Mocking Part A - RackMonitor - RackMonitorIncidenceTest

Instructions

Open the ***RackMonitorIncidentTest*** and answer these questions:

1. ***RackMonitorIncidentTest*** defines servers that will be unhealthy (health factor less than 0.8) and shaky (health factor less than 0.9). Since *RackMonitor* doesn't actually calculate the health factor on its own, some other code must be treating these servers specially to guarantee they always return an appropriate health factor. Which class does the special handling?

   Rack.getHealth()

2. Data that exists solely to make a situation where a service will exhibit a known, consistent behavior is called "test data". Why might we need a service to exhibit a known, consistent behavior? (Why must we be sure our test is consistent?)

   So we know the tests are consistent. Test should continue to pass even if the code it is testing changes.

3. Why should we avoid test data in production servers?

   Privacy - should not use real customer information is testing/development
   Safety - might accidently change production data with your tests
   Performance - You might slow down production processing when accessing production data

4. Given the server IDs for the unhealthyServer and noWarrantyServer, how many test servers does Rack probably need to handle? How many different types of servers does Rack probably need to test effectively?

   Need to define 4 different servers for testing: unhealthy, warranty
   unhealthy, non-warranty
   shaky, warranteed
   shaky, non-warranty

If we change ***RackMonitorIncidentTest*** to use mocks, our dependencies won't need to provide test data!

5. Let’s Trace through the code in *RackMonitor* that's used by the *getIncidents_withOneUnhealthyServer_createsOneReplaceIncident* test method.

   Right now we're only focused on the code path used by this test (there may be lines of RackMonitor that we're not testing in this specific test method). You shouldn't need to look at any of the code in any other classes, particularly not WarrantyClient or WingnutClient; just make a reasonable guess about how they work by reading their Javadoc.

   For each method that RackMonitor calls on its **dependencies** (external classes it uses), decide if it should be mocked. Classes we don't normally mock include: POJOs, core pieces of the Java library like Collections (Maps), and side effect code that doesn't contribute to what our service returns. Use the table below to keep track of your decisions; we've filled in a few cells to get you started.

Method	**Expected Input**	**Return Type**	**Mock it?**
Logger.info Logger.warn	String, Object	Logger	No (library)
Rack.getHealth	none	Map<Server, Double>	Yes (external)
Rack.calculateTestHealth	Server	Double	Yes (external)
Server.getServerId	none	String	No (indirect use thru Rack)
RackMonitor.arrangeReplacement	Rack, Server	void	No (internal)
RackMonitor.getUnit	Rack, Server	int	No (internal)
Rack.getUnitForServer	Server	int	Yes (external)
WarrantyClient.getWarrantyForServer	Server	Warranty	Yes (external)

If we change `RackMonitorIncidentTest` to use mocks, we can consolidate all those test servers into a single variable. We can also get rid of the code that sets up any dependency that we mocked.

Let's go do some Mocking!

1. Why didn’t we need code to initialize and configure our dependencies when we use mocks?

2. Were there any dependencies we can use the default behavior of the mock for?

3. Did you notice that using mocks made us set up something that we didn't need to set up when using test data? What was it?

4. Did mocking use *more* code or *less* code than test data?

5. Is the mocking code *more* readable or *less* readable? Which version more directly specifies what the test actually does better?