Variability vs. Repeatability – An Experience Report

Jonathan LI ON WING

jonathan.lionwing@mail.mcgill.ca
jwing@groupon.com

Abstract

Have I been approaching testing all wrong this whole time?

In my earlier years of being a test engineer, I was taught to write tests that are repeatable: given a certain set of parameters and inputs, map out the expected output – repeatable and simple. However with a developer-test ratio of 10 to 1, I was quickly falling behind. I found myself having to decide between spending time on my automation test suite and exploratory testing. The former gave great reliability regarding regression, but the latter caught most of the bugs. The former was repeatable and could run practically 24 hours a day, the latter only when I found time. How about if we implemented some of the exploratory concepts in the automated test suite?

In this paper, I will discuss about my experience in adding variability to test automation and creating an oracle to be able to determine correctness. I will demonstrate that, in my experience, adding randomness to the tests can help quicken delivery time, improve the discovery of bugs, and increase confidence, without compromising repeatability, when compared to traditional automation methods.

Biography

Jonathan Li On Wing is a Software Development Engineer in Test at Groupon in Seattle, WA. He earned his B.Sc. Cum Laude in Software Engineering from McGill University specializing in Artificial Intelligence and Requirements Analysis. He has 9 years of industry experience in various roles working at small and large companies such as Microsoft, Oracle, Expedia, and SMART Technologies.

Jonathan's interests lie in Human-Computer Interaction, Automated Testing and Software Life Cycle Processes. He believes in bringing testing upstream, and has been improving software testing processes in several companies. Jonathan sees himself as a customer advocate and enjoys challenging everything.

1 Introduction & Motivation

The goal of software testing is to determine the correctness and completion of a given product based on its requirements. These requirements come in many forms which include requirement specification and through personal experience. Dr. Cem Kaner defines testing as an “investigation done to provide stakeholders information about quality of a product or a service." This examination is done to provide a level of confidence to stakeholders about the level of quality. However, as Edsger Dijkstra states, “testing can be used to show the presence of bugs, but never to show their absence!” So, if we are to provide a level of confidence, we ought to be careful in how we test, and how we discover bugs.

Throughout most of my academic career and for a part of my professional career, test cases would be scripted with expected inputs leading to expected outputs. They would either be run manually or automated. Initially they were mostly run manually, and as our profession started to mature, I found that we were moving towards having all our test cases automated. Additionally, as the feature code increases, manually testing has a harder time keeping up to cover all old regression testing and new functionality. Moreover, as time progressed, the developer-test ratio became exceedingly more developer heavy. Also, it was more efficient to have a machine run the tests. Unlike me, it can run multiple tests simultaneously and quicker, and seemingly it didn’t complain when asked to work than 40 hours a week.

There is no denying, that when it came to scripted tests, the machine would be more efficient, but how about exploratory tests. It has been discussed at length the benefits of exploratory tests. In my experience, scripted tests were very good at finding regression bugs or acceptance tests, but would not discover additional bugs. While exploratory testing would be pretty good at finding new bugs due to the fact there is a freedom in what we test – we adapt our tests based on our experiences and we vary our inputs and steps.

This leads me to ask: can we automate these tests? Since exploratory tests in one sense, requires human intuition, the quick answer would be no. However, can we take some aspects and use them for automation? Part of exploratory testing is learning and adapting our testing. This seemed quite hard, but definitely something I would like to tackle in the future. Another part is varying our tests. This is the motivation for the paper.

2 Methodology

In this section, we will consider the way I went about adding variability in our test framework.

2.1 Previous Methodology

Typically, we had tests similar to a framework shown in Figure 1, in one form or another. To test a service, the tests would gather the inputs and outputs from data files. The inputs would be used to call the web service and compare the response from the call, to the corresponding output.

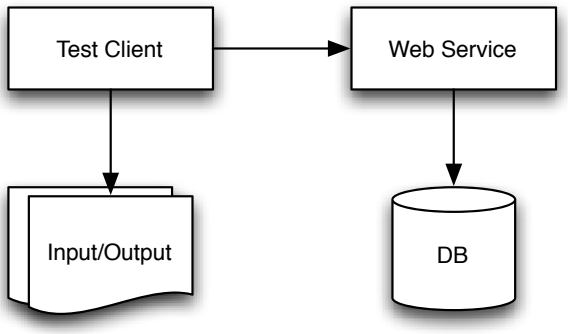

Figure 1

The problem with this methodology is its maintainability. First, the data files would need to be maintained as the data in the database could be changing. Additionally, as features get added, responses may not look exactly the same – this requires changing the data files often.

Another drawback is that these tests would always be testing the same thing over and over again. These tests, once written, found no new bugs. It did, however, grant confidence that with each new build, there were no regressions found.

2.2 Implemented Methodology

Our approach to deal with the aforementioned issues was two-fold.

2.2.1 Method 1

Let us consider a web service that performs a hotel search. A feature may be:

- As a client of the web service, I want to be able to get the details like its location and description for a specific hotel.

These features are straightforward and can be simple fetches to the database. Using the feature above, the methodology used to test it is as follows (a high level view can also be seen in Figure 2):

Algorithm 1 – Testing ability to get details of a hotel
Query database for a random hotel Create an object model from Database response Send hotel to HTTP Client, which will build an http request Send the request, and create an object model for the response Compare the two objects *(Repeat as often as required)*

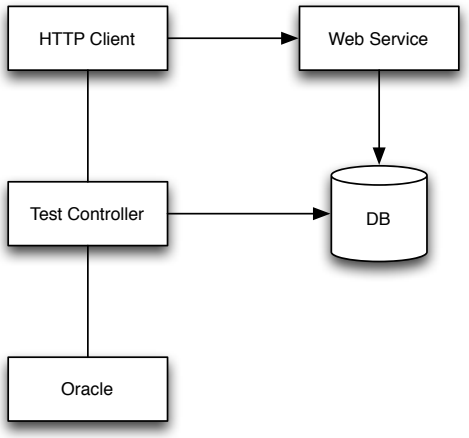

Figure 2

2.2.2 Method 2

Let us consider the same web service, but consider the following feature:

- Given a region[1], return all the active hotels in this region.

In this case, as it is not a simple fetch, we would need a way to validate a response's validity. There are three things that need to be validated:

- Are all the returned hotels in the region (see Algorithm 2);
- Are there any missing hotels (see Algorithm 3);
- Is the information, such as the hotel details, correct? (This can be tested similarly to previous test)

Algorithm 2 – Testing if all hotels returned are in the region

Query database for a random region
Send the region to the HTTP Client
Query the service for all the hotels
Oracle verifies that all hotels are inside region's polygon
(Repeat as often as required)

Algorithm 3 – Testing if there are any missing hotels

Query database for a random region
Query database for all hotels within a bounding box
Send the region to the HTTP Client
Query the service for all the hotels
Oracle verifies that, for each hotel
    Determine whether it should be inside or outside the region
    If inside, ensure it is in the response
    Otherwise, ensure it is not

We can also target our tests so that we can do more in-depth testing. Using the same story, we can come up with test cases such as:

[1] Regions are already created in the database and used production data. They are can be represented as multipolygons.

- Verify that a region that spans across the 180th meridian correctly returns all hotels
- Verify that a very large region returns all the valid hotels

In these cases, when we query the DB, we add the limitations when we search for random regions. That is, we can test with a random region that spans across the 180th meridian.

2.2.3 Logging

These tests will be quite hard to use if at any time the tests can randomly fail. Therefore, we also ensure that for each test run, we log enough information as to what failed, why, and how to reproduce it.

3 Observations

Although it was fun and interesting to build this framework, was it useful and worth it? The following are my observations.

3.1 Time

One of the main concerns is the time it takes to build this framework, and for that we will consider three parts: initial construction, adding new tests, and maintenance.

3.1.1 Initial construction

Custom frameworks always take more time to build than an out-of-the-box data-driven test framework (such as TestNG). As we already had an in-house flexible reporting engine, there was no need to build that part, and thus I would estimate took me a couple weeks longer – I was not working on this project at 100%.

3.1.2 Creating tests

Adding tests could be quite easy, and it varied between being faster and up to 3 times slower. When features could be hard to test by hand, it was simpler and faster, and if they were simple, it could take longer. Test cases generally didn't take longer than a few hours to write and run. Creating tests for more complex features were faster to write – for example, lets look at a scenario involving finding hotels within a region.

In a standard data-driven test approach, I could either create the appropriate data or use production data. Let us assume that the data is already there. I would need to find a nice region to test; usually we chose New York City. To determine which hotels are found in the region we could rely on many third party libraries available. If the code is not yet ready, I could create a response based on my investigation – which included the distance between the hotel and the city center. If the code was ready, I could compare the response with each hotel. Finally, they could be added to the data files. This process could be repeated for each test scenario, such as using a region across the 180th meridian – in other words slow, and to be frank, tedious.

In our approach, it was quite easy. Using the same example, my test case would look similar to the code seen below. Line 3, can be changed to get a random region that spans the 180th meridian if we wanted to add that test.

Pseudocode 1 - Testing if all hotels returned are in the region

```
01 testAllHotelsBelongInRegion() {
02  //Get 10 random regions
03  List<Region> regionsToTestWith = DBWrapper.getRandomRegion(10);
04  foreach (Region region in regionsToTestWith) {
05    Response response = HttpClient.getHotelsInRegion(region);
06    Oracle.assertVerifyHotelsBelongInRegion(response, region);
07  }
08 }
```

Of course, it is quick and easy to add tests when the functions like `getRandomRegion()` and `assertVerifyHotelsBelongInRegion()` are already implemented. However, the former utility function is a fairly trivial SQL query, and for the latter, there are many third-party libraries that can take care of finding if a point is inside a polygon. This works for us, but even if we did not have third-party libraries, in general I find that verifying the correctness of a response from a web service is easier to implement than the feature code. However, yes, this can make creating tests take more time.

As for simpler tests, such as testing whether we can get the details of a hotel, a function could look like the following:

Pseudocode 2 - Testing ability to get details of a hotel

```
01 testGetHotelDetails() {
02  //Get 10 random hotels
03  List<Hotel> hotels = DBWrapper.getRandomHotel(10);
04  foreach (Hotel hotel in hotels) {
05    Response response = HttpClient.getHotels(hotel);
06    Hotel hotelReturned = new Hotel(response);
07    Oracle.assertEquals(hotel, hotelReturned);
08  }
09 }
```

3.1.3 Maintenance

As our expected outputs are not hard coded, the time spent on maintenance is considerably lower. If a new field is created or modified, then we can just change the model class. For some change in functionality, we would just need to change the oracle's function. In general, this effort takes us less time.

Note that in some cases, maintaining tests may not be longer. Consider the story of getting the hotel details. Verifying that the fields are correct and that there are no new additions or missing fields can easily be done using traditional methods, by having a library generate the expected outputs and inputs. In this case, there is no difference in maintenance time.

3.2 Bug Count

As this was implemented with a new project, the way I compare how good this method performs, is by considering the bugs that would not have been caught in a traditional method. As expected, by adding variation, we were able to catch more bugs, albeit just about 10%. However, the majority of additional bugs caught, were of low priority. Of the ones that were caught, most were due to data issues – data we were not expecting – which I suppose is a good thing we were testing with random real data. An example of such issue can be seen in Figure 3, what is considered inside the polygon?

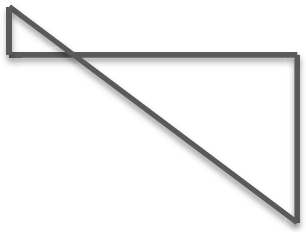

Figure 3

Another issue that was caught by the tests, although traditional test methods may have caught it, is the way distances are calculated. There are a few methods to calculate distance between two coordinates on the globe, as it was not specified which in the requirements, by calculating distances independently, we were able to figure out that we needed to define and be consistent in our distance calculations.

3.3 Confidence

Confidence is a hard metric to measure. Did we provide the right confidence level to our stakeholders? I do not know. As for the test team, we feel more confident. Using the same story example regarding finding hotels in a region, I do not imagine myself verifying the results of more than 3 to 5 regions – not to mention the chance of human error. However, by having randomly chosen 10-50 regions every night, I can be fairly confident that all the regions have been tested, even if there are new ones created.

Additionally, by varying the permutations, having tests run about 12 hours a day, I am pretty sure that is more permutations that I can test in my lifetime whether manually or by traditional automated data-driven tests.

4 Conclusion

As it is close to impossible to determine all the possible use cases a particular feature will be used, it is in my opinion that varying our automated tests was quite beneficial. Not only did they uncover more bugs, they were able to make the automation more maintainable, and it was easier to add new tests. However, even though they did uncover additional bugs, most of these bugs were of lower priority. These additional bugs that were found were mainly edge cases that did not much business value. They did, however spark conversations as to whether these bugs had other implications or whether the design was correct. In general, however, it was deemed more beneficial to know about the bugs and make a conscious decision not to fix them.

On a personal note, it was also much more interesting to add these tests as opposed to modifying and generating XML and JSON responses.

4.1 Comparison of variability versus repeatability

The following is a comparison of the pros and cons of adding variability as opposed to repeatability.

Variability	Repeatability
- Uncover more bugs - Easier to maintain - Generates more confidence - Tests more variation	- Faster to develop initially - Does not require coding skills - Predictable

4.2 Criticism

This work has generated a lot of criticism. The following are the main issues.

4.2.1 Repeatability and predictability

As described in the introduction, one of the main characteristics of a good test case is repeatability – running the tests over and over should yield the same result. How can we trust the tests if they fail one second and pass the next? In most cases, engineers will ignore the failures and just try rerun the tests.

If tests are failing randomly, then usually it means that either there is a data set that exposes a bug in the test code or feature code. It warrants investigation. In the traditional methodologies, we would not have caught the bug, therefore isn't it better to know of a bug?

Note that it is important to keep tests green. So if we ever got a data set that made tests fail, we added a blacklist such that when we got random data from the database, we would ignore those on the blacklist. At times, the blacklist would be a condition, not necessarily identifiers – since the list of ids can be long.

4.2.2 Duplicating feature work

When proposing this idea, or sharing our work, it is often thought that we are just reproducing the feature work. In fact, we are writing code to verify the correctness, which generally is simpler and quicker to write. Although, when it is simple fetches, it may be quite similar. The general concern is: is it worth it? The team can only answer this question, and it definitely will not be suitable for all teams and scenarios. Perhaps a mixture of test approaches?

4.2.3 Testing the tests

One question that arises often is how do you test the tests, and are there any concerns for false positives or false negatives? That is in fact a concern, but not as big as expected based on my experience.

There are two possibilities with the tests, the fail or pass. When a test fails, it could mean that there is a bug in the test code, feature code, or both. After some investigation, we will find out what, and it will be resolved. If a test passes, it means that either there isn't a bug found, or that both feature code and test code are expecting the same results. As they were done independently, and the tests will vary over time, the chances that both consistently passes is minute.

In my experience, it has happened once, but neither traditional data-driven tests, nor exploratory testing would have caught. It took weeks for the developer team to find and fix, and it was due to an assumption both the developers and testers made.

4.3 Future Work

To deal with the repeatability concern, it could be interesting and useful to add a quick and easy way to replay the tests. These can come from replaying based on logs, or storing a test session id with a seed.

Part of exploratory testing, and one of the reasons it is touted as one of the best testing methodologies, is its ability to adapt. I would like to see if I could use machine learning and statistical analysis to have the tests adapt itself and explore areas on its own.

References

TestNG. http://testng.org (accessed June 8, 2015).

Whittaker, James A. *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*. Upper Saddle River, NJ: Addison-Wesley, 2010.

Acknowledgments

I would like to thank all those who helped me review this paper and bounce ideas off: Sean Chitwood, Lory Haack, Yumi Rinta, Robert Sabourin, Devan Samineedi, and Shruti Van Wicklen.

I would also like to thank my wife and son for being patient as I wrote this paper – and always being there.