Proposed Automated Testing Framework for Concurrent Software (Part B)

by

Team M1

Greig Cairns

Junaid Shakoor

Mohamad Amine Belabbes

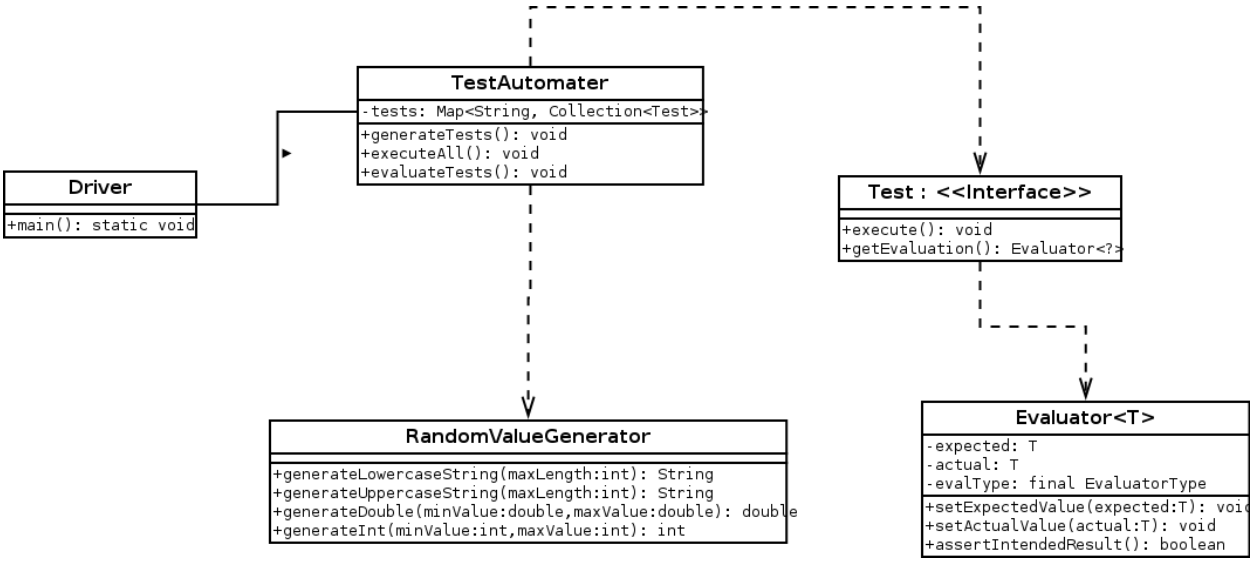
Liam Hourston

William McDonald

Table of Contents

1. Introduction to Proposed Framework

The proposed framework, in theory, layouts the groundwork for a set of modules that work together to detect bug patters allowing developers to recognize and write better tests, validate, instrument. Following is a high-level view of the Proposed Framework.



The above diagram represents a high-level view of the proposed framework system for concurrent software testing. The parts being thoroughly implemented contain the tester module and the evaluator module. Together these two modules communicate with the driver module for system controlling. Following are some detailed description of each module, the types of bugs tested, and their final evaluation.

1.1 Driver Module

This module acts as the access point to the system and allows setting up of all resources. Essentially, this module is called Driver because it controls the setup, execution for all the modules in the proposed system, and retrieves the evaluation results after testing.

1.2 Tester Module

The proposed testing framework structure resembles as the figure above where different types of testing bugs implement the TEST interface which are being used by the TestGenerator and TestExecutor classes. This module, namely TestAutomater, is responsible for generating tests for all the test types we have in the Tests module. It then creates multiple instances to have a higher probability of catching bug patterns. This allows multiple category of bugs, as described in section 2.1 below, to be injected and tested with this structure.

The setup process also includes random generation of values which are passed as parameters to the test instances. The random generator class allows multiple types of values to be generated ranging from upper

and lower case strings, ints, and doubles. This custom-made library's methods require the range we want the value to come from and in other cases the maximum length of the string.

The tester module then goes ahead with the execution of ever generated test instance in a sequential loop. The final responsibility of the tester module is to generate the evaluated results that it obtains from the evaluation module which exists inside every test type's instance and shows the result to the user.

1.4 Evaluator Module

The Evaluator Module has been constructed to be manually linked to the tests packages. The Evaluator has a generic state that just acts as a mediator for storing the expected and the actual results for comparison in a later stage of the processing. This module has a generic Type <T> that is defined at the time of its creation by every test class. One of the parameters the Evaluator expects for comparing is the custom defined *ASSERT_EQUALS* and *ASSERT_NOT_EQUALS* enumerators. Based on the comparison it is requested to do by the user, it then evaluates if the test was passed or not.

The Evaluator then takes in all the testing that it has been asked to do throughout the system and then produces statistics based on tailor made specifics given by the user. At the end of execution, we can see based on the statistics on how many test samples ran and cross checked with the expected values.

2. Evaluation Results

2.1 Testing Types and Categories

Following are the testing types and categories we injected into our system.

Note: The following code with examples may be subjected to slight changes and modification to work in accordance with the automated test generator module. Therefore, the example screenshots cited below may not contain all the information that is required for it to run as part of the system.

2.1.1 Resource Locking Tests

The wrong lock test is used to determine if two threads that operate at the same time are interfering with each other due to not properly locking certain shared resources as they both use a different type of lock that does not prevent the other from accessing it.

TEST 1: Wrong Lock Pattern

In our test for Wrong Lock pattern, the thread 'dr' is using the deposit() method on CurrentAccount ts1 inherited from account which uses the balanceLock reentrant lock as shown in figure 1.1 and thread 'wr' uses the withdraw() as defined in CurrentAccount method which uses the otherLock reentrant lock as seen in figure 1.1.1; they do not effectively lock the balance resource, meaning that there are occasions in operation where these locks can prevent the correct final balance from being calculated properly as the threads have acted in parallel rather than in series and one threaded lock condition has changed the balance while the other has been using it.

```
public void deposit(double amount) {
    System.out.println("Depositing " + amount + " in " + name + "...");
    balanceLock.lock();
    try {
        this.balance += amount;
        getBalance();
        fundsAvailableCondition.signalAll();
    } finally {
        balanceLock.unlock();
    }
}
```

(*figure 1.1 A)*

```
public boolean withdraw(double amount) {
    System.out.println("Balance before withdraw = " + this.getBalance());
    System.out.println("Withdrawing " + amount + " from " + name + "...");
     ReentrantLock otherLock = new ReentrantLock();
     otherLock.lock();
    try {
        this.balance -= amount;
        getBalance();
        return true;

    } catch (Exception e) {
        return false;
    } finally {
         otherLock.unlock();
        System.out.println("Balance after withdraw = " + this.getBalance());
    }
}
```

(*figure 1.1 B)*

The evaluation that determines whether the bug has occurred or not is done by comparing the expected value to the actual value of the balance. The expected value is calculated as follows:

Starting Balance + Money In - Money Out

This value is compared to the generated outcome of running two threads, one which uses deposit() on a CurrentAccount and one that uses withdraw(). If it is the same then the test passes, if not then something has gone wrong in the execution of the code, possibly the wrong lock bug.

```
Initial balance of h is 0.0.
Balance before deposit = 0.0
Balance before withdraw = 0.0
Depositing 10.0 in h...
Withdrawing 2.0 from h...
Balance after withdraw = 8.0
Balance after deposit = 8.0
Expected final balance = 8.0
Actual final balance = 8.0
```

(*figure 1.2)*

Output appears as shown in figure 1.2, this can be used to determine where a bug has occurred in operation if one does. In our testing of the code however we have found it hard to find occurrences of the wrong lock bug, after thousands of runs the expected and final balances were the same in all but a very few number of cases, which are impossible to locate through just the terminal but can be seen in figure 1.3.

```
======New test: 99999 generated=====
Initial balance of owrjiieng is 0.0.
Balance before deposit = 0.0
Depositing 11.0 in owrjiieng...
Balance before withdraw = 0.0
Balance after deposit = 11.0
Withdrawing 1.0 from owrjiieng...
Balance after withdraw = 10.0
Expected final balance = 10.0
Actual final balance = 10.0
============================
============================
Number of tests passed: 1
Total number of tests: 100000
Percentage passed: 0.001%
```

(*figure 1.3)*

Any chance of this bug possibly occurring could be removed by altering the code of the withdraw method as seen in figure 1.4 so that it used balanceLock instead of otherLock.

```
public boolean withdraw(double amount) {
    System.out.println("Balance before withdraw = " + this.getBalance());
    System.out.println("Withdrawing " + amount + " from " + name + "...");
    balanceLock.lock();
    //ReentrantLock otherLock = new ReentrantLock();
    //otherLock.lock();
    try {
        this.balance -= amount;
        getBalance();
        return true;

    } catch (Exception e) {
        return false;
    } finally {
        balanceLock.unlock();
        //otherLock.unlock();
        System.out.println("Balance after withdraw = " + this.getBalance());
    }
}
```

(*figure 1.4)*

TEST 2: No Lock Pattern

In our test 2 for No Lock pattern, there's a thread pool of a few number of threads consisting of two types: A thread that TopUpLoader increments the accountBalance with a synchronized block and the other thread FreeLoader that increments the accountBalance without a synchronized block. Both threads have a counter to check how many times they ran.

```
private Runnable freeLoader = new Runnable() {
    @Override
    public void run() {
        while (run) {
            synchronized(this){     //calculating every Run
                times1++;
            }
                accountBalance++;   //NON-Synchronized Operation
        }
    }
};

private Runnable topUpLoader = new Runnable() {
    @Override
    public void run() {
        while (run) {
            synchronized(this){     //calculating every Run
                times2++;
            }
            synchronized(this){
                accountBalance++;   //Synchronized Operation
            }
        }
    }
};
```

(figure 1.5)

At the end of the sample run we check that the counters from both those threads incrementing the starting accountBalance by one unit plus the starting balance itself would be our expected balance.

Starting AccountBalance + no. times TopUpLoader runs + no. times FreeLoader runs

Our expected account balance should be equal to the summation of these three fields and should not have any discrepancy between the two fields. If so, therefore we have encountered the no lock pattern. However, one of our test runs obtained the following results

```
TopLoader -> did: 37728.0
TopLoader -> got: 37728.0
TopLoader -> did: 37729.0
TopLoader -> got: 37729.0
TopLoader -> did: 37730.0
TopLoader -> got: 37730.0
TopLoader -> did: 37731.0
TopLoader -> got: 37731.0
TopLoader -> did: 37732.0
TopLoader -> got: 37732.0
TopLoader -> did: 37733.0
TopLoader -> got: 37733.0
TopLoader -> did: 37734.0
TopLoader -> got: 37734.0
TopLoader -> did: 37735.0
TopLoader -> got: 37735.0
FreeLoader-> did: 37727.0
FreeLoader-> got: 37736.0
FreeLoader-> did: 37737.0
FreeLoader-> got: 37737.0
FreeLoader-> did: 37738.0
TopLoader -> did: 37736.0
Expected Balance:37743 Balance Obtained: 37737
```

(figure 1.6)

However, fixing our bug pattern means we have to mutate the code into working according to how a standard, thread-safe, code structure should work. Therefore, we'll be surrounding all resources with a proper locking mechanism and ensure that the resources are locked and the operations perform when the resources become available. Following is an example of Runnable threads with mutated code and their respective output:

```
    private Runnable freeLoader = new Runnable() {
        @Override
        public void run() {
            while (run) {
                synchronized(this){     //calculating every Run
                    times1++;
                }
                synchronized(this){     // MUTATED to Perform
                    accountBalance++;   //Synchronized Operation
                }
            }
        }
    };

    private Runnable topUpLoader = new Runnable() {
        @Override
        public void run() {
            while (run) {
                synchronized(this){     //calculating every Run
                    times2++;
                }
                synchronized(this){
                    accountBalance++;   //Synchronized Operation
                }
            }
        }
    };
```

(figure 1.3)

```
FreeLoader-> got: 1733.0
FreeLoader-> did: 1734.0
FreeLoader-> got: 1734.0
FreeLoader-> did: 1735.0
FreeLoader-> got: 1735.0
FreeLoader-> did: 1736.0
FreeLoader-> got: 1736.0
FreeLoader-> did: 1737.0
FreeLoader-> got: 1737.0
FreeLoader-> did: 1738.0
FreeLoader-> got: 1738.0
FreeLoader-> did: 1739.0
FreeLoader-> got: 1739.0
FreeLoader-> did: 1740.0
FreeLoader-> got: 1740.0
FreeLoader-> did: 1741.0
TopLoader -> did: 1281.0
Expected Balance:1741 Balance Obtained: 1741
```

(figure 1.4)

2.1.2 Non-Atomic Operation Tests

With this bug pattern injection, two different non-atomic operations are used to find instances of the bug. The approach with both operations is to create and execute two main threads and then use an executor service to run multiple 'worker threads', in a JAVA environment, to have a higher probability for the bug pattern to take place.

Test 1:

In our **test 1** for this category (Figure 2.2 – A, B) we're using the x++ non-atomic operation as our example. The code keeps track of the amount of premium and standard accounts opened over a period of time. Two threads, one for each account type, will add to the total amount of each. when each thread is executed, a total amount of accounts opened is also executed. In theory, the total amount of premium/standard accounts should be equal to the total accounts opened but from the below test output this is not the case, proving the bug occurrence.

```
1  package tests;
2
3  import java.util.concurrent.ExecutorService;
5
6  public class NonAtomic {
7      private static int openedTotal = 0;
8      private static int standardAcc = 0;
9      private static int premiumAcc = 0;
10     private boolean run = true;
11
12     public NonAtomic() {
13         ExecutorService ThreadGroups = Executors.newFixedThreadPool(1000);
14         ThreadGroups.execute(standardAccount);
15         ThreadGroups.execute(premiumAccount);
16
17         try {
18             Thread.sleep(1000);
19         } catch (InterruptedException ex) {
20             System.out.println("Error");
21         }
22         run = false;
23         try {
24             Thread.sleep(1000);
25         } catch (InterruptedException ex) {
26             System.out.println("Error");
27         }
28     }
29
30     private Runnable standardAccount = new Runnable() {
31         @Override
32         public void run() {
33             while (run) {
34                 openedTotal++;
35                 standardAcc++;
36                 System.out.println("Account #" + standardAcc + " Standard account Total: " + standardAcc);
37             }
38         }
39     };
40     private Runnable premiumAccount = new Runnable() {
41         @Override
42         public void run() {
43             while (run) {
44                 openedTotal++;
45                 premiumAcc++;
46                 System.out.println("Account #" + premiumAcc + " Premium Account Total: " + premiumAcc);
47             }
48         }
49     };
50     public static void main(String[] args) {
51         new NonAtomic();
52         System.out.println();
53         System.out.println("Standard Account Total = " + standardAcc);
54         System.out.println("Premium Account Total =  " + premiumAcc);
55         System.out.println();
56         System.out.println("Total Accounts Opened = " + (standardAcc + premiumAcc));
57         System.out.println();
58         System.out.println("The total amount of registered openings " + openedTotal);
59         System.out.println();
60         System.out.println("Expected output: Total Accounts opened == Total amount of registered openings," + "\n"
61                 + "if the registered openings is < than the total opened then the nonatomic bug is found ");
62     }
63 }
```

(figure 2.2)

```
Account #15901 Premium Account Total: 15901
Account #15902 Premium Account Total: 15902
Account #15903 Premium Account Total: 15903
Account #15904 Premium Account Total: 15904
Account #14719 Standard account Total: 14719

Standard Account Total = 14719
Premium Account Total =  15904

Total Accounts Opened = 30623

The total amount of registered openings 30608

Expected output: Total Accounts opened == Total amount of registered openings,
if the registered openings is < than the total opened then the nonatomic bug is found
```

(figure 2.3)

In keeping with the theoretical use of a banking system, this would mean that in this test. A number of accounts either premium or standard were not registered opened.

Test 2:

In our test 2 (figure 2.4, 2.5) we're using the x+= non-atomic operation, the structure remains relatively the same. Two bank accounts that implement the += operator for adding to the account balance. The amount added to the balance each time the += operation is executed is a fixed amount. A total balance for the two accounts is executed withe same fixed amount each time either of the threads is executed.

```
12
13      public NonAtomic_2() {
14          ExecutorService ThreadGroups = Executors.newFixedThreadPool(1000);
15          ThreadGroups.execute(balanceAcc1);
16          ThreadGroups.execute(balanceAcc2);
17
18          try {
19              Thread.sleep(1000);
20          } catch (InterruptedException ex) {
21              System.out.println("Error");
22          }
23          run = false;
24          try {
25              Thread.sleep(1000);
26          } catch (InterruptedException ex) {
27              System.out.println("Error");
28          }
29      }
30      private Runnable balanceAcc1 = new Runnable() {
31          @Override
32          public void run() {
33              while (run) {
34                  balance1 += amount;
35                  totalBalance += amount;
36                  System.out.println("Account #" + balance1 + " Standard account Total: " + totalBalance);
37              }
38          }
39      };
40      private Runnable balanceAcc2 = new Runnable() {
41          @Override
42          public void run() {
43              while (run) {
44                  balance2 += amount;
45                  totalBalance += amount;
46                  System.out.println("Account #" + balance2 + " Premium Account Total: " + totalBalance);
47              }
48          }
49      };
50
```

(figure 2.4)

```
Account #17552 Premium Account Total: 34021
Account #17553 Premium Account Total: 34022
Account #17554 Premium Account Total: 34023
Account #16530 Standard account Total: 33750

Account 1 Balance: 16530
Account 2 Balance: 17554

Account 1 & 2 Balance: 34084

Total Balance: 34023

Expected output: Total Balance == Account 1 & 2 Balance,
if the Total Balance is < than Account 1 & 2 Balance, then the nonatomic bug is found
```

(figure 2.5)

In theory, the total balance should be equal to the balance of account1 + account2 but it is clear that this is not the case, once again proving the bug pattern occurrence.

In keeping with the theoretical use of a banking system, this would mean that in this test. It would be possible to not only miss a deposit from a customer into their account, but could also mean that the total amount taken for the bank, for that period of time would also be wrong.

This bug can be resolved by the use of synchronization so that two or more threads are unable to access the same resource simultaneously. Meaning that each time the resources are updated, they are done correctly and none are updated at the same time resulting in only one update for that resource.

```
private Runnable balanceAcc1 = new Runnable() {
    @Override
    public void run() {
        synchronized (balanceAcc1) {
            synchronized (balanceAcc2) {

                while (run) {
                    balance1 += amount;
                    totalBalance += amount;
                    System.out.println("Account #" + balance1 + " Standard account Total: " + totalBalance);
                }
            }
        }
    }
};
private Runnable balanceAcc2 = new Runnable() {
    @Override
    public void run() {
        synchronized (balanceAcc1) {
            synchronized (balanceAcc2) {
                while (run) {
                    balance2 += amount;
                    totalBalance += amount;
                    System.out.println("Account #" + balance2 + " Premium Account Total: " + totalBalance);
                }
            }
        }
    }
};
```

(figure 2.6)

```
Account #30732 Standard account Total: 30732
Account #30733 Standard account Total: 30733
Account #30734 Standard account Total: 30734
Account #30735 Standard account Total: 30735
Account #30736 Standard account Total: 30736

Standard Account Total = 30736
Premium Account Total =  0

Total Accounts Opened = 30736

The total amount of registered openings 30736

Expected output: Total Accounts opened == Total amount of registered openings,
if the registered openings is < than the total opened then the nonatomic bug is found
```

(figure 2.7)

```
    private Runnable standardAccount = new Runnable() {
        @Override
        public void run() {
            synchronized (standardAccount) {
                synchronized (premiumAccount) {

                    while (run) {
                        openedTotal++;
                        standardAcc++;
                        System.out.println("Account #" + standardAcc + " Standard account Total: " + standardAcc);
                    }
                }
            }
        }
    };
    private Runnable premiumAccount = new Runnable() {
        @Override
        public void run() {
            synchronized (standardAccount) {
                synchronized (premiumAccount) {
                    while (run) {
                        openedTotal++;
                        premiumAcc++;
                        System.out.println("Account #" + premiumAcc + " Premium Account Total: " + premiumAcc);
                    }
                }
            }
        }
    };
```

(figure 2.8)

```
Account #33548 Standard account Total: 33548
Account #33549 Standard account Total: 33549
Account #33550 Standard account Total: 33550
Account #33551 Standard account Total: 33551
Account #33552 Standard account Total: 33552

Account 1 Balance: 33552
Account 2 Balance: 0

Account 1 & 2 Balance: 33552

Total Balance: 33552

Expected output: Total Balance == Account 1 & 2 Balance,
if the Total Balance is < than Account 1 & 2 Balance, then the nonatomic bug is found
```

(figure 2.9)

2.1.3 Notify Operation Tests

In our test for a lost notify bug we created two runnables, *notifyRunnable (figure 3.1)* and *notifiedRunnable (figure 3.2)*, each one adds a string to the arraylist *strings* when run. These were each run by a separate thread, *notifiedRunnable* will only add it's string to *strings* if it is notified by *notifyRunnable* within 5 seconds.

```
private Runnable notifyRunnable = new Runnable() {
    public void run() {
        lock.lock();
        strings.add(String1);
        System.out.println("String 1 Added");
            lock.unlock();
    }
};
```

(figure 3.1)

```
private Runnable notifiedRunnable = new Runnable() {

    public void run() {

        lock.lock();
        final long TIMEOUT = 5;
        try {
            boolean notifySent = notifySentCondition.await(TIMEOUT, TimeUnit.SECONDS);
            if (notifySent) {
                strings.add(String2);
                System.out.println("Notify received");
            }
        } catch (InterruptedException e) {
        } finally {
            lock.unlock();
        }
    }
};
```

(figure 3.2)

The generated *string's* array list is compared to another which was generated from the same strings to evaluate whether or not String2 has been added due to the notify being received.

```
======New test: 9 generated=====
String 1 Added
```

(figure 3.3)

Output can be seen in figure 3.3 showing a running where the notify was not received and string 2 was not correctly added to *strings* as a result, occurs on every running of the test. Without the notify being present then the test will pass every time, showing that the search for the lost notify bug has been successful. To prevent this the code could be altered as shown in figure 3.4. While this does not entirely prevent the lost notify bug it does allow for some of the notifies to be picked

up by the *notifyRunnable* and for the bug not to occur on some runs, we cannot remove it entirely due to the interleaving.

```
private Runnable notifyRunnable = new Runnable() {
    public void run() {
        lock.lock();
        strings.add(String1);
        try {
            notifySentCondition.signalAll();
        } catch (Exception e) {
        } finally {
            lock.unlock();
        }
    }
};
```

(figure 3.4)

2.1.4 Sleep Operation Tests

Using methods such as sleep() or yield() forces the change of order of concurrent events and therefore, increase the probability of finding concurrent bugs.

In the following example, the method move on is supposed to update the saving account to true when the age of the customer becomes greater than 20. We first start by creating threadGroup '*g1'* and '*g2'*. The threadGroup '*g1'* uses the main threadGroup as its parents while '*g2'* using '*g1'* as its parent. We create an object of type Bank called *'med'* and pass three arguments to the constructor which are: A string for surname, a Boolean for the current account and the age. Every object of type Bank has the Boolean value for the saving account initialized as false as standard.

We use two methods from the class Bank to illustrate the sleep() bug pattern: moveOn() which checks if the age is greater than 20; if so, the *Saving Account* becomes false. The second method is supposed to update the fields surname, current account (Boolean) value and age.

Furthermore, we place moveOn() in the parent group and update in the child group. First, we would like to update and then call moveOn() to check the new value of the age. So, we add a call to the thread to sleep a certain amount of time (0.10 seconds) just enough to let the child thread update the fields. We have two printing statements supposed to print the details of the customer, one before printing the thread details and one after. Both should show the same output when there is no bug:

```
Surname: Belabbes
Age: 22
Current Account: true
Saving Account: true
java.lang.ThreadGroup[name=main,maxpri=10]
java.lang.ThreadGroup[name=Mine,maxpri=10]
Surname: Belabbes
Age: 22
Current Account: true
Saving Account: true
```

(*figure 4.1)*

However, some interleaving can happen when the bug is introduced such as:

```
Surname: Belabbes
Age: 22
Current Account: true
Saving Account: false
java.lang.ThreadGroup[name=main,maxpri=10]
java.lang.ThreadGroup[name=Mine,maxpri=10]
Surname: Belabbes
Age: 22
Current Account: true
Saving Account: true
```

(*figure 4.2*)

```
Surname: Belabbes
Age: 22
Current Account: true
Saving Account: false
java.lang.ThreadGroup[name=main,maxpri=10]
java.lang.ThreadGroup[name=Mine,maxpri=10]
Surname: Belabbes
Age: 22
Current Account: true
Saving Account: false
```

(*figure 4.3*)

Therefore, we can assume that we have encountered the bug pattern due to the sleep method calling in a concurrent environment.

```
public static void main(String args[]) {

    ThreadGroup g1 = new ThreadGroup (Thread.currentThread().getThreadGroup(), "Mine");
    ThreadGroup g2 = new ThreadGroup (g1, "subGroup");

    Bank med = new Bank("Bel abbes", true, 20);

    Thread a = new Thread(g1, "T2") {
        public void run() {
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                System.out.println("Something went wrong");
            }
            med.moveOn();
        }
    };
    Thread b = new Thread(g2, "T2") {
        public void run() {
            med.update("Belabbes", true, 22);
        }
    };
    a.start();
    b.start();
```

(*figure 4.4*)

However, such bug can be resolved by using wait and notify. In the parent thread, we can introduce a synchronization mechanism and wait for the child thread until it finishes and notify the parent thread to take action. Here, the parent is thread a and thread b is the child thread, we use b.wait to wait for b to finish before taking action.

You

```
public static void main(String args[]) {

    ThreadGroup g1 = new ThreadGroup (Thread.currentThread().getThreadGroup(), "Mine");
    ThreadGroup g2 = new ThreadGroup (g1, "subGroup");

    Bank med = new Bank("Bel abbes", true, 20);
    ThreadB b = new ThreadB(g2, "T2", med);
    b.start();
    synchronized (b) {
        try {
            b.wait();
        } catch (InterruptedException e) {
            System.out.println("Something went wrong");
            }
            med.moveOn();
        };
```

(figure 4.5.1)

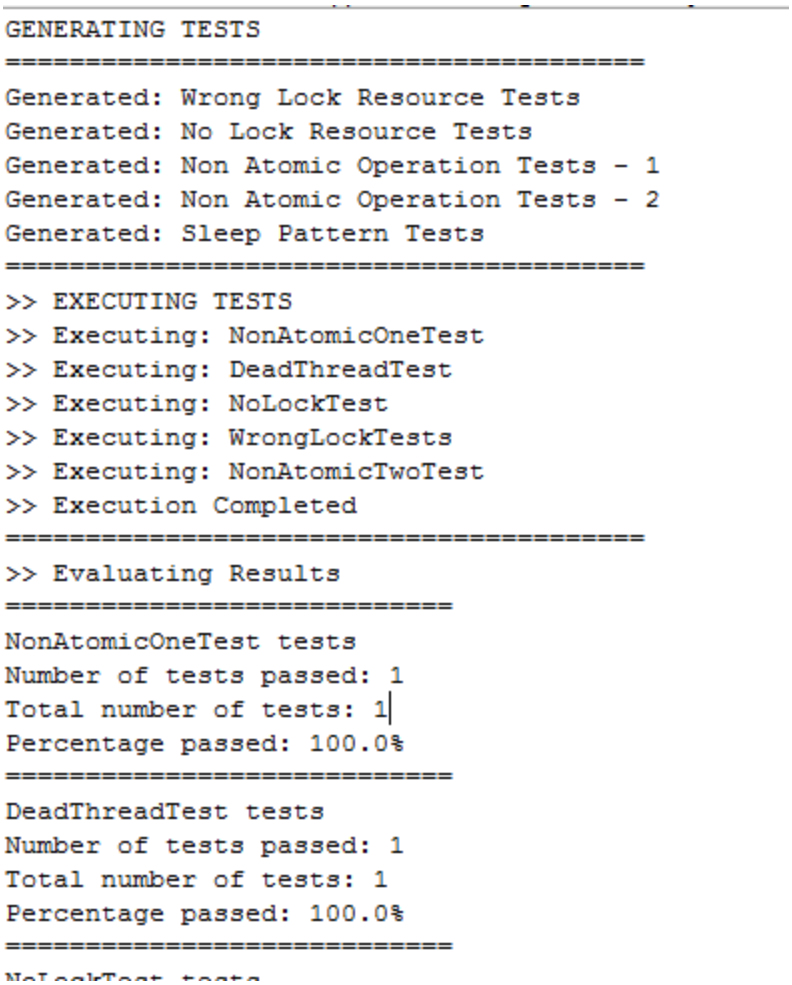
Meanwhile, thread b, we execute our operation update and call notify() so the parent thread a executes its method moveOn();

```
public class ThreadB extends Thread {
    private Bank med;

    public ThreadB(ThreadGroup g2, String string, Bank med) {
        this.med = med;
    }

    public void run() {
        synchronized (this) {
            med.update("Belabbes", true, 22);
            notify();
        }
    }
};
```

(figure 4.5.2)

2.2 Results

Following our detailed analysis of injection and mutation examples, we run all of our tests through the main tester module that has generated and executed all, and now the system pulls all evaluation results to show us all the tests that were executed and their pass and fail statistics. The system keeps an active count on the number of tests ran and their success rate as well as their percentage pass criteria. The following image shows us a glimpse of what to expect while the program is running.



(figure 4.6)

This allows the user to have an easier to understand high-level output of the stats without having to go into much detail. The evaluation module is constructed as such to accommodate command line as well as a graphical interface if built and would work seamlessly.