GMU-Team 1: Documentation Report

IDT Spring 2014

Brannon Crymes - Akshay Karthik – Sriram Rajaraman

2014

Contact Details

Brannon Crymes - bwinter2@gmu.edu

Akshay Karthik - akshay.karthik@gmail.com

Sriram Rajaraman - s.rajaaraman89@gmail.com

Contents

[3 Introduction 2](#_Toc377500112)

[3.1 Running the System 2](#_Toc377500113)

[3.2 Building the System 3](#_Toc377500114)

[4 Usage 4](#_Toc377500115)

[4.1 Configuration 4](#_Toc377500116)

[4.2 Standard Testing API 4](#_Toc377500117)

[4.3 Report Writers 6](#_Toc377500118)

[4.4 Testing Race Conditions 6](#_Toc377500119)

[4.5 Log Viewer 6](#_Toc377500120)

[5 Architecture 7](#_Toc377500121)

[5.1 GMUT 7](#_Toc377500122)

[5.2 Test Builder 7](#_Toc377500123)

[5.3 Predicates 7](#_Toc377500124)

[5.4 Report Writer 7](#_Toc377500125)

[5.5 Log Viewer 7](#_Toc377500126)

[6 Requirements Traceability 8](#_Toc377500127)

# Introduction

We were asked to implement a system that allows developers to place checks within their code that can record the results of their code and asses the logic of the system. We call our system GMUT or GMU Tester. When we architected the system, we focused on simplicity of use, extensibility, and performance. We used plain java with no additional libraries to minimize the footprint of our system. The system is well documented and tested and fails gracefully if the application crashes. It does not leak memory and its thread safe API makes it easy to use in a variety of applications.

## Running the System

The logging framework can be run directly through the com.idt.contest.college.winter2014.Main interface. It works in both batch script mode as well as menu driven mode. The system returns results in a log.txt file that follows a specific format. This format was chosen to be able to search through via regular expressions and is documented in documentation/report\_format.txt. This format is used in logviewer as well as all of the included ReportWriters however, the extensible nature of the system allows for custom ReportWriters in any format that a developer wishes to use. There are three key access points to the system:

1. gmu\_main.jar is the com.idt.contest.college.winter2014 jar with the functions annotated by the GMUT testing API. It can be run without arguments to access the menu driven application. If an argument is provided, it runs using a batch script.   
   java –jar gmu\_main.jar  
   java –jar gmu\_main.jar batchscript.txt
2. gmu\_logviewer.jar is the application to view the logs.  
   java –jar gmu\_logviewer.jar
3. gmu\_tests.jar is the unit tests for the GMUT API. It is a simple console application that runs all of the unit tests that verify that the system meets the given requirements.  
   java –jar gmu\_tests.java

## Building the System

Building the system is simple, it can be built from eclipse.

1. Import project into eclipse (file > import existing project)
2. Run Configurations
   1. LogViewer.java for log viewer
   2. Main.java for the IDT application
   3. InternalTestRunner.java for the internal tests application

Also provided is an Ant build.xml file for Apache Ant build system. This system automatically builds all three jars as well as places the default batchfile.txt file in the main directory. It also builds the javadocs for the project and places it in the documentation directory.

To run the ant build file, simply run the following console command. This assumes that java as well and ant is available on the console’s PATH.

ant -buildfile build.xml

The ant file performs five actions:

1. Build /src into gmu\_main.jar with com.idt.contest.college.winter2014.Main as the main class.
2. Build /src into gmu\_logviewer.jar with edu.gmu.team1.idt2014.logviewer.LogViewer as the main class
3. Build /test into gmu\_tests.jar with edu.gmu.team1.idt2014.test.InternalTestRunner as the main class
4. Copy batchscript.txt into the main directory.
5. Generates javadocs and places them into the documentation/javadoc folder.

# Usage

Before working with the GMUT system, it is essential to understand what a test is in this framework. This framework is developed in terms of predicates. In math, a predicate is a function that takes a set of values, and returns a Boolean value. An example of a predicate is equality (if x = 5, return true, else false). In this library, a test is defined as a mapping between an input predicate (one that evaluates the input) and an output predicate (one that evaluates the output). For example, let us take the example function isGreaterThanTwo. We can explain each test as a predicate.

if input is 0 output should be false - ( input == 0 ) -> ( output == false )  
if input is 3 output should be true - ( input == 3 ) -> ( output == true )

Similarly, any test in this library is a mapping of a predicate to another predicate. There are many predicates provided and many more can be used by subclassing the Predicate class.

## Configuration

By default, the system starts out enabled with the reporting system configured to output to a file called log.txt. It is very simple to enable or disable the API. Simply call GMUT.disable() to disable the testing system. The output location can be configured by using GMUT.setReportWriter where you may pass in a new ReportWriter. Further details on ReportWriter usage is in section 4.3.

## Standard Testing API

When looking at the problem statement, we split our solution into two key sections. Declaring the test cases, and actually performing the tests. These requirements, along with emphasis on API simplicity lead to the usage pattern described here. The standard testing API has a single point of access. “edu.gmu.team1.idt2014.GMUT”. GMUT is a thread safe singleton and should be accessed statically. GMUT has two key functions, addTest and test. addTestis the test creation interface and testperforms the actual testing. To best understand how the system works see section 5.3 as it describes in detail what happens when GMUT.test is called.

Let us take a look at the example function (isEven) which takes an integer and returns true if the number is even, false if the number is odd).



Here, the creation of the test cases are done in lines 2-6, the actual testing is done in lines 8 and 11. One could read line 4 as if saying, if the input is 10 (first predicate is the input predicate), the output should be true (the second evaluates against the output). Finally in lines 8 and 11, we say, run the test (with the arguments (output, branch, input)).

GMUT.addTest returns an instance of ITestBuilder. ITestBuilder is a system that follows the factory pattern and sequentially builds up a test case until line 6 where it builds the tests and inserts it into the main GMUT system. There are two variants of the test creation method. .test expects two predicates in the order of input and output. .testNote does the same except it has the added functionality of marking a note next to each test.

Actually running the tests is as simple as calling GMUT.test with the arguments in the order, output, branch, inputs. The test method supports variable length arguments so one may simply list all of the inputs to the method being tested. Thus, usage of the API is very simple, in the beginning of the function, create the test cases

## Report Writers

ReportWriters are pluggable classes that can be set in GMUT that allow for results of tests to be logged. The Library comes with three ReportWriters already implemented. These are FileReportWriter, ConsoleReportWriter, and StringReportWriter. By default these ReportWriters follow the report format expected by logviewer however it is trivial for a developer to subclass AbstractReportWriter and have it write the files in a new format. FileReportWriter is the default ReportWriter for and has a variable FILENAME which is (by default) set to “log.txt”. By changing this variable, a developer may easily change the location of the file. As expected ConsoleReportWriter writes the results of the tests to System.out. StringReportWriter logs the results of tests to a public variable report.

The report format is detailed in documentation/report\_format.txt however, a quick description of the file format follows:

[<DATE>][<TIME>][<PASS/FAIL>][c:<CLASS>][m:<METHOD>][b:<BRANCHES>][i:<INPUTS>][o:<OUTPUT>][n:<NOTES>]

The above is a description of a single line, each line ends with windows style line endings “\r\n”. Multiline objects replace “\r\n” with “{r}{n}”. A line starting with tilde (~) is a comment line and is not parsed for ReportWriter. A line starting with tilde + exclamation mark (~!) is for custom testers (especially ThreadTester).

## Testing Race Conditions

To check if a race condition happens in a single method, first create an id for a variable that is shared by threads. This should be done once when the class containing the method is instantiated.

ThreadTester.createStateTracker("forks");

Inside a method where race condition might happen, get the current state of the variable by using the id, the ID should ideally be unique to each method.

int currentThreadState = ThreadTester.getState("forks");

Before the variable is changed, you want to compare the thread's state with the global state with:

ThreadTester.compareWithGlobalState("forks",currentThreadState,"Release Forks");

In this method, the first parameter is the ID, followed by the current thread’s state and then a note to log about the current method. If you have more than one statement that modifies the variable, then increment the local state by one after each modifier statement. Sometimes, you might have one method accessing and another method modifying. An example could be getTime(); setTime(new Time());. In this instance, a race condition may occur between thee two lines of code. To check for race conditions between these lines, first create an id for a variable that is shared by threads.

ThreadTester.createStateTracker("forks");

Inside an accessor method (i.e. getTime()), create an entry with:

ThreadTester.createStateTrackerOverride(Thread.currentThread().getName(),ThreadTester.getState("forks"))

Then, inside a modifier method (setTime()), check if the global state has changed between the methods:

ThreadTester.compareStates(currentThreadState,tester.getState(Thread.currentThread().getName()),"Release Forks");

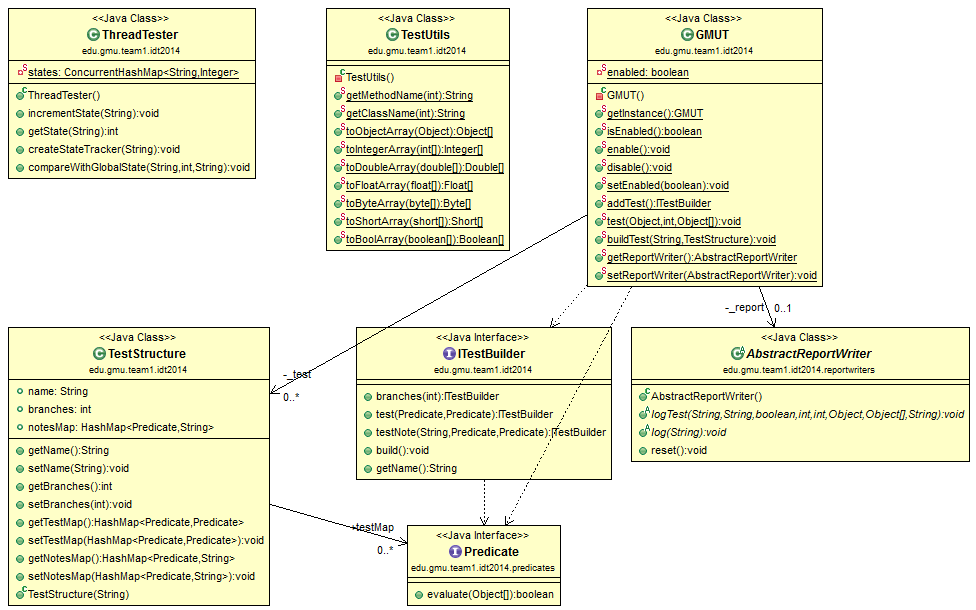
## Log Viewer

# Architecture

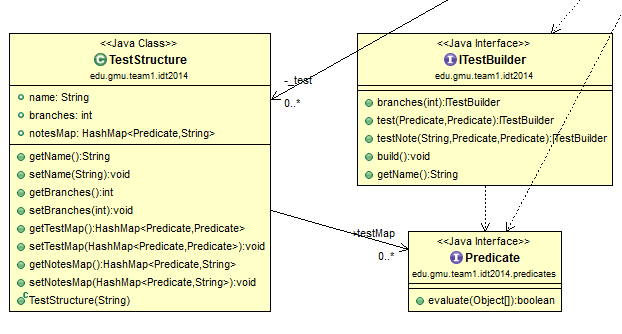
## Overview

When designing this architecture we focused on the five key evaluation points (performance, correctness, maintainability, usability, and elegance). We focused on the making the API as lightweight as possible and easily extensible. As a result, we split the problem into three key sections.

1. Defining Test Cases
2. Performing the Tests
3. Logging the Results

We realized that decoupling these components would drastically increase maintainability and extensibility. For example, it is simple in our system to change the logging of the results by extending AbstractReportWriter and passing into GMUT. As a result, when discussing the architecture, it is necessary to look at these three sections individually and then see how they interact.

## Defining Test Cases

In this library, a test is the mapping between a predicate for input and a predicate for output. A predicate is an Interface with a single method (evaluate) which takes a number of arguments and returns a Boolean value. One of our concerns was future proofing the library so the Interface follows the SAM (Single Abstract Method) property. If this library is used in Java, 8 to use Java 8 style Lambda functions in creating Predicates, simply uncomment line 11 in /src/edu/gmu/team1/idt2014/predicates/Predicate.java. In addition the library comes with many predicates predefined and available for use. The most used among these are Equals, MultiEquals, and ArrayEquals. Equals checks for equality of two objects (input == 10). MultiEquals checks for multiple objects being equal to each other (10, “hello” == 10, “hello”). ArrayEquals checks that two arrays are equal ([1, 2, 3] == [1, 2, 3]).

After defining predicates, it is simple to use TestBuilder. TestBuilder is a fluent class that utilizes the builder pattern to generate a TestStructure. A TestStructure is the name of the class, the method, and a list of test cases. TestBuilder iteratively builds a new TestStructure and is Thread Safe so multiple tests can be built at the same time.

A TestStructure is simple a value object that stores all of the tests for a given class and method and is stored in GMUT and searched through when testing any method. TestStructure stores the tests in a HashMap thus it is imperative that Predicates do not override hash Code or ensure that each predicate has a unique hash code otherwise collisions in HashMap will cause tests to be overwritten.

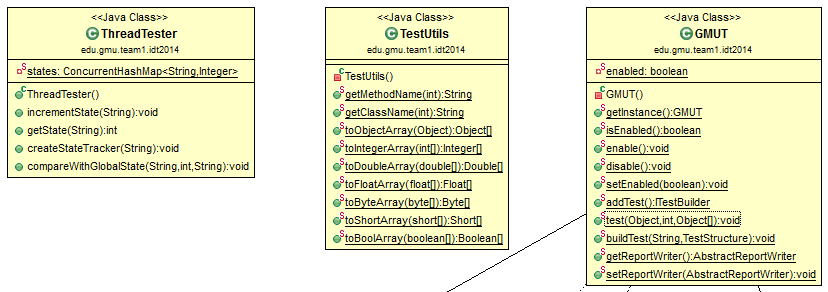
## Performing Tests

### Standard API:

In contrast to the complexity of building a test case, actually evaluating a test is relatively simple.

Let us take a look at this algorithm.

1. Check if testing is enabled, if not exit test method.
2. Check if test cases exist for a given method. If they don’t, exit test method.
3. Get the test structure (list of predicate to predicate mappings)
4. For each test case in the test structure
   1. Evaluate the first predicate with the input of the function (e.g if input = 10)
      1. Evaluate the output with the second predicate (output = true)
      2. Log the value of the second predicate (true = passed test, false = failed test)

When developing the GMUT system, it was imperative that (for simplicities sake) there be a single point of access for a developer. This lead to a single point of access for testing. Testing essentially compares the input and output to see if they are what was expected. Also, in java a function can only have one output, it may have many inputs however. As a result, .test has the arguments input, branch of execution, and then a variable length argument of inputs. However, in order to make predicates generalizable and not input/output specific, all predicates take in variable length arguments to evaluate. As a result, most predicates, (especially in dealing with outputs) specifically look at the first argument in the list of variable length values.

Another consideration in architecting this system was looking at thread safety. This function is thread safe as no global variables are modified while performing testing. Each thread accesses the same instance of ConcurrentHashMap which holds the test structures. However, these accesses are all retrievals and no thread will be writing into the structure so no race conditions occur.

### ThreadTester:

Let's say you have a variable int x = 10 and you have a method that does:

method {  
int y = getX();  
setX(y\*2)  
}

In a SINGLE thread, the thread will read x as 10, then set it 10\*2 (20). That's easy. However, let's say it is a two-threaded program. I am going to call the Threads: t1, t2; let’s look at this scenario.

1. T1 - reads the variable so y = 10 (local variables are located in the stack. AKA each thread has its own y and is not affected by other threads)
2. T2 - reads the variable so its y =10
3. T1- sets the variable x = 10\*2 = 20 (Global variables are affected by other threads. Its a shared resource)
4. T2- sets the variable x = 10\*2 = 20

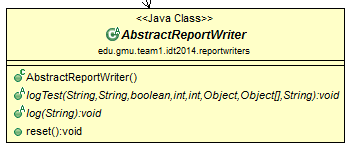
Now this is wrong. At the end, x should be 40. This is what race condition is. The last thread does not realized that the variable has been changed. We have to track the state of the global variables and make sure Threads’ have the current state of the variable. Let’s use the same example with two threads: T1, T2

int x = 10; it is currently at state 1 and this is when the variable is initialized in the beginning.

1. T1 reads the variable as 10 at state 1.
2. T2 reads the variable as 10 at state 1.
3. Before modifying the variable, check the states. T1 state is 1 and the global state is 1. That means that the global state has not been modified in anyway and T1 has the up-to-date version of the variable.
4. T1 sets the variable as 20 at state 2 - the global state increments when variable is modified.
5. Now we have check the states for T2 and global state. T2 state is 1 and the global state is 2. Race Condition has occurred. T2 has a previous version of the variable.

As long as the global state and Thread's state does not match, race condition has occurred because some other thread has modified the variable

## Logging Results

When architecting the reporting system, it was necessary to make it generalizable and configurable. As a result, GMUT holds an instance of AbstractReportWriter. AbstractReportWriter is an abstract class that has two functions, logTest (which logs the results of a test) and log (which logs comments). GMUT has only one instance however the instance can be overwritten with a new one. It is easy to extend AbstractReportWriter and replace it and this was one of the key considerations as not all developers will want to use the same report format. It was necessary for us to generalize this as we ourselves used three different ReportWriters in our testing. We used StringReportWriter which writes to a string, ConsoleReportWriter, which writes to System.out, and the default, FileReportWriter which writes to log.txt.

# Requirements Traceability