IDT Programming Contest – College Winter 2014

Built-in Automated Testing

This year's contest challenge is focused on automated testing at the code level. Traditionally, software developers write a class or method, and then develop a series of unit tests to verify that the code works as expected. Unit tests are developed separately from the code and the two are usually stored in separate modules. Contest participants will be asked to implement an API/framework in Java that allows a developer to add checks directly into their code that should record the results and produce an easily readable report. The report produced at the end of executing the instrumented software should speak to the health of the code (logic is working as expected) and the amount of code that was covered at runtime (code coverage).

Problem Context

Large scale software can have incredible complexity and it is possible for a small change in the software code to have large consequences. Concepts like inheritance, aggregation, and composition encourage building new functionality on top of existing code, resulting in dependencies. Because of this coupling, changes in one location could affect the logic in another. In order to verify that a method logically works as expected, developers put together unit tests that exercise the method in a number of ways, and verify that each of the results meet expectations. Unit tests are typically automated to run at night and produce a report on the health of the code base for developers to review in the morning. In addition to unit testing libraries, developers often use code coverage libraries to detect which code blocks have or have not been executed at runtime. Imagine that you have a simple method with an If and Else block. If you have single test for this method, there is no way that you could exercise both the If block and the Else block in one pass. A code coverage library would indicate which block has not been executed and the developer would use this information to write additional tests. Unit testing and code coverage are two automated testing techniques that are used to ensure delivery of the highest quality software possible.

Challenge Description

The API/framework that you develop should allow a developer to insert checks directly into the code that they are writing, similar to how a logger allows a developer to insert logging statements directly into their code. Your solution should produce a human readable report that indicates whether methods that were executed at runtime are working as expected (logic verification). For Example consider the following extremely simple logic:

```
public boolean isGreaterThanTwo(int inputValue) {
    if (inputValue > 2) {
        return true;
    }
    else {
        return false;
    }
}
```

The goal is to create a very succinct API that can be used to verify that the logic is working as expected for certain values. The previous logic with this built in testing might look like:

```
public boolean isGreaterThanTwo(int inputValue) {

    BuiltInTester.expecting(inputValue, 0, "return false");
    BuiltInTester.expecting(inputValue, -1, "return false");
    BuiltInTester.expecting(inputValue, 2, "return false");
    BuiltInTester.expecting(inputValue, 3, "return true");
    BuiltInTester.expecting(inputValue, Integer.MAX_VALUE, "return true");


    if (inputValue > 2) {
        BuiltInTester.log("return true");
        return true;
    }
    else {
        BuiltInTester.log("return false");
        return false;
    }
}
```

The report output from running with input values of [2, 500, -1] might look like:

```
10/21/13: 06:45:54AM: isGreaterThanTwo with input "2" PASSED with "return false".
10/21/13: 06:45:54AM: isGreaterThanTwo with input "-1" PASSED with "return false".
```

Notice that though the value 500 was presented to the program it did not produce any results in the report from our example BuiltInTester API. This is expected given the design choice of this example API to focus on direct equality checks for specific expected values. Since 500 is not setup to be expected via one of the BuiltInTester.expecting() lines it is ignored by the API and the subsequent BuiltInTester.log() is also ignored so as not to clutter up the output report when we don't have a specific result (PASS/FAIL) to record/display.

The above examples are only one indication of how this problem might be solved. Feel free to design your own API that solves the problem and potentially looks nothing like the examples above. In addition the above examples only show the simplest case of keying on a single value within a single method, thought should be applied in the design as to whether or not class comparisons are needed and whether or not to support internal tests that are not limited to simple equality or the internals of a single method.

In addition to Logic verification the submissions should provide information on the code coverage achieved by a specific test. For example in the previous sample we can see that the work done to

annotate the code with the Built In Test statements makes it possible to extract information about the number of different paths through the code. Looking at the number of different expected values in the `BuiltInTester.expecting` statements in the example we can see that there are only two expected paths through the code. During a specific test run we can then determine how many of these expected values were actually seen to generate a path coverage metric.

Your API/framework is expected to:

- At least cover all of the primitives in Java
- Be as lightweight as possible, both in the API calls and their internal implementation
- Be globally controlled so it is simple to enable or disable
- Handle multi-threaded execution of tested code
- Produce readable reports (even in the multi-threaded environment)

IDT has provided a sample application (com.idt.contest.college.winter2014.jar) that you will test with your API/framework. The sample application has two modes: a command line style menu for a user to interact with (menu mode) and a batch processing mode that accepts a script file as an argument in order to drive multiple methods in sequence (batch mode). **Your team should use your API to test the code inside of the com.idt.contest.college.winter2014.codetotest package, and your delivery will include the sample application instrumented with your API/framework code**.

Sample Application Documentation

The sample application, com.idt.contest.college.winter2014, was created as a Java project using Eclipse and Java 7. For ease of use, we have provided the application as an executable jar (com.idt.contest.college.winter2014.jar), a compressed Eclipse project (com.idt.contest.college.winter2014.zip), and an example batch script to drive the application.

The Java project has three packages containing java files:

com.idt.contest.college.winter2014, containing Main.java which contains the main method
com.idt.contest.college.winter2014.codetotest, containing files for you to instrument with your API
com.idt.contest.college.winter2014.framework, containing files to make the sample application run

To import the Eclipse project:

1. Unzip the archive file in your preferred workspace location
2. Launch Eclipse
3. File > Import > General > Existing Projects into Workspace
4. Browse to find the unzipped archive in your preferred workspace location
5. Select the com.idt.contest.college.winter2014 project and press Finish

To run the Eclipse project in menu mode (user drives the application):

1. Once the application has been imported, right click on Main.java
2. Right click on Main.java and select 'Run as Java Application'
3. When the application starts, the user will be presented with a menu in the Console window

To run the Eclipse project in batch mode (batch script drives the application):

1. Once the application has been run the first time in menu mode, a run configuration will exist
2. Click on the menu option 'Run' at the top of the application
3. Select Run Configurations
4. In the run configuration, select the Arguments tab
5. Add batchscript.txt (which is included at the top level of the project) to Program Arguments
6. Click apply and run
7. When the application executes, you will see results fill the command prompt window

To run the executable jar in menu mode (user drives the application):

1. Move the jar to your preferred workspace location
2. Launch a command prompt and change directory to preferred workspace location
3. java –jar com.idt.contest.college.winter2014
4. When the application starts, the user will be presented with a menu in the command prompt window

To run the executable jar in batch mode (batch script drives the application):

1. Move the jar to your preferred workspace location
2. Launch a command prompt and change directory to preferred workspace location
3. Move the batchscript.txt file to your current directory (it will be the only argument for the application)
4. java –jar com.idt.contest.college.winter2014 batchscript.txt
5. When the application executes, you will see results fill the command prompt window

Requirements and Traceability

Your software solution is expected to include at a minimum the required functionality. On top of implementing the following requirements, you will be asked to produce a traceability document describing how each requirement was tested by your team. Each requirement should be supported by a one sentence 'test case'.

1. The Built-in Testing API should be able to be added directly into the code that it is testing.
2. The Built-in Testing API should be able to support the byte Java primitive type.
3. The Built-in Testing API should be able to support the short Java primitive type.
4. The Built-in Testing API should be able to support the int Java primitive type.
5. The Built-in Testing API should be able to support the long Java primitive type.
6. The Built-in Testing API should be able to support the float Java primitive type.
7. The Built-in Testing API should be able to support the double Java primitive type.
8. The Built-in Testing API should be able to support the boolean Java primitive type.
9. The Built-in Testing API should be able to support the char Java primitive type.
10. The Built-in Testing API should be able to support the String Java type.
11. The Built-in Testing API should be able to support the int array (int[]) Java type.
12. The Built-in Testing API should verify that a given block of code performs as expected.
13. The Built-in Testing API should provide code coverage metrics.
14. The Built-in Testing API should produce a human readable report of results.
15. The Built-in Testing API should be configurable to enable or disable testing at runtime.

Tool Selection

Since the challenge of this contest is to develop a built-in automated testing framework, we request that you do not use any third-party unit testing or code coverage tools in your solution. Use of third-party tools to handle other aspects of the problem is acceptable and even encouraged.

Design Architecture and Documentation

With a loose set of requirements, this contest is also meant to be an exercise in design architecture. As each group will approach the problem differently, it is important that all participants provide documentation that will allow the judging panel to understand your solution. Be sure to describe execution details, important directory hierarchy information, location of test scripts, additional software/library dependencies, and other information that might be important for users of your system.

Presentation

All selected finalists are required to prepare a presentation that will be given on Saturday, February 22nd, 2014 at Washington-Lee High School in Arlington, VA as a part of the contest awards ceremony. Presentations should be prepared using Microsoft Power Point or a similar presentation medium, and should focus on how the team has addressed the requirements, problem solution, implementation, execution (demonstration), and their delivery. The presentation will factor into each team's final score, and can be an excellent way to include less technical individuals on a team.

Submissions

Submission should include your solution implement in the Java programming language and any additional libraries or tools that your group used. Your solution should be submitted through the submission section of this website and will be evaluated based upon performance, correctness, maintainability, usability, elegance and various aspects of your delivery. All submissions are due by 11:59pm on Saturday, February 1st, 2014 and must include:

- All source code and compiled classes of your framework
- All source code and compiled classes of the instrumented sample application
- Additional libraries or tools
- Software documentation
- Requirements traceability documentation