HelpDesk4

Starter code: **AmicaJavaTraining/Projects/HelpDesk/HelpDesk3**

Submit as: **CompletedWork/HelpDesk/HelpDesk4**

In this exercise you will build JUnit tests for some classes in the HelpDesk application. Make a local copy of the starter code and create a new Maven project in the **HelpDesk3** folder. The POM already includes JUnit, Hamcrest, and Mockito, and is configured so that Maven's Surefire plugin will recognize JUnit-5 annotations.

HelpDeskTest

1. Open the starter source file **HelpDeskTest.java**, found in **src/test/java** in the package **com.amica.help**. See that it defines a bunch of constant values, and a custom matcher **HasIDs**. The matcher will make it easy to check the results of any of the **HelpDesk** query methods that returns a **Stream<Ticket>** by asserting that the tickets in the stream have the expected IDs, in the right sequence.
2. Give the class a field **helpDesk** and initialize to a new **HelpDesk** object.
3. Many of the test methods will need to prime the help desk with a few tickets, in order to drive behavior and check results. Add a method **createTicket1** to the test class, taking no parameters and returning nothing. Implement it to call **helpDesk.createTicket**, passing **TICKET1_ORIGINATOR**, **TICKET1_DESCRIPTION**, and **TICKET1_PRIORITY**.
4. Add a method **createTicket2**, along the same lines and using the **TICKET2_*** set of constants.
5. Write a test method that calls each of your helpers to create two tickets in the help desk, and then call **helpDesk.getTicketByID**, passing **TICKET2_ID.** Assert that the description of the returned ticket equals **TICKET2_DESCRIPTION**. (You could check all of the ticket state, but this is enough to identify it, and we're going to get deeper into how the ticket is assigned in other test methods.)
6. Run your test class now, and see that we have an early problem: we can't just add tickets to the help desk. We need to set up technicians first.

7. Add a **@BeforeEach** method that calls **helpDesk.addTechnician**, passing **TECH1** for both the ID and name, and any integer for the extension. Call the method two more times, using values **TECH2** and **TECH3**.
8. At the end of this same method, add a call to **Clock.setTime**, passing 100. This will support creation of tickets and their history events, even though we're not going to worry about particular dates and times in this test class.
9. Run your test again and it should now succeed. You can run after any of the upcoming test methods, too – just keep checking that tests are being passed as expected.
10. Write a test to see that, if you call **helpDesk.getTicketByID** before adding any tickets, you get a **null** result.
11. Let's go back and cover the case we tripped across before adding technicians. That is a correct behavior, on the part of **HelpDesk**, and we should prove that out with a test method that expects the exception. Create a method that calls **createTicket** on a new **HelpDesk** instance (used just for this test method), and assert that the **IllegalStateException** is thrown.
12. Let's look at how tickets are assigned to technicians. To do this, we're going to need references to the **Technician** objects held by the help desk.
13. Add fields **tech1**, **tech2**, and **tech3** to the test class, each of type **Technician**.
14. Since we don't create these, and there is no **getTechnicianByID** method, we'll have to rely on **getTechnicians**, and grab the individual references from that. Add code to the **@BeforeEach** method to call **helpDesk.getTechnicians** and then call **iterator** on that. Capture the results as a local variable **iterator**, of type **Iterator<Technician>**.
15. Set **tech1**, **tech2**, and **tech3** each to the results of a call to **iterator.next**, in sequence.
16. Add a test method that creates ticket #1, and then asserts that, if we get that ticket by ID from the help desk, the associated technician is equal to **tech1**. Also assert that **tech1.getActiveTickets** has a count of 1L. (Stream counts are long integers.)
17. Write another test method that creates both ticket #1 and #2, and asserts that ticket #2 is assigned to **tech2**.

18. Now we'll look at the query methods. Write a test method that creates both tickets, and then gets ticket #2 by ID and resolves it. (Pass any text you like as the resolution reason.) Then assert that exactly one ticket is returned if you call **helpDesk.getTicketsByStatus**, passing **Status.ASSIGNED**, and one if you call with **Status.RESOLVED**.

19. Let's take a different approach to checking the "not-status" method. In a new test method, create both tickets and call **helpDesk.getTicketByNotStatus**, passing **Status.WAITING**. This should give you a stream with both tickets, and because ticket #2 has a higher priority than ticket #1, they should be streamed in that order – 2, then 1. Here's where the prepared matcher comes in: you can **assertThat** the stream returned from that method call **hasIDs(2, 1)**. The custom matcher iterates through the stream for you and checks the ID of each ticket.

20. When that test is passing, try tweaking the expected IDs – more or fewer of them, different order – and see the error messages that appear in the test report. You might want to review the code in **HasIDs** to see how these messages are assembled.

21. Develop at least one meaningful test for each of the other query methods that returns a stream of tickets: finding by tags, technician, and text.

    Tests for latest activity, ticket history, and other help-desk behaviors are optional. Note that many of these will be covered in the next test class, where we use the full test data set from earlier exercises in a sort of integration scenario.

HelpDeskScenarioTest

22. There's a lot of good code in the **TestProgram**, but it is not a JUnit test. Let's build a second test for **HelpDesk** that taps into all of that data and the rich sets of assertions that are based on it. Start by creating a new class **HelpDeskScenarioTest**.
23. Give the class a field **helpDesk** of type **HelpDesk**, but don't initialize it yet.
24. Also define a field **tags** of type **Tags**. Notice too that while everything in **TestProgram** is **static**, we're working with instance variables and instance methods in this class.
25. Copy over the helper methods **addNote**, **suspend**, **resume**, and **resolve**. These will all work as written, once you remove the **static** keyword.
26. Create a **@BeforeEach** method. Copy in the code that initializes **tags** to a new **Tags** object; sets synonyms into **tags**; initializes **helpDesk** to a new **HelpDesk** based on **tags**; and adds the four technicians to **helpDesk**.
27. Copy in the code that sets up all of the ticket data.
28. Now … the original program didn't have Hamcrest at its disposal, so it built a few helper methods of its own. Rather than reworking all of the calls to those methods to use Hamcrest, we'll keep the methods as adapters that turn around and call the Hamcrest assertions. Start with **assertTrue**: this can call **assertThat**, and pass **error** (the custom error message) first, and then **condition**, and finally the matcher **equalTo(true)**.
29. **assertEqual** does something similar: the first parameter to **assertThat** will be the call to **String.format** with the **error** message and **actual** value, and then **actual** and **equalTo(expected)**.
30. **assertCount** and **getCount** can be copied over verbatim.
31. Create a @**Test** method **test1_Tickets**, and copy in the code from the corresponding method in **TestProgram**. You can get rid of the **System.out.println** calls, as we'll get a much more useful test report from the JUnit test runner, and console output during the run of a unit test can be distracting.
32. Run your test class and see that it succeeds.
33. Convert the remaining test methods, each to a **@Test** method, and run.

    Notice that the order is not predictable, as it was from **TestProgram.main** – but, thanks to JUnit, the test don't need to be run in a specific order anymore.

TicketTest

34. Open the starter source file **TicketTest.java**, and see that a few things have been put in place for you: various constants for test data; a target **ticket** reference; a custom matcher for **Event**s and a helper method **assertHasEvent** that checks to see that an event with certain expected values is found at an expected index in the ticket's history; and a **@BeforeEach** method that sets the synthetic clock and initializes the **ticket**.
35. Create a **@Test** method that asserts that all of the **ticket**'s fields were initialized as expected: ID, originator, description, and priority as provided to the constructor; and the rest as they should be initialized: status to **Status.CREATED**, no technician or tags yet.
36. In this same method, call **assertHasEvent** and pass 0, **Status.CREATED**, and "Created ticket.".
37. Run this first test and see that the class passes the test.
38. Create a second test method that proves out the **Ticket**'s **compareTo** implementation. Since the code under test relies first on priority, in descending order, you can **assertThat** the prepared ticket is **lessThan** a ticket that you create on the fly with a lower priority, and **greaterThan** a new ticket with a higher priority. You can check two tickets with the same priority, too, and they should be ordered in ascending order by ID. Run the test class again and see that both cases are passed (and make a habit of running your class each time you complete a new test method from here on out).
39. Write a test method that calls **Clock.setTime** to a new, later time, creates a **Technician** object and passes it to **ticket.assign**. Assert that the ticket's state is correctly modified as a result.

    It would be nice to assert that the ticket makes the expected call to **assignTicket** on the given technician. The best we can do with an actual **Technician** object is to check the size of the **activeTickets** collection – which is not bad, but it does make our test dependent on logic in that class, which might be wrong or might change over time.

40. So, instead of creating a “real” technician object, call Mockito’s **mock** method, passing **Technician.class**. Stub out the **getID** and **getName** methods, and also **toString**, because the ticket will call these when generating its assignment event.

41. With that in place, after asserting correct state for the ticket, you can call Mockito’s **verify** method, passing the mock technician, and on that call **assignTicket**, passing **ticket**. When you next run your tests, this code will assert that the expected call was made, and with the expected ticket object as an argument.

42. A useful helper function at this point would be **passOneMinute**, which would call **Clock.getTime**, add 60,000 milliseconds to it, and call **Clock.setTime** with the new value. You can then more easily bump the clock forward before each new event in a ticket’s lifecycle, which in turn guards against false positives when checking the timestamps on events because they will be expected to be distinct.

43. Break out the code in your assignment test that sets the clock, creates the mock technician, and assigns it to the ticket, to a helper function, and call it from your test.

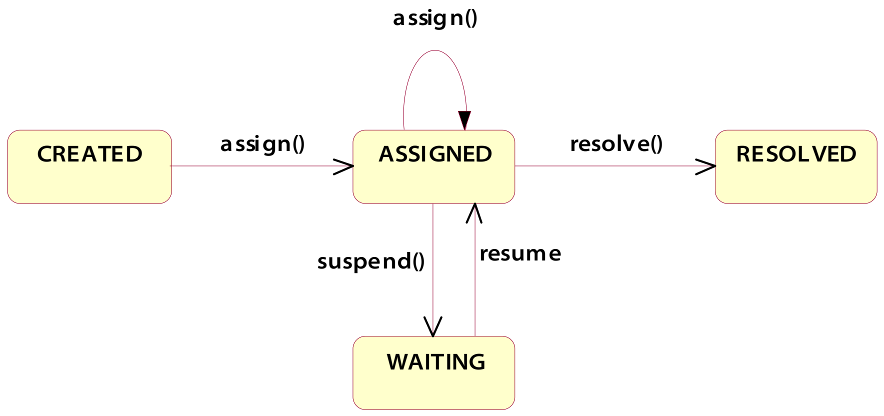
44. Create a new test method for resolving a ticket. It can start by calling your new helper function, because we can’t test resolution without first assigning the ticket. Then call **resolve** and pass a prepared reason, and assert correct state changes and history.

45. Be sure to **verify** a call to **resolveTicket** on the mock technician, too. This will require that you declare the **technician** reference as a field, so that it can be shared between test methods. Move the code that sets up the mock technician to the **@BeforeEach** method.

46. You’ll need similar test cases (and perhaps similar helper methods) for each of the state transitions**: suspend**, **resume**, and **addNote**. You can work these up now, or skip this step and come back to it after developing some of the other types of tests.

We'll focus for a while now on error handling. **Ticket** is pretty lax right now: it checks for a sensible status before assigning or resolving, but other methods don't check; and none of the code guards against other invalid inputs. Over the next few steps you'll write tests that are failed; improve the code in the target class; and then see it pass the new tests.

47. Add a test method to prove that the constructor will reject null values for originator, description, or priority, by throwing an **IllegalArgumentException**. This test will fail, so add code to the constructor to check for these conditions and to throw the exception.
48. Add similar tests for null technician in **assign** and null reasons or notes in **suspend**, **resume**, **resolve**, and **addNote**. Improve **Ticket** so it passes these tests.
49. Write a test that asserts that a ticket will refuse to be assigned once it's been resolved, instead throwing an **IllegalStateException**. The class should pass this test, right out of the gate.
50. Add similar tests to prove out a strict lifecycle for tickets, as shown here. Only the state transitions diagrammed should be allowed.



The class will fail many of these tests, and you'll add more error-handling code until all of your tests are passed.

51. Add some tests of the tagging feature: prove that the ticket will reflect tags that have been added, and that it won't list the same tag more than once, even if the tag is added repeatedly.
52. Add some tests for calculating time to resolve a ticket. Be sure to test error handling here, too: a ticket that hasn't been resolved should throw a state exception.
53. Add tests for the text-search feature. Check that the ticket can find text in its description or in its events.