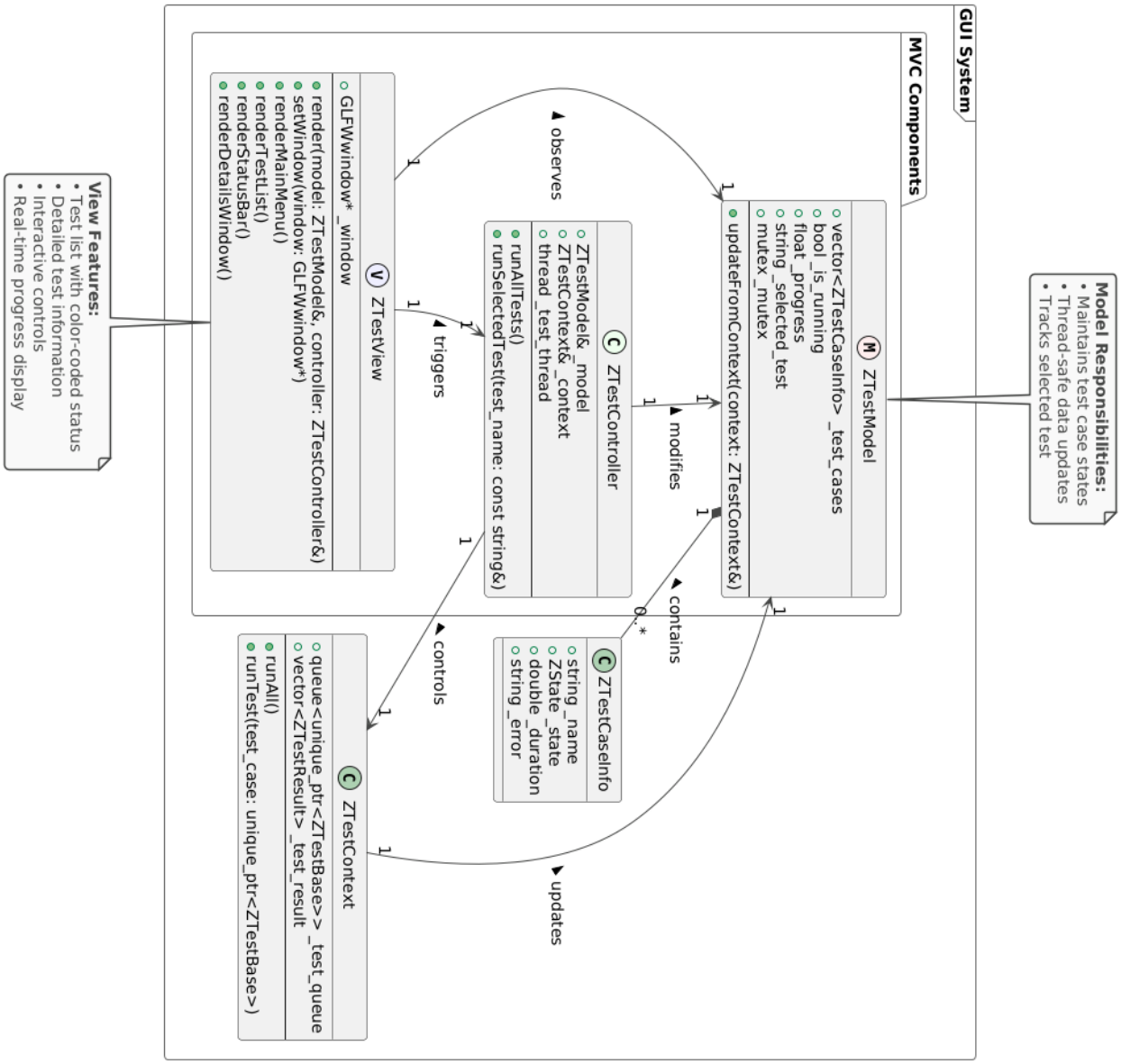
GUI System
MVC Components
Model Responsibilities:
• Maintains test case states
• Thread-safe data updates
• Tracks selected test
ZTestModel
vector<ZTestCaseInfo> _test_cases
bool _is_running
float _progress
string _selected_test
mutex _mutex
updateFromContext(context: ZTestContext&)
ZTestController
ZTestModel& _model
ZTestContext& _context
thread _test_thread
runAllTests()
runSelectedTest(test_name: const string&)
ZTestView
GLFWwindow* _window
render(model: ZTestModel&, controller: ZTestController&)
setWindow(window: GLFWwindow*)
renderMainMenu()
renderTestList()
renderStatusBar()
renderDetailsWindow()
ZTestCaseInfo
string _name
ZState _state
double _duration
string _error
ZTestContext
queue<unique_ptr<ZTestBase>> _test_queue
vector<ZTestResult> _test_result
runAll()
runTest(test_case: unique_ptr<ZTestBase>)
View Features:
• Test list with color-coded status
• Detailed test information
• Interactive controls
• Real-time progress display
observes
modifies
triggers
contains
controls
updates
1
0..*

图 5: gui 类图

5 系统关键难点

5.1 待测函数的自动类型推导

我们设计在使用 **ZtestSingleCase** 时候，用户只需要传入函数名、参数和期望结果，就可以由 **ZtestSingleCase** 接管运行的具体逻辑和测试结果的比较。这要求我们在构建 **ZtestSingleCase** 的时候，实现自动推导待测函数的参数类型，返回值类型，以便调用函数用于测试和测试结果验证。主要使用泛型和闭包实现。

5.2 待测试文件和测试文件的分离和动态加载

在简单的测试中，似乎不分离待测文件和测试文件的问题不大。但是当待测项目的规模不断增大，项目的测试用例也变多时，如果仍然不分离，代码的维护和修改将非常麻烦。作为测试框架，我们需要提供将待测文件和测试文件分离的能力，将测试文件放入 **test** 文件夹下，待测文件放入 **src** 文件夹下。我们设计的是用户可以通过动态库链接的方式，实现测试执行。从更高的视角来看，此处我们的 GUI 测试框架相当于是主程序，而用户的待测代码和测试用例相当于插件。我们需要提供接口 **ZTestAPI**，使得主程序可以调用插件函数 (测试文件和待测文件中的函数)。

图 6: 目录文件结构

```
//待测试函数导出示例
ZTestAPI int subtract(int a, int b) { return a - b; }
```

5.3 多种构建方式的实现

实现多种测试用例构建方式，实际上是为了为用户提供更好的体验。从实现上来说，链式创建构建主要使用了 builder 设计模式，单个测试用例的创建主要使用了工厂模式。而宏定义实现主要使用了将宏展开到继承定义。多种语法的实现方便了用户，但是如何将这些语法构建的类统一的管理又是一个难点。

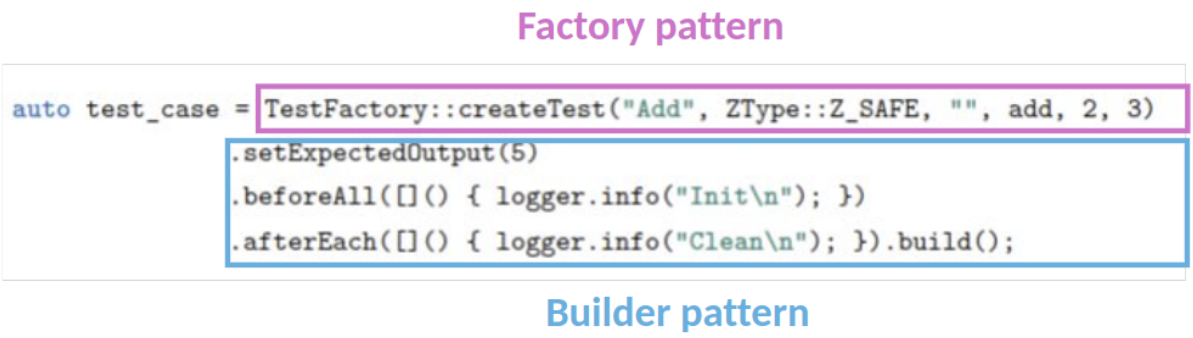

图 7: 实现链式创建

5.4 安全的多线程评测的实现

实际上并不是所有的函数都是多线程安全的，所以我们需要对于待测函数进行分类，分别进行多线程安全的评测。实际上一共可以定义两种类型的测试用例，一种是 **z_safe** 类型的测试用例，一种是 **z_unsafe** 型，两种类型分别对应两种测试模式，**z_safe** 类型的测试用例是线程安全的，**z_unsafe** 类型的测试器是线程不安全的，默认为 **z_safe** 类型。如果测试时候有 **z_unsafe** 类型出现，则会等待其执行完成，再执行 **z_safe** 类型的测试用例。

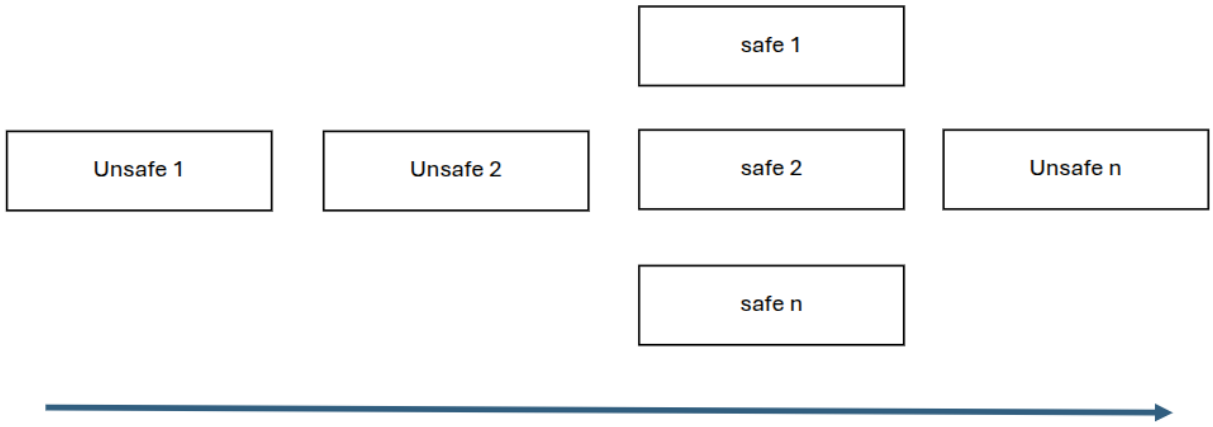

图 8: 实现安全的多线程运行

5.5 测试结果数据存储

我们使用了多线程评测以后，每次评测的顺序都不一定是一样的，但是我们不希望每一次评测的结果都是不一样的顺序显示在 GUI 上，同时我们还希望对历史测试结果进行存储，否则会极大降低用户体验。从设计的角度来看，我们应当增加一个中间层，更加确切地说，我们可能需要在中间添加一个类似于数据库的模块，来存储测试结果，并且为 **ZTestModel** 使用。

5.6 指针的管理

考虑到本框架使用了大量的指针，很容易就会出现内存泄漏等问题，所以我们决定使用 C++11 中的智能指针来管理内存，来提升系统的可靠性。

表 2: 详细责任分工与技术实现要点

功能模块	负责人
GUI 实现	吴泓庆
断言机制	王瑞箐
测试用例管理	郑辰阳、叶穗华
测试用例执行	郑辰阳
报告结果生成	齐彦淞

职责描述与实现细节

GUI 实现
- 基于 Dear ImGUI 框架构建可视化界面
- 实现测试列表的三栏布局 (名称/状态/耗时)
- 开发进度条组件与状态栏实时更新
- 实现测试详情窗口的错误信息渲染

断言机制
- 设计类型安全的模板化断言宏 (EXPECT_EQ/ASSERT_TRUE)
- 实现彩色错误输出
- 开发可扩展的异常处理框架 (ZTestFailureException)
- 支持自定义比较算子的注册机制

测试用例管理
- 构建链式 API(TestBuilder) 实现流畅接口
- 开发宏展开系统 (ZTEST_F) 实现自动注册
- 设计测试套件 (ZTestSuite) 的树形组织结构
- 实现前置/后置钩子的依赖注入机制
- 开发插件式测试加载系统 (动态库集成)

测试用例执行
- 构建线程池调度器 (ZTestContext)
- 实现安全/非安全测试的隔离执行策略
- 设计测试优先级队列与超时终止机制

报告结果生成
- 设计结构化结果存储格式 (ZTestResult)
- 实现 ANSI 转义码的彩色控制台输出
- 构建历史测试结果的版本对比系统
- 集成数据可视化组件 (图表生成)

Reference

[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.