BuiltInTester Specifications
By the PHS IDT 2014 Team (phs_winter2014)

Jonathan Ni
Kent Ma
Diwakar Ganesan

Abstract. The purpose of this document is to outline the and usage of the BuiltInTester API, as well as provide documentation for the source code.

1. Introduction

The `BuiltInTester` API was developed to give developers an easy way to debug their programs, and identify possible errors in the code. In this document, we will provide documentation for the different parts of our code, provide examples of usage, and as well as identify common errors.

2. Documentation

A `BuiltinTester` needs to be created as an object in order for it to be used. There are multiple constructors that control if it is enabled, and if it throws exceptions or just gives an error printed with a return code.

```
public BuiltInTester()
```

No-args constructor disables the tester and disables exception throwing.

```
public BuiltInTester(boolean enable)
```

Allows the user to enable the tester, but disables exception throwing.

```
public BuiltInTester(boolean enable, boolean throwsException)
```

Allows the user to enable the tester and exception throwing.

The `expecting` function is used to give the API possible inputs for the function being tested, and outputs that correspond to the inputs. The prototype for this function looks like this:

```
public int expecting(Object inputValue, Object possibleValue,
                  Object expectedOutput, String variableID, String functionID,
                  Class<?> inputType, Class<?> outputType)
                  throws IllegalArgumentException
```

Explanation of Parameters:

`inputValue`

The value that is being tested.

`possibleValue`

A possible value for the variable that is being tested. The tester will only function if the current value of the variable matches one of the possible values given. It can be any type.

`expectedOutput`

The output that matches with the given possibleValue. For example, if the function finds the square root of a number, if possibleValue is 64, expectedOutput would be 8. It can be any type.

`variableID`

The name of the variable being tested. If the variable being tested is called x, this parameter's value should be "x". Acts as an ID for the variable.

`functionID`

A unique string to identify the function the tested code resided in.

`inputType / outputType`

These parameters are used to keep track of the datatype of the inputs and outputs. For example, if the input is an integer, and the output is an array of booleans, `inputType` would be `Integer.class`, and `outputType` would be `Boolean[].class`. If the input type is a user defined object `Foo`, then the `inputType` should be `Foo.class`. With this, the `Object` parameter can be casted to the right type and use the `equals()` function to check for equality. When passing primitives, it is important to remember to pass the `Object` type rather than the primitive type (use `Integer.class` rather than `int.class`)

Returns:

0 on success, 1 on failure if not throwing exception, 2 on not enabled.

Throws:

`IllegalArgumentException` if failure and throwing exception.

In order to keep track of all this data, the use of hashmaps is employed. There are 3 hashmaps that are used throughout the API to organize the data being fed into the system. These are: `expectedValues`, `givenValues`, and `variableResidences. expectedValues` maps a `String` to an `ArrayList` of `Outputs`, which references to all of the possible outputs to a function. `Output` is a class that was created as a part of the API to aid in organizing data. See below for documentation regarding this class. Because one function has many possible outputs, `ArrayList` was used instead of a reference to a single object. `givenValues` maps a `String` to an Object, which references to the current value of the input to the function. `variableResidences` maps a `String` to a another `String`, which holds the function name.

```
public static boolean arrayEquals(Object array1, Object array2,
                    Class<?> type)
```

Explanation of Parameters:

`array1`

One of the objects being compared.

`array2`

The other object being compared.

`type`

The component data type of the objects.

Returns:

true if the arrays are equal in value.

While developing this program, we realized it would be tremendously useful if our API was able to support multidimensional arrays. Since we do not have control over how many dimensions the user is working with, checking for equality would be difficult iteratively. Therefore, we decided to implement a recursively defined function that checks to see if two objects are equal using the Java Reflection API. The function works with any data type, and with any dimensioned array. To check for equality with multidimensional arrays, `arrayEquals` uses recursion, with each iteration removing one dimension of the array. After the parameters are no longer arrays (the base case), it checks for equality amongst them. It is important to note that this method will work even if the input is not an array.

The `log` function is arguably the most important one in our API. It works to check for equality amongst the actual output (the value that was logged) and the expected output that was given in the expected function.

```
public int log(String variableID, Object actualOutput)
```

Explanation of Parameters:

`variableID`

The string ID of the variable being tested. Must match the string ID given in the expecting functions.

actualOutput

The actual output of the function (the value being logged).

Returns:

0 if success, 1 if error if not throwing exception, 2 if not enabled, -1 if logged variable not found.

Throws:

IllegalArgumentException if failure and throwing exception.

First, log iterates through all the possible outputs for a given input. It then fetches the current value of the variable (stored in the givenValues hashmap), and the input data type. Then, a while loop is used to isolate the component type of the input and output data types. If they are not arrays, nothing happens. Then, the code checks for equality amongst the possible value for input (possibleInputValArr), and the actual input value (givenValArr) using the arrayEquals function described above. According to the specifications, we were only supposed to print if the input to the function matched one of the possible inputs. When a match is found, it uses the StringBuilder class to create string representations of the data, to make the report readable by humans. Then, using the arrayEquals function, it checks for equality amongst the expected output, and the actual output. Depending on the result, the appropriate string is logged using the logInternal function.

---

logInternal is a very simple function which creates a human-readable report and gives it to the Logger class, which prints it.

The next feature discussed is the Output class. It was created to aid in the organization of the data inputted into the API. One Output object contains a reference to a possible input value, an output associated with that input, and the data types of the respective objects. It contains simple getter and setter methods to access the objects.

An additional feature to our tester API is the profiling application. API users can start, lap, and stop time profiling for a section of code.

```
public long startProfile(String codeID)
```

Explanation of Parameters:

codeID

The ID associated with the section of code being profiled. This will be used later when the profiling is stopped or lapped.

Returns:

The current time in nanoseconds, or -2 if not enabled.

Starts profiling of a certain code segment. This records the current time in nanoseconds in a hashmap and returns it.

```
public long lapProfile(String codeID)
```

Explanation of Parameters:

`codeID`

The ID associated with the section of code being profiled. This will be used later when the profiling is stopped or lapped.

Returns:

The time since `startProfile` was called with the `codeID`, in nanoseconds, -1 if failure, or -2 if not enabled.

Throws:

`IllegalArgumentException` if failure and throwing exception.

Laps profiling of a certain code segment. This records the current time in nanoseconds in a hashmap and returns the time elapsed since the last `startProfile` call with the `codeID`. This does not delete the entry in the hashmap.

```
public long stopProfile(String codeID)
```

Explanation of Parameters:

`codeID`

The ID associated with the section of code being profiled.

Returns:

The time since `startProfile` was called with the `codeID`, in nanoseconds, -1 if failure, or -2 if not enabled.

Like `lapProfile`, except removes the entry after obtaining the time value.

```
public static void globalEnable()
public static void globalDisable()
```

Enables/disables `BuiltInTester` as a whole -- all `BuiltInTester` objects.
Nota bene: if `globalEnable()` is called, some testers may still be disabled because the instance disable takes precedence over the global enable.

```
public void enable()
public void disable()
```

Set the enable state of individual `BuiltInTester` objects.

A table of when an instance `BuiltInTester` is enabled is given below:

	GLOBAL enable	GLOBAL disable
INSTANCE enable	enabled	disabled
INSTANCE disable	disabled	disabled

3. Example Usage

A. Scalars:

```
/* Trivial example: function that prints 2 */
public void print2()
{
        BuiltInTester tester = new BuiltInTester(true); // Initializes + enables the tester
        int numberToPrint = 3;
        /* Setting up the basic test */
        tester.expecting(numberToPrint,3,2,“numberToPrint”,
                        “public int print2(String)”,Integer.class,Integer.class);
        /*
         * BUG BELOW:
         * numberToPrint isn’t 2, so the function has the wrong output.
         */
        System.out.println(numberToPrint);
        tester.log(“numberToPrint”, numberToPrint);
}
```

Sample output:

```
3
-----
[I] Variable numberToPrint FAILED in function public void print2(String) with input value [3]; Actual
Output: [3 ] ---- Expecting Output: [2]
```

In this simple example, the function `print2` is supposed to print “2” by printing `numberToPrint`. The value of `numberToPrint` was given to `tester.expecting`’s `possibleValue` argument, so it *is* considered a valid input. However, since it doesn’t match `expectedOutput`‘s given value, the test fails. This example outlines the simplest use of `BuiltInTester`: to check that variables are their expected values.

It is important to note that the number logged into the system is the actual output of the function.

B. Arrays

```
/*
 * Counts from 1 to 5
 */
public void count_to_5()
{
        BuiltInTester tester = new BuiltInTester(true); // Initializes + enables the tester
        int numberList[] = {1, 2, 3, 4, 4};
        /* Setting up the basic test */
        tester.expecting(numberList,new int[]{1,2,3,4,4},
                         new int[]{1,2,3,4,5},“numberList”,
                         “public void count_to_5()”,Integer.class,Integer.class);
        /*
         * BUG BELOW:
         * numberList doesn’t count from 1 to 5, so the function prints “1,2,3,4,4” instead of
         * “1,2,3,4,5”
         */
        for (int i: numberList) {
                System.out.print(i + “ “);
        }
        tester.log(“numberList”, numberList);
}
```

Sample output:

```
1 2 3 4 4
-----
[I] Variable numberList FAILED in function ‘public void count_to_5()’ with value [1 2 3 4 4]. Logged
Output: [1 2 3 4 4] --- Expected Output: [1 2 3 4 5]
```

The example is supposed to count from 1 to 5 by reading off of array `numberList`. However, as the array contains `{1,2,3,4,4}` instead of `{1,2,3,4,5}`, the function prints the wrong values, which is caught by `expecting()`.

C. Enable/Disable Testing

```
public void testFunction()
{
  BuiltInTester tester = new BuiltInTester(true);
  tester.disable();
  System.out.println(tester.startProfile(“testFunction()”));
  tester.enable();
  System.out.println(tester.startProfile(“testFunction()”));
  BuiltInTester.globalDisable();
  System.out.println(tester.startProfile(“testFunction()”));
  tester.disable();
  System.out.println(tester.startProfile(“testFunction()”));
}
```

Sample output:

```
-2
1434912662688514
-2
```

```
-2
```

The example alternates between enabling and disabling the tester and all testers. In the tester, there are two boolean, one static belonging to all testers, and one instance belonging to the one tester. The global boolean is enabled by default. When the instance boolean is disabled and the global boolean is enabled, the instance boolean takes precedence and it prints an error, -2. When both are enabled, it prints the current time in nanoseconds. When the instance boolean is enabled and the global boolean is disabled, the global boolean takes precedence and it prints an error, -2. When both are disabled, it prints an error, -2. See the enabling functions in the documentation above for more information.

4. Common Errors and Troubleshooting

a. Incorrect data type

If the programmer does not pass the Object data type (ie `int.class`) then the API will not function properly. If exceptions are enabled, then the expecting function will throw `IllegalArgumentException` (see above). If they are not, then the expecting function will return 1. To fix, simply pass the object type of the primitive.

b. `ERROR: Could not process menu option - class not found - invocation target exception`

This error is specific to the Framework given to us by IDT. When trying to incorporate our API into the classes given to us, this is a very common error. Usually occurs when the data type that you passed into the expecting function does not match the data type that was passed to expecting. Check to see if the inputs are properly casted to the right data type.

c. `[E] Profiler does not contain codeID` or `Exception in thread "main" java.lang.IllegalArgumentException: Profiler does not contain codeID`

This error occurs when the user attempts to stop or lap the profiler with an ID that was not previously initialized with `startProfile`.

d. `[E] Primitive type passed instead of Object type` or `Exception in thread "main" java.lang.IllegalArgumentException: Primitive type passed instead of Object type`

See (a) for more info.

e. `[E] Trying to log variableID that was not added with expecting function` or `Exception in thread "main"java.lang.IllegalArgumentException:`

```
Trying to log variableID that was not added with expecting function
```

This error occurs when the user attempts to log an object into the API that was not a part of the list of expected input/output combos. To fix, check to make sure the input of the function was previously given to the API through the expecting function.