HelpDesk4

Starter code: HelpDesk/HelpDesk3

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 "Jupiter" annotations. You'll add classes to **src/test/java**, and make some changes to code in **src/main/java**.

TicketTest

1. Create a class **com.amica.help.TicketTest**, under the **src/test/java** folder.
2. It's a good habit to start out by defining constants that you'll use in your tests, because you'll almost always wind up using a given value at least twice: once in creating an object or calling a method, and once in asserting that it is reflected in a getter or some other observable state. You'll need values for at least a ticket ID (which should be 1), originator name, description, and priority; and various reason and note texts and tag values. A fixed start time for the simulated clock will be useful, too. Also you'll be creating a technician, so ID, name, and extension for the technician will be worthwhile.
3. Give your class a **ticket** field, of type **Ticket**.
4. Create a **@BeforeEach** method, and in it call **Clock.setTime**, so that you'll be able to create tickets and generate events and the time information will be there for timestamps.
5. Now set **ticket** to a new **Ticket** with your defined originator, description, and priority.
6. Create a **@Test** method that simply asserts that the values returned from the ticket's getter methods are correct. Check the values you passed explicitly to the constructor, and also values that should be set automatically, such as ID and status, and there should be no assigned technician, an empty tag set, and one event in the ticket's history. Get this event and assert that it has the expected values.
7. Run your test and see that your first test case is passed.

8. 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).

9. Now, everything may be fine … or it may not. JUnit, for a long time, has insisted on determining the order in which your test methods are run, and it is technically predictable but only if you can run hash algorithms in your head. Otherwise you get what you get, and sometimes that exposes issues of test isolation – which is why they're so stubborn about doing it this way. In this case, if your initialization test ran first, you should be fine. But if it didn't, then the creation of a ticket, and even multiple tickets, for the comparison test will have driven the ID sequence up and the assertion that the ticket's ID is 1 will fail.

   So there is tension between unpredictable test order and the use of a static integer in **Ticket** to generate ticket IDs. For now, let's resolve this by compromising on test order. In more recent versions JUnit has grudgingly started offering an option to force an execution order on tests. Add the **@TestMethodOrder** annotation to the test class, with a value of **MethodOrderer.OrderAnnotation.class**. Add the **@Order** annotation to your first test method, with a value of 1.

   If you run now, you should see any issues with the ticket ID are cleared up. But this approach is brittle, and we'll eventually set it aside for a more robust solution.

10. 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.

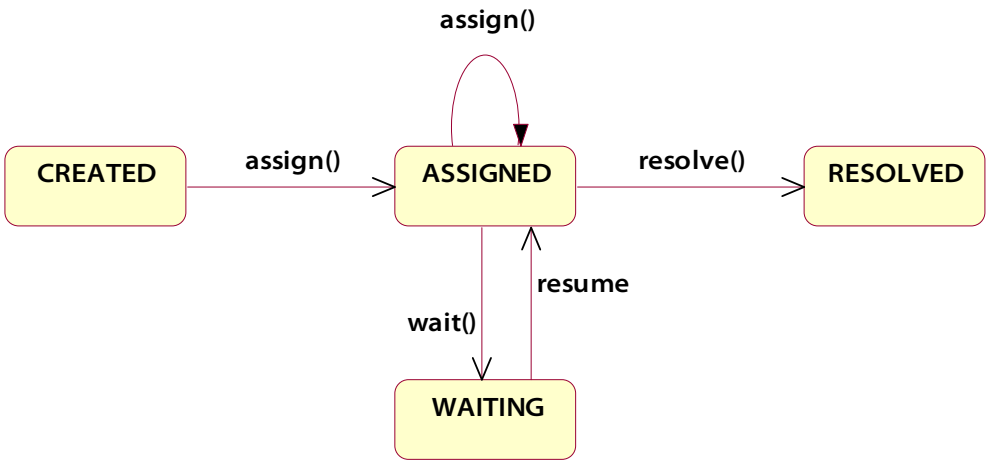
It would be nice to assert that the ticket makes the expected call to **addActiveTicket** on the given technician. The best we can do with an actual **Technician** object is to check the size of the **activeTickets** collection – which is not bad, but it does make our test dependent on logic in that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

11. 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.
12. 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 **addActiveTicket**, 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.
13. 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.
14. You might want to refactor in another way at this point, creating a helper method for asserting the state of an **Event**. You're already doing this for the creation and assignment events, and you will do similarly for resolution, notes, waiting, and resuming. The helper could be a **void** method that carries out several asserts, based on given expected values; or you might develop a custom **Matcher<Event>**.

15. Break out the code in your assignment test that sets the clock, creates the mock technician, and assigns it to the tick, to a helper function, and call it from your test.
16. 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.
17. Be sure to **verify** a call to **removeActiveTicket** 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.
18. You'll need similar test cases (and perhaps similar helper methods) for each of the state transitions: wait, resume, and add a note. 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.
19. 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.
20. Add similar tests for null technician in **assign** and null reasons or notes in **wait**, **resume**, **resolve**, and **addNote**. Improve **Ticket** so it passes these tests.
21. 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.

22. 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.

23. 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.

24. 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.

25. Add tests for the text-search feature. Check that the ticket can find text in its description or in its events. Consider edge cases such as a substring that would span the description and one of the events, or two successive event texts, as these should not be found, but in one possible implementation (that concatenates all of the searchable strings before checking if the substring is found) they might be.

HelpDeskTest

Now you'll create a test for the **HelpDesk** class. We're going to take a very different approach to this one: since there is already so much test logic written, in the **TestProgram** class, you'll essentially convert that class into a JUnit test, factoring various pieces of code into setup and test methods, and eventually creating a nested test dedicated to that program's more elaborate test scenario, so that other test cases can be written based on other pre-conditions.

26. Create the **HelpDeskTest** class, and have **TestProgram** handy for copying in code. Give the test class a field **helpDesk**, and initialize it to a new **HelpDesk** instance.
27. Give it a **@BeforeAll** method and a **@BeforeEach** method. Copy the first few lines of **runSimulation** – that call static methods on **Tag** to set up synonyms and preferred capitalizations – into the **@BeforeAll** method. The rest of **runSimulation** can be copied into the **@BeforeEach** method – so that the whole scenario will be run before each test case, which is a luxury we didn't have when running the original test program.
28. Copy the two static methods **assertThat** and **assertEqual**. This is unconventional, but since we have hundreds of lines of code that already use these helpers, it will be easier to turn them into adapters that call Hamcrest assertions and matchers, than to rework all of the test logic line-by-line.
29. Change **assertThat** to call Hamcrest's **MatcherAssert.assertThat**, passing the **error**, the **condition**, and the **equalTo** matcher to match **true**. (Use this qualified name, and not just **assertThat**, in your code; otherwise you'll be trying to call your own **assertThat** method.)
30. Change **AssertEqual** to call **MatcherAssert.assertThat**, passing **error**, **actual**, and **equalTo(expected)**.
31. Copy the **test1_Tickets** method into the test class. Annotate it as a **@Test**, remove the **static** modifier, and remove the **System.out.println** calls at the top and bottom.
32. If you test now, you'll see a failure, and not in the test method: you get an **IllegalStateException** in **runSimulation**. This is an (intended!) effect of the enhancements you made and tested on the **Ticket** class: the **runSimulation** method tries to do one thing that is not actually a legal state transition, which is to **resolve** a ticket directly that's in a **WAITING** state.

33. To fix this, find the offending call to **resolve**, and split it into a call to **resume** and an immediate call to **resolve**. Split the message up so that we resume with a message about receiving approval and resolve with a message about granting access.
34. You should be able to run the test now, and that first test case should pass.
35. Copy in the second test method, and run the test class again. Whoops! What happens?

    You're hitting the same issue with generated IDs as you did in **TicketTest**: we can't control the value of the next ID from outside of the class, and we can't control the order in which we create tickets, and so their IDs are unpredictable. A lot of our test logic relies on identifying those tickets by their IDs as they're returned in various query methods; so the band-aid approach of enforcing a test order will be hard to support.

    It's time to refactor the code! It really makes more sense for **HelpDesk** to be assigning IDs, anyway. As it is now, we can't have two instances of **HelpDesk** that would each have a clean ID sequence.

    So take a break from test-writing, and change the system in those two classes: remove **nextID** from **Ticket** and define it instead, as an instance variable, on **HelpDesk**. Give the **Ticket** constructor a new, initial parameter **ID**, and use that to initialize the ticket's ID. In **HelpDesk**, the **createTicket** method will increment **nextID** and pass the value in the call to create a **new Ticket**. **ReopenedTicket** will be affected by this change, too, because it will need to take that extra **ID** parameter in its constructor, and pass it along to the superclass constructor; and **reopenTicket** will have to adjust accordingly.
36. You'll need to adjust the code in **TicketTest**, too – and in the process you can drop the ordering annotations from the previous part of the workshop. Just assign specific IDs to your tickets as you create them.
37. Now run your test class. It should be happier now, because (a) the ID sequence restarts with each new instance of **HelpDesk**, (b) you create a new instance for each test, thanks to your **@BeforeEach** method, and (c) each test independently expects that consistent sequence of 14 IDs.
38. Copy in each of the remaining test methods, one at a time, and test as you go. Notice that you are changing the help-desk data in the code for test 7 … and the tests are not guaranteed to run in numerical order. But thanks to JUnit's lifecycle, the whole scenario is re-started before each test method, so this is all right.

You'll add some more test cases, to cover concerns not addressed in the original test program (which you can delete now, by the way, as it won't run cleanly anyway!). To support these other cases, you'll push most of your current logic down into a nested test.

39. Define a public inner class called **TechnicianTests**, and annotate it as **@Nested**.
40. Move the existing **@BeforeEach** method into the inner class. This means that the code here will not run before any tests defined on the outer class, but will still run for the existing, now-nested test cases.
41. Add one test method to the outer class, that proves that the help desk will throw an **IllegalStateException** if you try to create a ticket before any technicians have been set up.
42. Now create another class, **ScenarioTests**, as an inner class of **TechnicianTests** – so a "grandchild" of the outer class **HelpDeskTest**.
43. Give this class its own **@BeforeEach** method. Split the logic in the existing **@BeforeEach** method so that the first few lines, that create the technicians, stay at the **TechnicianTests** level, while the rest of the code moves down to the **ScenarioTests** level.
44. Add a test to the **TechnicianTests** class that proves that a new ticket automatically advances to the **ASSIGNED** state. Create the new **Ticket** with hard-coded, literal strings for now – don't use pre-defined constants.
45. Run your tests. Unless you anticipated this issue and set the simulated clock, you will run into a **NullPointerException** here – because you're trying to create a ticket, which involves generating a time-stamped event, and the only calls to **Clock.setTime** are in the **ScenarioTests** class now. Add a call to set the clock to some legal time in the **@BeforeEach** method of **TechnicianTests**, and the test should pass.
46. Now, try a little experiment. Symbolic constants would be better for this new ticket, and for a bunch of others you'll create in a few more test methods. Add this import to your source file:

```
import static com.amica.help.TicketTest.*;
```

47. Change your new Ticket's arguments to be the constants you defined earlier in **TicketTest** for ID, originator, description, and priority.
48. Run your test. Hmm! Most of your methods are still fine, but test 8 is failing. Why is that? What happened?

JUnit's lifecycle guarantees are great, but there's only so much it can do, especially where statics and class loading are concerned. Here was the road to ruin: by importing symbols from **TicketTest**, you cause that class to be loaded when you run **HelpDeskTest**; this causes all of that class' static variables to be initialized; some of those are **Tag**s, and are therefore added to the global store of tags held as a static in the **Tag** class; and test 8 includes assertions about the size and contents of that store, which are now incorrect.

Sigh. And, in fact, this only serves to alert us to a problem that has been there all along, because when we run **mvn test** – as we inevitably will do when checking in code, or on another machine that does a nightly build, etc. – it may well have loaded **TicketTest** in order to run it, and then run **HelpDeskTest** in the same JVM, and we'd hit the same issue at that point.

The best answers for testing this part of the application would include breaking this code out to a separate **TagTest**, and even a separate class just for this feature of **Tag**, and then assuring that such a test would run in its own JVM.

49. For now, just comment out the lines that call **Tag.getTags** and iterate through it confirming its contents. Your test should again pass
50. Add a few tests for assignment, in the **TechnicianTests** class: a single ticket should go to Andree; a second ticket should go to Boris; or, if the first ticket were immediately to be resolved, then that second ticket would also go to Andree.

Extensions

If you have extra time, try developing tests for **Event**, **Technician**, and **Tag**; or try a test for **ReopenedTicket**. This last one is especially challenging if you try to balance code coverage and reuse of test logic, because a lot of the expected behavior is the same as for **Ticket** and so there's a lot of test logic in **TicketTest** that it's appealing to reuse. But it takes some refactoring of that class to make the test logic flexible enough to work for both classes.