Test Harness

Phase #2 – Development

Group: Santhosh Srinivasan, Jiawen Zhen, Alifa Stith

1. Architecture

The high-level architecture for Test Harness can be reasonably divided into three distinct components: *TestController*, *TestLibrary*, and *TestUtilities*. The *Client* component is included to represent an easily extensible input source (console, GUI, etc.).

1.1. TestController Package

The *TestController* package is the main entry point to the Test Harness application and will be responsible for the following functionality: (1) launching the Test Harness application and receiving input provided by the user; (2) sending each *TEST* function to be run by *TestLibrary*; (3) initializing and maintaining queues (future); and (4) storing and maintaining a summary of test results.

1.2. TestLibrary Package

The *TestLibrary* package can be thought of as providing the main functionality of the application. This is the package responsible for running each test individually, obtaining the result, and catching and handling any exceptions. The *TestLibrary* package also produces and formats the test result log messages as well as any exception messages.

1.3. TestUtilities Package

The job of the *TestUtilities* package is to collect and log the output data throughout the execution of Test Harness. It calculates time information for each test as well as for the summary of test results. This package is crucially responsible for writing the logging information to permanent storage within the file system.

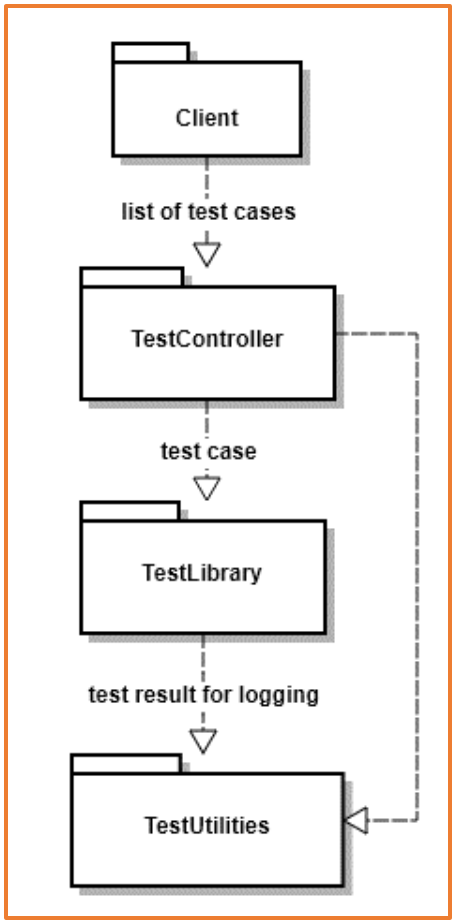

Figure 1. Block diagram of major components of Test Harness, with arrows indicating the flow of program execution.

1.4. Queue Interface

Currently, all our packages communicate directly through function calls. However, we attempted to organize our code in such a way to be easily extended to support concurrency in the future. Particularly, we plan on implementing a queue to hold test results and log messages. The *TestLibrary* package will enqueue log messages onto the queue as it obtains test results, while the *TestUtilities* package dequeues and logs them. This interface will allow us to run *TEST* functions in parallel without causing I/O errors when logging results.

2. Design

The design of Test Harness breaks down the architecture from the previous section into individual classes, with each package containing two classes. The *TestController* package is comprised of *TestHarness* and *TestResultCounter*; the *TestLibrary* package contains *TestRunner* and *TestExceptionHandler*; and the *TestUtilities* package is made up of *TestLogger* and *TestTimer*.

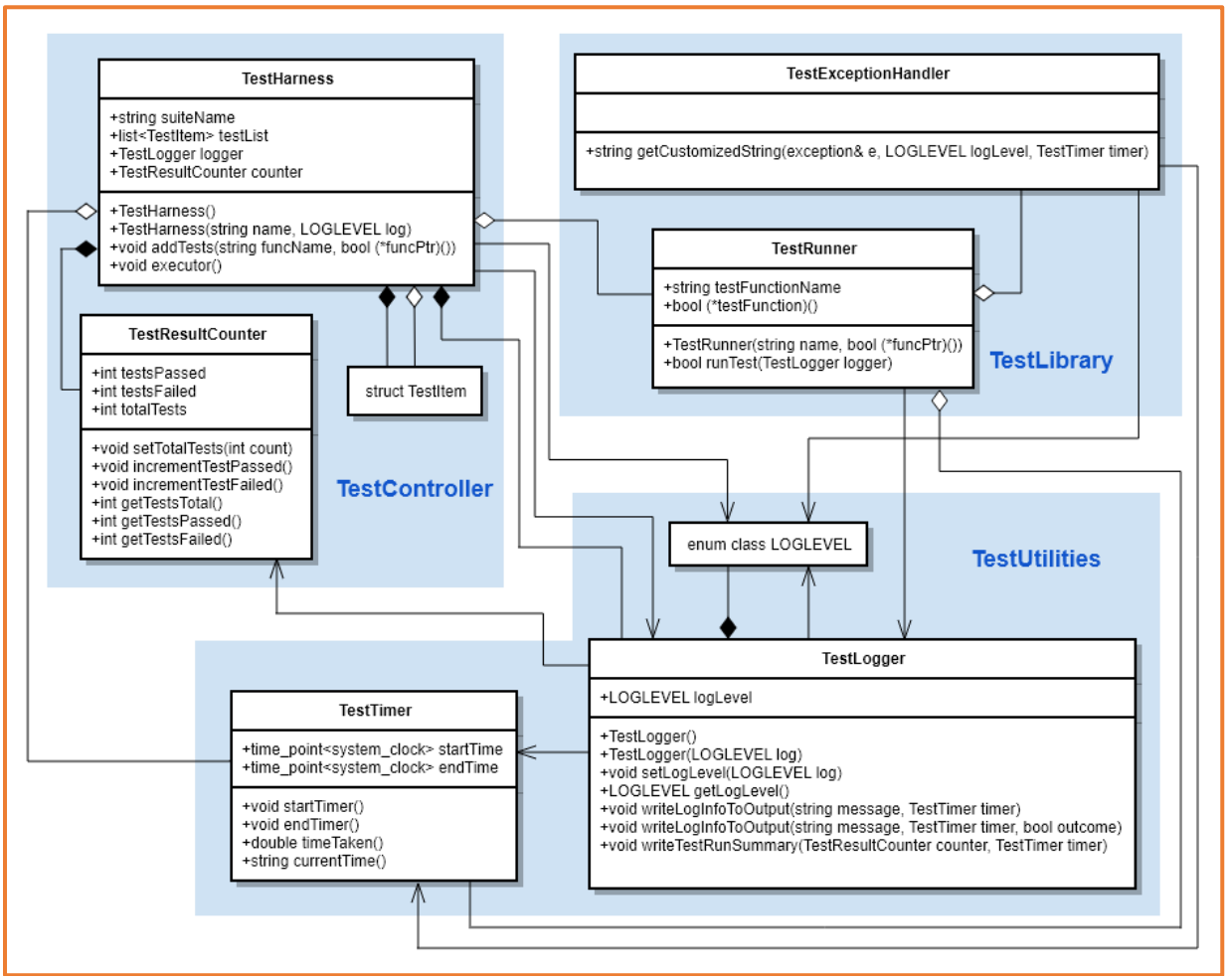

Figure 2. Class diagram (UML) for Test Harness. Classes are grouped by package membership, and lines show INHERITANCE, COMPOSITION, AGGREGATION, and USING relationships between classes. For a clearer image, please see external diagram in the class_diagram.png file.

2.1. Classes in TestController Package:
TestHarness and TestResultCounter

The *TestHarness* class is the entry point into the Test Harness application. It stores a list of function pointers (and their names), which should be populated with *TEST* functions by the user. *TestHarness* also initializes and stores the logger throughout the run, passing it to other classes when needed. When *TestHarness* is executed, it loops through each TEST in its list, creating *TestRunner* objects to run each test. Finally, *TestHarness* provides a communication interface between the *TestResultCounter* class, which stores and maintains a summary of test results, and the classes of the *TestUtilities* package.

2.2. Classes in TestLibrary Package:
TestRunner and TestExceptionHandler

Execution of each individual *TEST* function is handled by *TestRunner* objects. An instance of *TestRunner* is responsible for receiving a test, attempting to run the test and storing its result, catching any exceptions, and then passing the resultant message(s) to the *TestUtilities* package for logging. *TestRunner* also utilizes the *TestTimer* class to store the running time of the test. If an exception was caught when the *TEST* was run, *TestRunner* will obtain a custom string from *TestExceptionHandler* to describe the results. *TestExceptionHandler* determines the type of exception raised, as well as the log level of the logger, and returns a custom string depending on these parameters.

2.3. Classes in TestUtilities Package:
TestLogger and TestTimer

The *TestLogger* class receives logging and exception messages from *TestRunner*, formats them, and prints them to the console. It also obtains and logs the summarized results of the entire Test Harness session from the *TestController* package. The *TestTimer* class maintains and provides timestamp information for each test as well as the entire session summary. It is utilized by the *TestLibrary* package to time each test and set message timestamps, and by the *TestController* package to store intermediate data prior to logging the summary.

3. Team Member Roles

During phase 1, our team collaborated through Zoom calls to settle on the architecture and design of the project. The entire team was responsible for the architecture and design concept. A GitHub repository was created to share generated documentation and code produced during this project. A separate Slack channel was created for team communications. We also agreed that none of us intended to claim “ownership” of entire packages; instead, we intended to collaborate heavily during the implementation phases of the project. Additionally, we agreed to communicate regularly, provide feedback through code review at least weekly, and assist one another in the most complex and time-intensive sections of the source code.

Our initial meeting for phase 2 consisted mainly of discussing specific implementation details, writing some pseudocode, and delegating responsibilities. We split phase 2 into two general sub-phases, each to span across approximately 1½ weeks. We also created another GitHub repository to keep our notes, documentation, and source code better organized. During phase 2a, we planned on implementing the *TestUtilities* and *TestLibrary* packages. These packages establish the foundation of the application, so they were important to add first. We split the work into three sections and used a random number generator to assign ourselves responsibilities: Alifa would implement the *TestRunner* class, Jiawen would implement the *TestExceptionHandler* class, and Santhosh would implement the classes of the *TestUtilities* package.

The next weekend, we scheduled another meeting to share code review feedback and discuss some design and implementation changes, which we would take the next week to fulfill. We also discussed a general plan in which we would collaborate on developing the *TestController* package, adding *TEST* classes, and updating the documentation. The final week or so of work was spent on these tasks, as well as adding some changes requested by other team members and bug fixes.