SWE 437-001

Software Testing and Maintenance

Assignment 4: Junit

CLI (Command Line Interface)

*Students:*

Behrad Behmardi

Pisith Yim

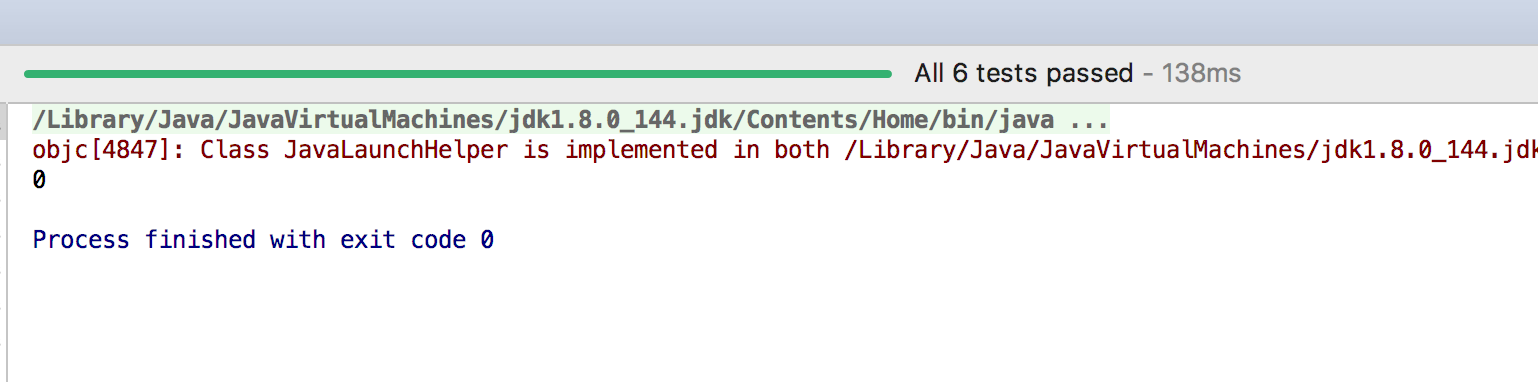
*Professor:*

Dr. Jeff Offutt

Git Repository: http://github.com/behmardibehrad/SWE-437-001-Assignment-1.git

**Part I**

**A. Screen Shots verifying the functionality:**

****

Junit test cases (Full test class is included in another attachment)  
  
*/\*\*  
 \* Test if search is found  
 \* Should pass if the keyword is found  
 \* It has low observability, especially if we have to search from a database with hundreds of records  
 \* It's easier to control the input by changing the input in the test  
 \*/*@Test  
**public void** testSearchNotEmpty(){  
 QuoteList searchedList = **quoteList**.search(**"Behrad"**, 0);  
 *assertFalse*(Arrays.*asList*(searchedList).isEmpty());  
}  
  
*/\*\*  
 \* Test if search is not found  
 \* Should pass if nothing return from the search  
 \* It has low observability, especially if we have to search from a database with hundreds of records  
 \* It's easier to control the input by changing the input in the test  
 \*/*@Test  
**public void** testSearchForEmpty(){  
 QuoteList emptyList = **quoteList**.search(**"homeefeojo"**, 0);  
 *assertFalse*(emptyList.getSize() != 0);  
}  
  
*/\*\*  
 \* Test for nullpointerexecption when passing null argument  
 \* Should pass if null argument passed in  
 \* It has low observability, especially if we have to search from a database with hundreds of records  
 \* It's easier to control the input by changing the input in the test  
 \*/*@Test(expected = NullPointerException.**class**)  
**public void** testSearchNullString(){  
 QuoteList nullString = **quoteList**.search(**null**, 0);  
 System.***out***.println(nullString);  
}  
  
*/\*\*  
 \* Test for nullpointerexception is the list is null  
 \* Should pass if the list is null  
 \* It has low observability, especially if we have to search from a database with hundreds of records  
 \* It's easier to control the input by changing the input in the test  
 \*/*@Test(expected = NullPointerException.**class**)  
**public void** testSearchNullList(){  
 **quoteList** = **null**;  
 QuoteList invalidArgs = **quoteList**.search(**"Berad"**, 2);  
 System.***out***.println(invalidArgs);  
}  
  
*/\*\*  
 \* Test if wrong mode argument was passed in  
 \* List should be empty accoding to search function  
 \* Therefore should pass if list size if 0  
 \* It has low observability, especially if we have to search from a database with hundreds of records  
 \* It's easier to control the input by changing the input in the test  
 \*/*@Test  
**public void** testSearchInvalidMode(){  
 QuoteList invalidMode = **quoteList**.search(**"Berad"**, 6);  
 *assertEquals*(0, invalidMode.getSize());  
}  
  
*/\*\*  
 \* Test with string object or empty string  
 \* Since any quotes contain empty string somewhere  
 \* This should return all the quotes back  
 \* It has low observability, especially if we have to search from a database with hundreds of records  
 \* It's easier to control the input by changing the input in the test  
 \*/*@Test  
**public void** testSearchInvalidArgs(){  
 String objStr = **new** String();  
 System.***out***.println(objStr.length());  
 QuoteList objArg= **quoteList**.search(objStr, 1);  
 *assertEquals*(10, objArg.getSize());  
}

**B.**

**Observability:**

Since all of our tests are tying to access the xml or in a real world would be a database of some sort, they all have low observability. Although, in this case the xml contains only small amount of data, but in the real world, we have a huge amount of data. Accessing a large database can be a very difficult task, the tester or programmer might have limited access or resources to the system, or even worse. Furthermore, the connection between remote resources are not always consistent, therefore it’s much harder to observe the behavior of the output or test.

**Controllability:**

It’s easy to control the inputs of our tests, because we don’t need to get different inputs from sensors or modules. Most our inputs and behaviors are from users. We can try to mimic possible input that users might enter to cover all of our tests. Programmers can also use different assertions to cover all of the test scenarios.

**Part II**

@Test  
**public void** testRemoveDuplicates()  
{  
 **words**.add (**"cookie"**);  
 **words**.add (**"cake"**);  
 **words**.add (**"cake"**);  
 **words**.add (**"pie"**);  
 **words**.removeDuplicates();  
 *assertTrue* (**"removeDuplicates method"**, **words**.getFirst().equals (**"cookie"**));  
}

**A.**

removeDuplicates should retain the first instance, and remove subsequent duplicate instances of the string. Using RIPR or fault/failure model to describe the flaw in our test method above. Based on the definition in the book, the RIPR model applies even when the fault is missing code (Amman & Offutt 20).

Reachability: The fault must be reached in order to properly test our method. In the case of the test method above, we didn’t reach our fault, since all the program does is removing the duplicated term which is *cake*. Then we pursued to check the first element in the list, which is *cookie*, the assertion passes because the fault didn’t reach.

Infection: After the location is executed, the state of the program must be incorrect (Amman & Offutt 20). In this case the state of the program is not infected yet, because the fault never reached. Remove the *cake* string didn’t change the fact the first element is still *cookie*.

Propagation: The infected state must propagate through the rest of the execution and cause osme output or final state of the program to be incorrect (Amman & Offutt 20). In our test method, our fault has not been affected through out the list yet, plus the assertion method also passes since the first element is still cookie, therefore it has not propagated throughout yet.

Revealability: Tester must observe part of the incorrect portion of the final program state (Amman & Offutt 20). Although, the test passes, the tester will never reveal the incorrect state of the list, since fault never reached, incorrect state never found, and incorrect behavior discovered.

**B: Improved testRemoveDuplicates method**

@Test  
**public void** testRemoveDuplicates()  
{  
 **words**.add (**"cake"**);  
 **words**.add (**"cake"**);  
 **words**.add (**"cookie"**);  
 **words**.add (**"pie"**);  
 **words**.removeDuplicates();  
 *assertTrue* (**"removeDuplicates method"**, **words**.getFirst().equals (**"cake"**));  
}