1 Testing Techniques

1.1 Why to write tests?

From https://wiki.openstack.org/wiki/Fuel/How_to_Test_Your_Code

- Testing helps you find errors in your code.
- Testing helps you write better code.
- Writing test cases will save you time later.
- Unit tests provide immediate feedback on your code.
- Test cases document intent.

These are some guidelines for writing unit tests.

1. A testing unit should focus on one tiny bit of functionality and prove it is correct.
2. Each test unit must be fully independent. Each of them must be able to run alone, and also within the test suite, regardless of the order they are called. The implication of this rule is that each test must be loaded with a fresh dataset and may have to do some cleanup afterwards. This is usually handled by setUp() and tearDown() methods.
3. Write both valid and negative tests
4. Try to avoid mocking and stubbing, favoring test parameters or attributes instead. When resorting to mocking and stubbing, only mock against a small, stable, obvious (or documented) API, so stubs are likely to represent reality after future refactoring.
5. Check the state of the system after test thoroughly. Example: when deployment is finished, notification should be created. The right test will check that the only one notification was created, and all fields of it are exactly which we expect. Bad test will filter DB to find any success notifications and will pass if any was found.
6. Test how the code handles exceptions. Think about how will user be notified about an error. It is bad practice to just show "Server error has occurred".
    - Example: test does PUT request with data which break the system inside (somewhere ValueError is raised). Response status code must not be 500.
    - Another example: Due to temporary network connectivity issue agent fails to access Nailgun API.
7. Test wisely. It perhaps has no sense to test if new notification can be created with default params, but it makes a lot of sense to test if valid notification object (with all attrs set) was created on node discovery. What message will be in notification if in POST request to /api/nodes memory is None? Or 0? Or some number less than 1 Gb, will it be rounded correctly?

8. Unit tests are also code. It must be clear, structured, follow DRY and other code standards.

9. If you changed some code unit, take a piece of paper and write down all the other code units that use this one. Go and find tests for each of these units, and check if there are tests for each unit which test interaction between them. If you discovered that there is no test for one of such code units, you must write it, no matter who missed it before.

10. Take a minute and think about what could possibly fail in your code. Or to what you've not payed attention enough. Make yourself a hacker, what data to provide to break your code? Write tests for all of this.

11. When fixing a bug, write failing unit test first. When the bug is fixed, test should pass.

12. Don't mix a few checks in one unit test: use separated tests. Do not write iterators to generate tests.

13. Try hard to make tests that run fast. If one single test needs more than a few millisecond to run, development will be slowed down or the tests will not be run as often as desirable. In some cases, test can't be fast because they need a complex data structure to work on, and this data structure must be loaded every time the test runs. Keep these heavier tests in a separate test suite that is run by some scheduled task, and run all other tests as often as needed.

1.2 Black Box and White Box Testing

Black Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is not known to the tester

White Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester.

Recommend writing black box testing before start the implementation on your code.

1.3 Give Test Cases for the Following Methods

1. Java ArrayList Methods (from Java8 documentation):

   (a) **add**(**E** e)
   Appends the specified element to the end of this list.
   Parameters:
   e - element to be appended to this list
   Returns: true (as specified by Collection.add(E))

   (b) **add**(int index, **E** element)
   Inserts the specified element at the specified position in this list. Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).
   Parameters:
   index - index at which the specified element is to be inserted
   element - element to be inserted
   Throws: **IndexOutOfBoundsException** − if the index is out of range (index $<$ 0 || index $>$ size())

   (c) **clone**()
   Returns a shallow copy of this ArrayList instance. (The elements themselves are not copied.)
   Returns: a clone of this ArrayList instance

2. Java PriorityQueue Methods (from Java8 documentation)

   (a) **offer**(**E** e)
   Inserts the specified element into this priority queue.
   Parameters:
   e - the element to add
   Returns: true (as specified by Queue.offer(E))
   Throws:
   ClassCastException - if the specified element cannot be compared with elements currently in this priority queue according to the priority queue's ordering
   NullPointerException - if the specified element is null

   (b) **poll**()
   Returns: the head of this queue, or null if this queue is empty

   (c) **remove**(**E** e)
   Removes a single instance of the specified element from this queue, if it is present. More formally, removes an element e such that o.equals(e), if this queue contains one or more such elements. Returns true if and only if this queue contained the specified element (or equivalently, if this queue changed as a result of the call).
   Parameters: o - element to be removed from this queue, if present
   Returns: true if this queue changed as a result of the call

3. ExtrinsicMinPQ API (from Heap Homework)

   (a) **getSmallest**()
   Returns the item with smallest priority.

   (b) **removeSmallest**()
   Removes and returns the item with smallest priority.