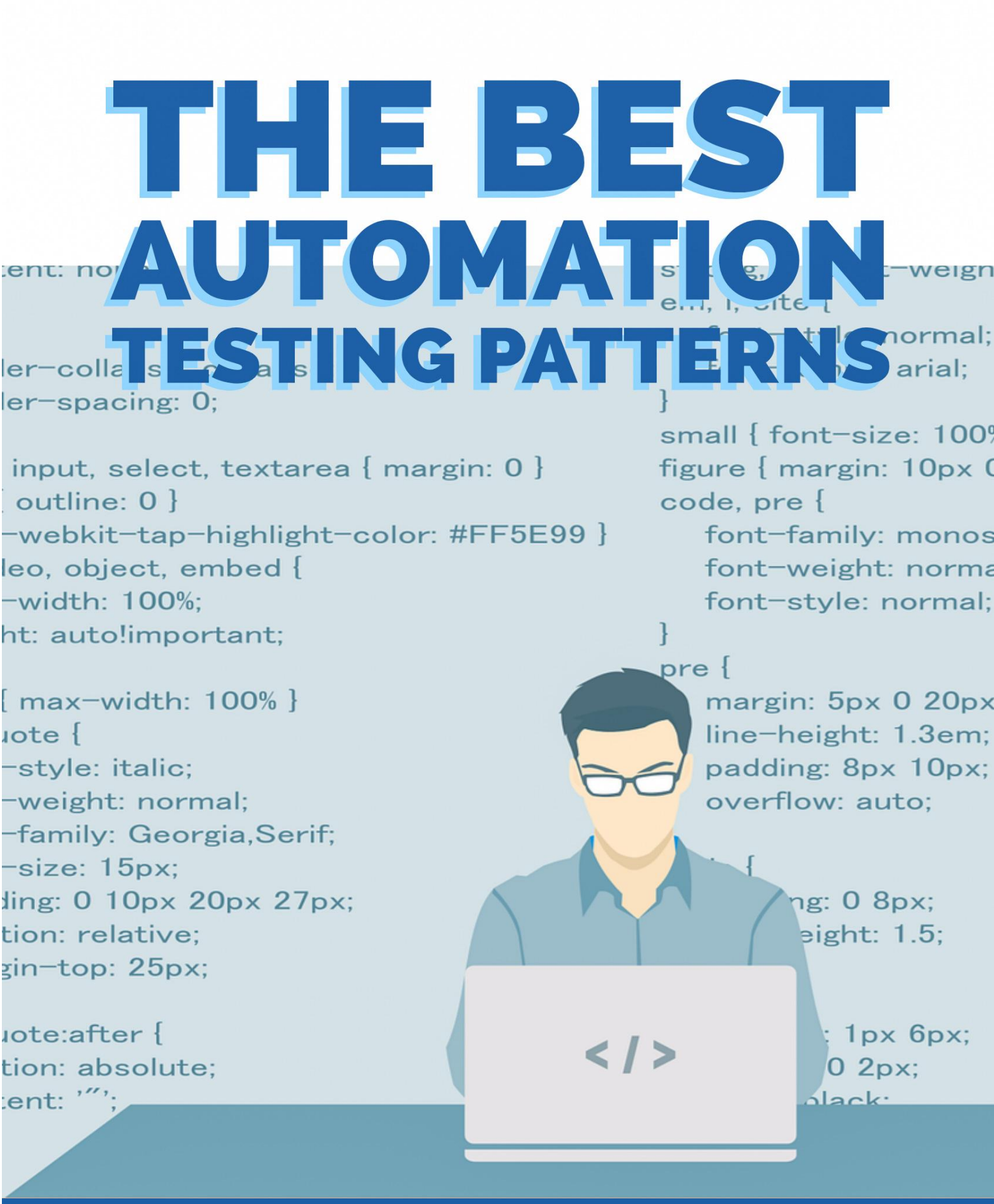
THE BEST AUTOMATION TESTING PATTERNS



ULTIMATE QA

NIKOLAY ADVOLODKIN

The Best Automation Testing Patterns

Nikolay Advolodkin

Ultimate QA

Table of Contents

CHAPTER 1
INTRODUCTION

Test automation is an extremely large and complex topic, wouldn't you agree?

There are so many important pieces of information that can improve your test automation. Hence, I decided to document all the automation patterns and anti-patterns as they develop.

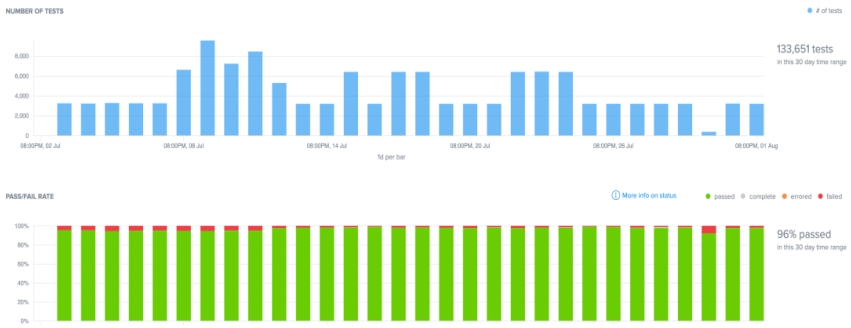
Some items I mention here include how we were able to achieve a 500% reduction in test suite execution time.

Even better:

The pattern that we used to achieve a 98% reduction in test case execution time. The tests went from 76sec to 1.6sec on average.

It gets better:

How does This client execute 30,000 tests a week with a 96% pass rate:



This is what quality automation looks like!

There are many more such lessons learned along the way…

Keep reading…

CHAPTER 2
TEST AUTOMATION PATTERNS

Pattern: Tests Should Be Atomic

An atomic test is one of the automated testing patterns is that testing should be atomic

WHAT IS AN ATOMIC TEST?

An atomic test is one that tests only a single feature. You can usually tell that a test is atomic when:

- The test will only have one assertion or two assertions at most. Because sometimes we need one assertion to make sure our state is correct

- Atomic tests have very few UI interactions and they're only on a maximum of two screens. In rare cases, an atomic test might navigate through 3 screens (although I'd like to see this example)

Here are some examples of atomic GUI tests:

```
[Test]
public void ShouldBeAbleToCheckOutWithItems()
{
    //Arrange
    var overviewPage = new CheckoutOverviewPage(Driver);
    overviewPage.Open();
    //We don't need to actually use th UI to add items to the cart.
    //I'm injecting Javascript to control the state of the cart
    overviewPage.Cart.SetCartState();
    //Act - very few UI interactions
    overviewPage.FinishCheckout().
      IsCheckoutComplete.Should().BeTrue("we finished the checkout process"); //Assert
}
```

```
[Test]
public void ShouldNotBeAbleToLoginWithLockedOutUser()
{
    _loginPage.Open();
    //Although I would likely never test a login through the UI. This is just a small example
    var productsPage = _loginPage.Login("locked_out_user", "secret_sauce");
    productsPage.IsLoaded.Should().BeFalse("we used a locked out user who should not be able to login.");
}
```

PROBLEMS SOLVED BY ATOMIC TESTS-

- Unreliable automated test results
- Slow tests
- Tests that are hard to debug
- Tests that don't provide the correct point of failure

Your automated test should form a single irreducible unit. This means that your tests should be extremely focused and test only a single thing. A single automated test should not do something like end-to-end automation.

> A good rule of thumb that I use on my teams is:
>
> *'Automated acceptance test should not run longer than 1 minute on your local resources'*

If your test runs longer than 1 minute, then you should keep reading on why that might be dangerous. The advantages of writing atomic tests are numerous…

> 500% improvement by running atomic parallel tests.
>
> In a recent case study, we found that 18 long end to end tests ran in ~20 min. Using a much larger suite of 180 atomic tests, with the same exact code coverage, running in parallel, we were able to decrease the entire suite execution time to.
>
> ~4min.

ADVANTAGES OF ATOMIC TESTS

1. ATOMIC TESTS FAIL FAST

First, writing atomic tests allows you to fail fast and fail early. This implies that you will get extremely fast and focused feedback. If you want to check the state of a feature, it will take you no longer than 1 minute.

2. ATOMIC TESTS DECREASE FLAKY BEHAVIOR

Second, writing atomic tests reduces flakiness because it decreases the number of possible breaking points in that test. Flakiness is less of a problem with unit or integration tests. But it is a large problem with acceptance UI automation.

Here's an example:

1. Open UltimateQA.com home page
2. Assert that the page opened
3. Assert that each section on the page exists
4. Open Blog page
5. Search for an article
6. Assert that the article exists

For UI automation, every single step is a chance for something to go wrong…

A locator may have changed, the interaction mechanism may have changed, your synchronization strategy may be broken, and so on.

Therefore, the more steps that you add, the more likely your test is to break and convey false positives.

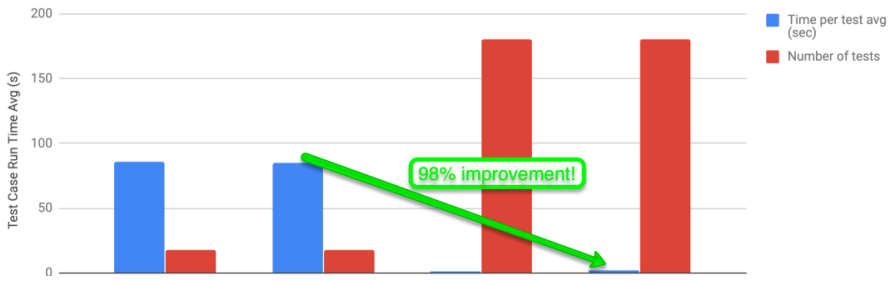
3. ATOMIC TESTS ALLOW FOR BETTER TESTING

The third benefit of writing atomic tests is that if it fails, it will not block other functionality from being tested. For example, the test I mentioned above. If it fails on Step 3, then you might never get to check if the Blog page works or that the Search functionality works. Assuming that you don't have other tests to check this functionality. As a result of a large test, you will reduce your test coverage.

4. ATOMIC TESTS ARE SHORT AND FAST

Finally, another great benefit of writing small tests is that they will run quicker when parallelized…

How I got a 98% performance enhancement in test execution speed with a single change?



98% improvements in average test case execution time having atomic, parallel test.

In the scenario above, I had a suite of 18 end-to-end tests that were NOT atomic and were not running in parallel.

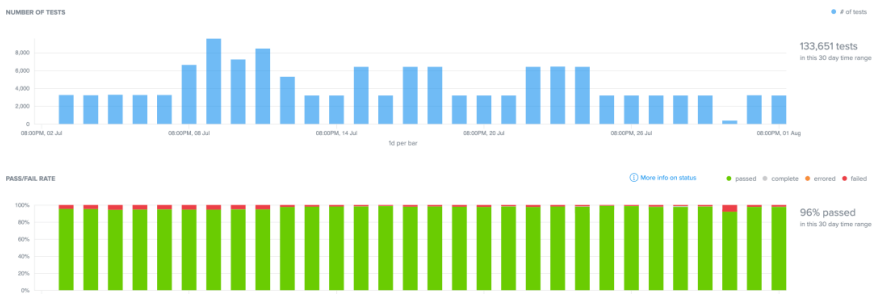
Maintaining the same code coverage, I broke down my tests into 180 tiny, atomic tests…

Ran them in parallel and decreased the average time of test case to 1.76s from 86s!

> 'Your entire automation suite run time will be as fast as your slowest test.'
>
> -Nikolay Advolodkin

By the way, I have seen automated tests that take 30 – 90 minutes to execute. These tests are extremely annoying to run because they take so long. Even worse, I've never seen such a test produce valuable feedback in my entire career. Only false positives.

CASE STUDY OF AMAZING AUTOMATION:



Break down:

- ✓ 30,000 tests/week
- ✓ 96% pass rate
- ✓ Average test run time: 30 sec (In the cloud!)
- ✓ The average number of Selenium commands per test: <30

Ummm, yea. That's what atomic tests will do!

> "Achieving an atomic automated test might be the single Most Powerful way to solve all UI automation problems such as flakiness and slowness. It just boils down to reducing dependency on the UI."

How to break up giant end-to-end UI tests?

Here's how to break giant tests into smaller actionable tests

Okay, you believe me, atomic tests are good.

But how can you break up your large end-to-end tests, right?

Trust me, you're not the only one struggling with this situation…

It gets worse:

On a daily basis, I encounter clients that have the same exact issue.

Furthermore, I wish that I could provide a simple answer to this. But I cannot…

For most individuals, this challenge is one of technology and culture.

However, I will provide a step by step guide to help you have atomic tests.

It won't be easy… But when you achieve it, it will be SO Sweet!

Here is a simple scenario:

1. Open Amazon.com
2. Assert that the page opens
3. Search for an item
4. Assert that item is found
5. Add item to cart
6. Assert that item is added
7. Checkout
8. Assert that checkout is complete

The first problem is that many automation engineers assume that you must do an entire end-to-end flow for this automated test.

Basically, you must complete step 1 before step 2 and so on… Because how can you get to the checkout process without having an item in the cart?

The best practices approach is to be able to inject data to populate the state of the application prior to any UI interactions.

HOW TO MANIPULATE TEST DATA FOR UI AUTOMATION?

1. You can inject data via several options:
2. Using something like a RESTful API to set the application into a specific state
3. Using JavaScript
4. Injecting data into the DB to set the application in a certain state
5. Using cookies

If you can inject data between the seams of the application, then you can isolate each step and test it on it's own

For example:

1. Use an API to send a web request that will generate a user
2. Use an API that will generate an item in your Amazon cart
3. Now you can pull up the UI to the cart page and checkout using web automation
4. Clean up all test data after

This is the best practices approach. You tested the checkout process without using the UI for steps one through three.

Using an API is extremely fast… A web request can execute in like 100 ms.

This means that steps 1,2,4 can take less than one second to execute. The only other step you will need to do is to finish the checkout process.

It gets better:

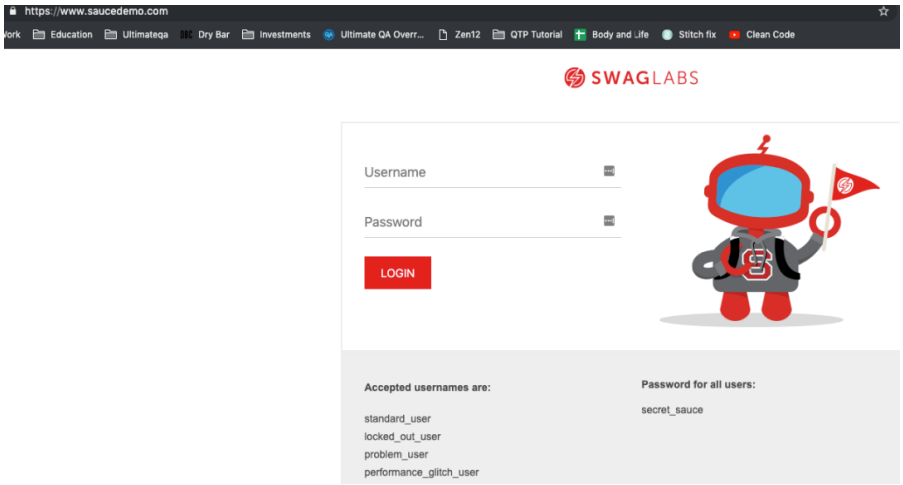
Using an API is much more robust than using a UI for test steps. As a result, you will drastically decrease test flakiness in your UI automation.

ANOTHER EXAMPLE OF CONTROLLING STATE WITH JAVASCRIPT

Probably the most common impediment to atomic testing is the login screen. And most of our apps have one.

So how do we remove this from our test so that our test can be atomic?

Here's one example:



Page with a login screen

1. We execute some JavaScript with our automation framework like this:

```
((IJavaScriptExecutor)_driver).ExecuteScript("window.sessionStorage.setItem('session-username', 'standard-user')");
_driver.Navigate().Refresh();
```

Congratulations, we are now logged in

Now use your UI testing tool to perform the single operation that you want to test.

Here's how a full atomic test would look like:

```
[Test]
public void ShouldBeAbleToCheckOutWithItems()
{
    //Arrange
    var overviewPage = new CheckoutOverviewPage(Driver);
    overviewPage.Open();
    //We don't need to actually use th UI to add items to the cart.
    //I'm injecting Javascript to control the state of the cart
    overviewPage.Cart.SetCartState();
    //Act - very few UI interactions
    overviewPage.FinishCheckout().
      IsCheckoutComplete.Should().BeTrue("we finished the checkout process");
//Assert
}
```

Notice how the test only has one UI action and one assertion…

That's a sign of an atomic test in UI automation.

WHAT IF YOU DON'T HAVE THE CAPABILITY TO INJECT DATA FOR TESTING?

I know that the world isn't perfect and many of us aren't lucky enough to have applications that are developed with testability in mind.

So what can you do?

You have two options:

1. WORK WITH DEVELOPERS TO MAKE APPLICATION MORE TESTABLE

Yes, you should work with the developers to make your application more testable. Not being able to easily test your application is a sign of poor development practices.

This does mean that you will need to leave your cube and communicate across silos to break down communication barriers.

Frankly, this is part of your job. You need to communicate across teams and work together to create a stable product.

> "If the product fails, the whole team fails, not just a specific group."

Again, it's not easy…

I've worked at one company where it took me two years to simply integrate Developers, automation, and manual QA into a single CI pipeline.

It was a weekly grind to have everyone caring about the outcome of the automation suite.

And at the end, our team was stronger and more agile than ever.

Trust me, this is doable and most developers are happy to help. But you must be willing to break down these barriers.

Here's the second option, and you won't want to hear it:

2. IF YOUR APPLICATION IS NOT AUTOMATION FRIENDLY, DON'T AUTOMATE

If you can't work with the developers because you're unwilling…

Or if the company culture doesn't allow you to…

Then just don't automate something that won't provide value. I know that your manager asked you to automate it…

However, we are the automation engineers. We are the professionals.

We must decide what to automate and not to automate based on our understanding of application requirements.

We were hired because of our technical expertise, because of our abilities to say what is possible, what is not possible, and what will help the project to succeed.

Although it might feel easy to say "yes, I will automate your 30 minute scenario", it's not right to do so.

> "If your manager is non-technical, they should not be telling you how to do your job. You don't see managers telling developers how to code. Why is it okay for managers to tell an automation engineer what to automate?"

The answer is it's not okay!

You must be the expert and decide on the correct approach to do your job.

PATTERN: TESTS SHOULD BE FAST

Your automated tests should be fast! I can't stress this enough. Unit and integration tests are already really fast, they run in milliseconds. A unit test can be as fast as one millisecond. So we really don't need to worry about this.

Here's the kicker:

UI tests are really slow because they run in seconds. Thousands of times slower than unit and integration tests!

So it's really important to focus on fast UI tests.

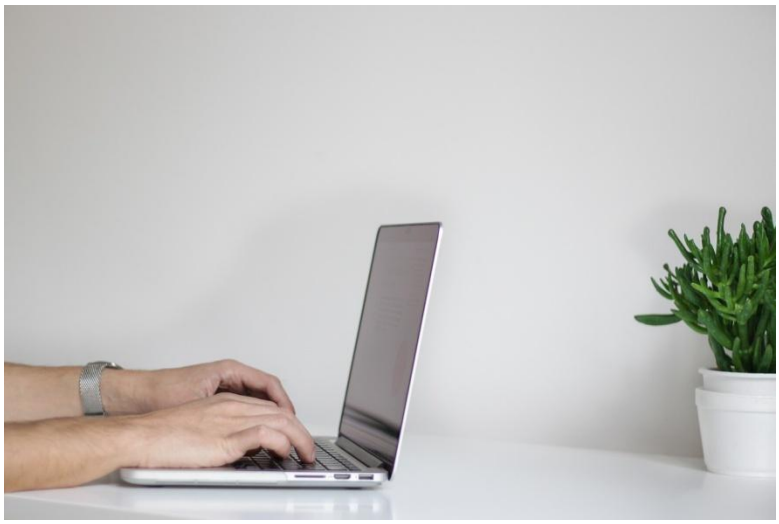
> "Your automated UI tests should take no longer than one minute on your local resources."
>
> "Nikolay Advolodkin, CEO at Ultimate QA

> "Danny McKeown stated in Automation Guild 2019 that his tests run no longer than two minutes."
>
> Danny McKeown, Automation Guild 2019

The reason for this boils down to the fact that you want your builds to be fast. Meaning no more than 20 minutes. Otherwise, nobody will have patience to wait so long for automation feedback.

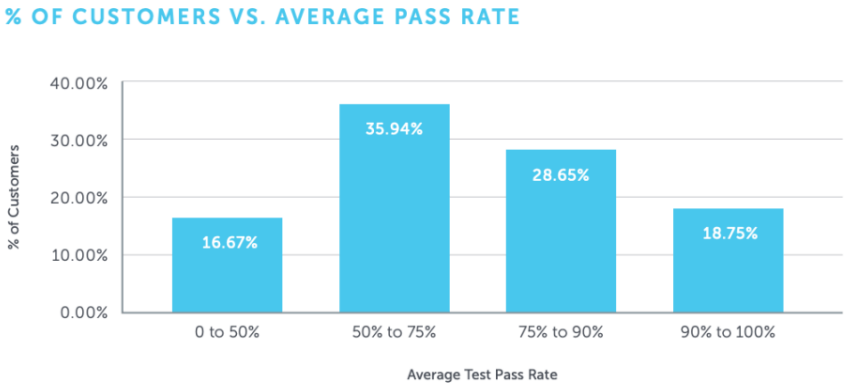
In order to accomplish a requirement of fast builds. You need small, fast tests, running in parallel. I'm sorry, there is no other way.

CHAPTER 3
TEST AUTOMATION ANTI-PATTERNS



Our industry pretty much sucks as a whole. Majority of us have no clue how to do test automation well.

For example, look at this analysis of 2 Billion Tests from all over the globe:



< 20 % of the world can execute their automation with > 90% pass rate https://saucelabs.com/news/sauce-labs-announces-results-of-inaugural-continuous-testing-benchmark-report-

How disappointing is that?

Some of my unit tests have been running without a failure for years. While our GUI automation can't even pass more than 9 times out of 10 runs.

This means that if you run your test every workday, in a 2-week span, your test will fail at least one time.

It gets better:

Now that we know the problem, we can work towards a solution! The question is why are we so bad at GUI automation?

It's because of these anti-patterns below…

Anti-Pattern: UI tests should not expose interactions with web elements

The benefit of using Page Objects is that they abstract implementation logic from the tests. The tests can be focused on the scenarios and not implementation. The idea is that the scenario doesn't change, but the implementation does.

For example, this method is performing a bunch of operations for some actions. At any point, we may need to change our steps. Maybe a new field got added and now we need to check a checkbox. Or, maybe one of the fields gets removed.

Even more common, you want to add logging. In that case, every test will need to be accommodated for this new flow (could be 1000s of tests).

```
UsernameField.Clear();
UsernameField.SendKeys(username);
PasswordField.Clear();
PasswordField.SendKeys(password);
LoginButton.Click();
SauceJsExecutor.LogMessage($"{MethodBase.GetCurrentMethod().Name} success");
```

It gets better:

The right way to solve this problem is to encapsulate all of the steps into a method called Login().

Now, it doesn't matter if we have to add logging, add an extra field, remove a field and so on. There will be a single place to update the Login steps, in the Login() method. Take a look below.

This test will only need to change for a single reason…

If requirements change, and there's no way around that:

```
[Test]
public void ShouldNotBeAbleToLoginWithLockedOutUser()
{
    _loginPage.Open();
    //Although I would likely never test a login through the UI. This is just a small example
    var productsPage = _loginPage.Login("locked_out_user", "secret_sauce");
    productsPage.IsLoaded.Should().BeFalse("we used a locked out user who should not be able to login.");
}
```

Anti-Pattern: Assuming that more UI automation is better

What do you think are some guaranteed ways to lose trust in automation and kill an automation program?

If you read the title and guessed the answer, great job

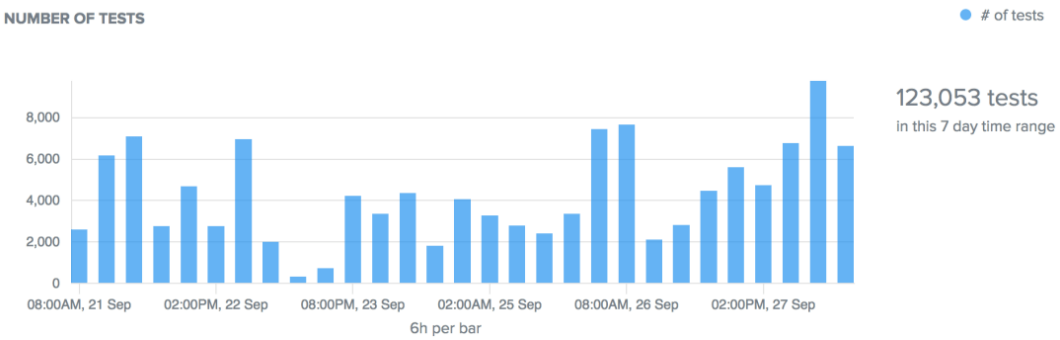
There are very few automation anti-patterns that will kill an automation program faster than using UI automated testing to try and test everything.

> "There are very few automation anti-patterns that will kill an automation program faster than using UI automation to try and test everything."

More automation is not necessarily better. In fact, I would argue that for an organization that is starting out, less stable automation is magnitudes of times better than more automation.

Here's some cool info:

I'm super lucky in that I get to consult and work with clients all over the world. So I've seen all sorts of organizations.

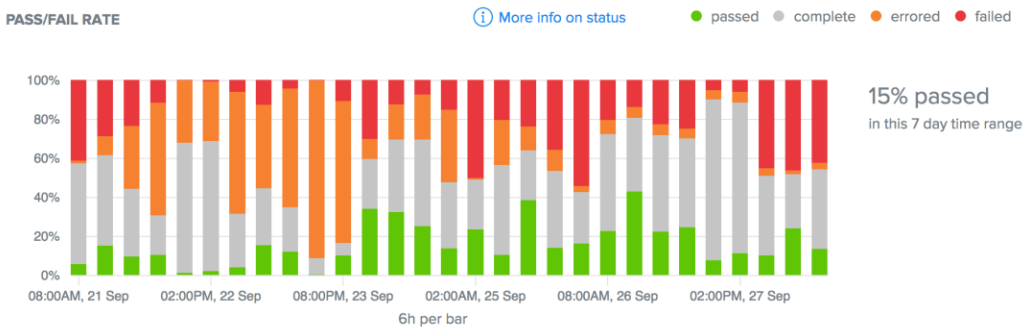


Organization that ran 123K UI tests in 7 days

This organization has executed 123K automated UI tests in 7 days!

Here's the kicker:

Take a look at this graph and how only 15% of the tests passed.



Very low passing rate

Can this organization really say that out of 100% of the features that are really being tested here, that 85% of those features contain bugs?

In that case this would mean that approximately ~104,000 bugs were logged in the 7 day period. That seems, highly unlikely, if not impossible…

So then, what are all of these failures and errors?

They're called false positives. Failing tests that are not a result of actual faults in the software being tested.

WHO IS SORTING THROUGH ALL OF THESE FAILURES?

Is there really someone on the team that is sitting and sorting through all of these failures?

~104,000 non-passing tests…

So what is the reason that these tests failed?

A. Because there is one bug in the application that caused all of these failures?

B. Because there are two or more bugs causing all of these problems?

C. Because there are ~zero bugs found and the majority of the failures are a result of worthless automation efforts?

I'd bet $104,000 that it's this option :)-

It gets worse:

How many automation engineers do you need to sort through 104,000 non-passing tests in one week?

When I ran a team of four automation engineers, we could barely keep up with a few non-passing automated tests per week.

So let's be honest… nobody is analyzing these non-passing automated tests…

So then what value are these test cases serving the entire organization? What decision do they help the business to make about the quality of the software?

If there was an 85% failure rate in your manual testing, do you move your software to production? Of course not…

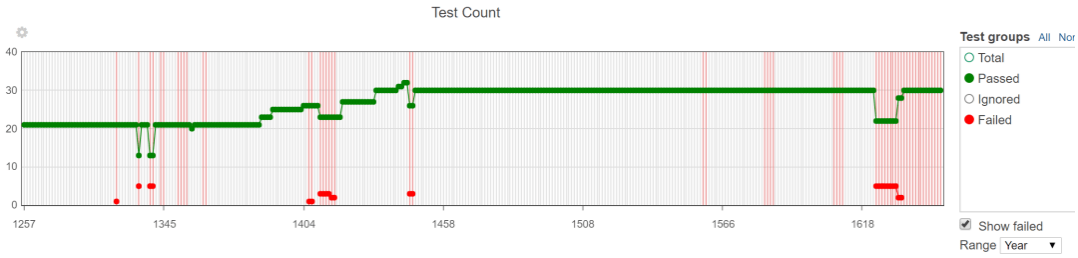
So why is it acceptable for so many automated tests to run, not pass, and continue to run?

It's because this automation is just noise now… Noise that nobody listens to… Not even the developers of the automation.

Automation Failed!

But, there's hope…

There are organizations that do automation correctly as well. Here's an example of one…



Automated tests executed over a year

Why is this automation suite more successful?

First, notice that it was executed over a year. And over a year there were not that many failures…

Yes, this doesn't necessarily imply that the automation is successful.

However…

Which automation would you trust more?

A. One that is passing for months at a time and gets a failure once every couple months?

B. Or the automation that has only 15% passing tests of which 104,000 of are not passing?

Here's where it gets interesting:

If you think about a single feature – Facebook login or Amazon search for example.

How often does that feature break based on your experience? Very rarely, if ever (based on my experience at least)

So if you have an automated test case for one of these features, which of the graphs above look more like how the development of the feature actually behaves?

That's your answer…

Your automated UI tests should behave almost identical to how actual development of a feature happens.

Meaning, passing majority of the time, like minimum 99.5% of the time and failing once in a blue moon, due to a real regression.

It gets better:

So what can you do to make your automation valuable?

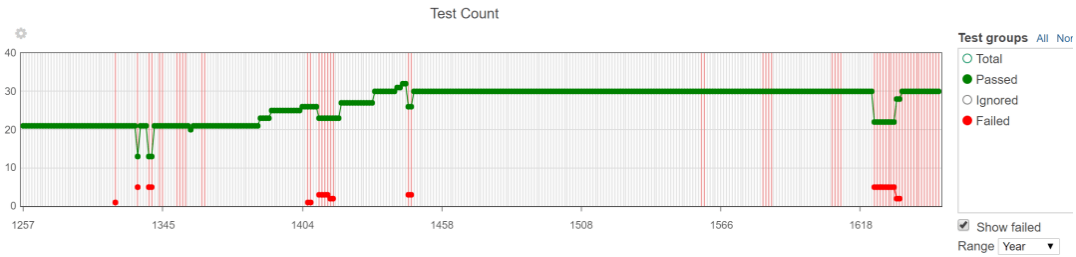
It's actually really simple…

> "If your automation is not providing a correct result more than 99.5% of the time, then stop automating and fix your reliability! You're only allowed a max of 5 false positives out of 1000 test executions! That's called quality automation.

What is 1000 executions? It means if you run your test once every single day, that's about 5 false positives every 3 years!"

Impossible, right?

Not at all. I actually ran the team that had these execution results below…



Automated tests executed over a year

Sadly, I no longer have the exact passing percentage of these metrics. But if you do a little estimation, you'll be able to see that the pass rate of this graph is extremely high.

Furthermore, I can say that every failure on this graph was a bug that was introduced into the system. Not a false positive which is so common in UI automation.

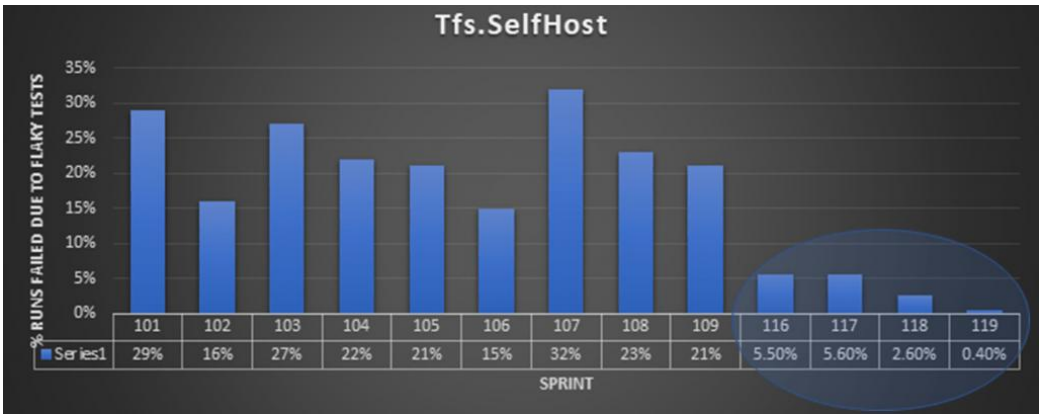
By the way, I'm not saying this to impress you…

Rather, to impress upon you the idea that 99.5% reliability from UI automation is possible and I've seen it.

Give it a shot and let me know your outcome

It gets better:

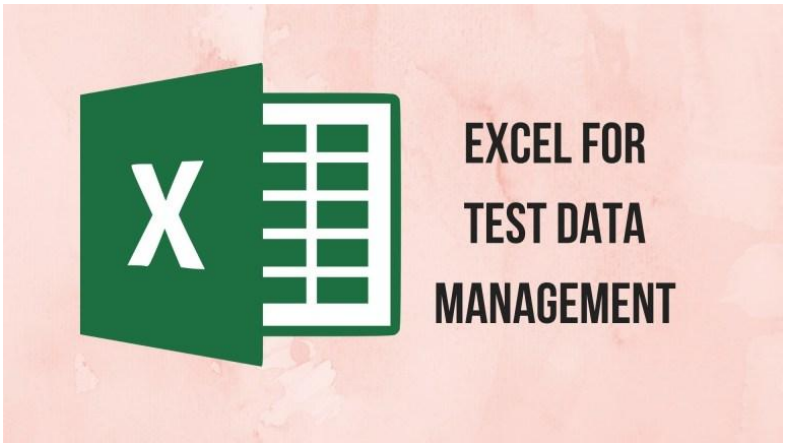
I recently came across an excellent post by a Microsoft Engineer talking about how they spent two years moving tests from UI automation to the right level of automation and the drastic improvement in automation stability. Here's the chart:



Check out after Sprint 116 when they introduced their new system!

Just another success story of a company that does automation at the right system level.

Anti-Pattern: Using complicated data store such as Excel



One of the most common questions from my students and clients is how to use Excel for test data management in test automation.

> "Don’t use Excel to manage your automation test data."

I understand the rationale behind using Excel for your test data. I’ve been doing test automation for a really long time and I know about Keyword Driven Frameworks and trying to let manual testers create automated tests in Excel. My friends…

It just doesn’t work… I wasn’t able to make it work myself and I’ve never seen anyone else make it work successfully.

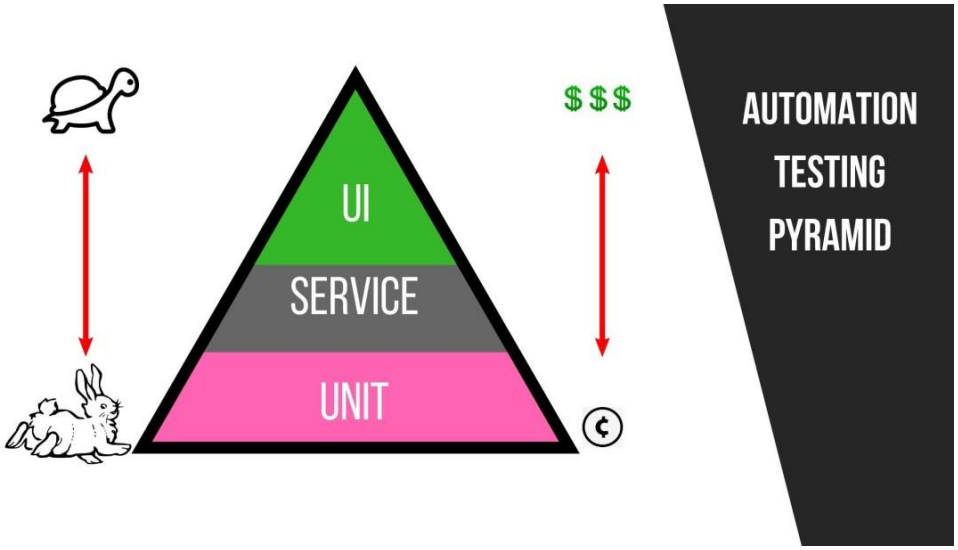
Why using Excel is an anti-pattern?

1. The logic to read and manage Excel adds extra overhead to your test automation that isn’t necessary. You will need to write 100s of lines of code just to manage an Excel object and read data based on column headers and row locations. It’s not easy and is prone to error. I’ve done it many years ago. All of this will eat into your automation time and provide no value to the business that employs you.
2. You will be required to manage an external solution component for your test automation. This means that you can never simply pull the code and have everything work. You will need to have a license for Excel. You will need to download and install it. And you will need to do this for all of your automation environments. Usually local, Dev, Test, and Prod. This means that you need to manage this Excel instance in all of these environments. This is simply another waste of your time.

What are your solutions?

1. The best solution is if you have an API that you can use to read test data. This is a robust and lightweight solution
2. If you don’t have an API, you can talk directly to the Database. This takes much less code and it’s much easier to manage than working with and external Excel object.
3. If you must use some data source, use a .csv or .json file. CSZ and JSON files are extremely lightweight, easy to read, and can be directly inserted into your automation code. This means that you will be able to simply download the code and have everything work without needing to perform other installations.

Anti-Pattern: Trying to use UI automation to replace manual testing



> Automated testing CANNOT replace manual testing

I have not read or seen any automation luminary who claims that automation can replace manual testing. Not right now at least… Our tools have a long way to go.

However, I know and have worked with managers and teams whose goal with test automation is exactly what cannot be done.

And so these individuals pursue a goal that is impossible… Obviously leading to failure.

Side note:

Use of automation can drastically enhance the testing process. If used correctly, automation can reduce, not replace the manual testing resources required.

WHY CAN'T AUTOMATION REPLACE MANUAL TESTING?

First, it's impossible to get 100% automated code coverage. It's actually impossible to get 100% code coverage in general…

That's why we still have bugs on all the apps in the world, right?

Anyways, if you can't automate 100% of the application, that means that you will need some sort of manual testing to validate the uncovered portions.

Second, UI automation is too flaky, too hard to write, and too hard to maintain to get over 25% code coverage…

This is based on experience and is an approximation… I don't have any hard data on this.

However, you actually don't want higher coverage than 25%. I guess it's possible that with a well designed, modular system, you might be able to get higher UI automation coverage.

But this is an exception, not the rule.

Here's the kicker:

Using a combination of unit, integration, and UI automation, you might get close to 90% test coverage…

But that's really hard. And this is just a side note.

Finally, there are some manual activities that cannot be automated technologically…

That's for now and for at least a few years in my opinion.

Some examples include UX Testing and Exploratory Testing.

So again, if you are trying to use automation to replace manual testing, it will be a futile effort.

WHAT IS THE SOLUTION?

Use the automation testing pyramid and don't try to replace manual testing with UI automation.

Use UI automation and any automation to enhance the testing process.

- ✓ A combination of manual testing and automated testing will create the best end user experience.

Anti-Pattern: Mixing functional automation with performance testing

Description coming soon… In the meantime, do your research about whether this makes sense. Remember, that at the end of the day, it is always faster to run ten, one minute tests in parallel than to run a single five minute test.

It's one minute to suite feedback time versus five minutes. Even if each test takes longer because of the setup and teardown, parallelization is still the most powerful way to scale your automation. Trying to scale your automation by combining tests together is not the right approach.

Anti-Pattern: Keyword Driven Testing

Keyword Driven Testing (KDT) is a remnant from the early to mid 2000s when QTP aka UFT was popular.

At one point, we believed that somehow we can write test automation code in such a way that manual testers will be able to string together a bunch of keywords aka functions to make tests.

We used Excel sheets to design test cases by stringing together a bunch of functions (using Excel sheets is also an anti-pattern)

Here's an example:

WHAT ARE THE PROBLEMS WITH KDT?

1. Almost nobody in the world is doing KDT any more

First, let's define MY success criteria:

- 100% reliability from test automation
- Automation executes on every pull request
- Full automation suite runs in less than 10 min
- Automation is used as a quality gate by the entire organization

I'm fortunate enough to work with dozens of clients and hundreds of automation engineers, every single year.

In my entire career, I have never seen a successful implementation of Keyword Driven Testing. That doesn't mean that there isn't one. But…

Why are they so rare? And do you really want to take the chance to beat these odds?

Even worse, no automation luminary (Simon Stewart, Titus Fortner, Angie Jones, Alan Richardson, Paul Merrill…) even mentions KDT when they talk UI test automation…

There must be a reason why, don't you think?

Sure, I've talked to people online that claim that KDT is working for them. Yet it's also funny that these people don't want to share anything publicly…

2. Unnecessary code must be written to read some external source like Excel

Someone has to write the logic to manage the reading and writing to an Excel spreadsheet. Or some other data source. But the Excel spreadsheet is most common with this approach because it's most user friendly for manual testers.

That logic is pretty absurd and takes A LOT of time to get right. Here's what mine looked like:

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Net;
using System.Runtime.InteropServices;
using log4net;
using log4net.Config;
using Microsoft.Office.Core;
using Microsoft.Office.Interop.Excel;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using OpenQA.Selenium;
using Environment = System.Environment;

namespace AutomatedTesting.DriverScript
{
    public class Program
    {
        #region Fields and Properties
        private static string _emailTo, _timeStamp;


        private static ILog _log = LogManager.GetLogger(typeof(Program).FullName);
        /// A suffix to append to the class name / namespace of a class to form a default assembly qualified name.
        private const string DEFAULT_ASSEMBLY_QUALIFIED_SUFFIX = " Version=1.0.0.0, Culture=neutral, PublicKeyToken=null";
        private static Dictionary<string, bool> _testResults = new Dictionary<string, bool>();


        #endregion


        public static int Main(string[] args)
        {
```

```
        var exitCode = ExitCode.Success;
        GlobalContext.Properties["hostname"] = Environment.MachineName;
        XmlConfigurator.Configure(new    FileInfo(string.Format(@"{0}\log4net.config",
AppDomain.CurrentDomain.BaseDirectory)));

        BaseTester bt = null;

        Application xlApp = null;
        Workbook wb = null;
        Workbooks workBooks = null;
        Worksheet testCasesSheet = null;
        Worksheet browsersControlSheet = null;
        CommandLineParameters parameters = null;
        var exHandler = new ExceptionHandler();
        var allTestData = new AllTestData();
        var testMetadata = new TestMetadata();
        var externalTestData = new TestDataFromExternalSource();

        if (args.Length == 0)
        {
          _log.Fatal("args is null");
          exitCode = ExitCode.EmptyArguments;
          //TODO send email
        }
        else
        {
          try
          {
            _log.Info("STARTED FRAMEWORK....");
            _timeStamp = DateTime.Now.ToString("yyyy.MM.dd.HH.mm.ss.fff");

            allTestData.Metadata = testMetadata;
            allTestData.External = externalTestData;

            parameters = FillInCommandLineParameters(args);
```

```
            Notifier.SendHeartbeat(parameters.Environment);
            Notifier.ResetExceptionMessage();


            //This is the path of the entire suite of tests for some application that is specified from command line
            var originalTestSuiteFilePath =
              Path.GetFullPath("TestControls/" + string.Format(ExcelManager.TestSuiteFileName, parameters.App));
            var uniqueTestSuiteFilePath = ExcelManager.CreateUniqueWorkbookInstance(originalTestSuiteFilePath,
              parameters, _timeStamp);


            //2. Create the DB objects to work with the test cases
            xlApp = new Application {FileValidation = MsoFileValidationMode.msoFileValidationSkip};
            workBooks = xlApp.Workbooks;
            wb = workBooks.Open(uniqueTestSuiteFilePath, ReadOnly: false, Editable: true);


            testCasesSheet = (Worksheet) wb.Worksheets["TestCases"];
            browsersControlSheet = (Worksheet) wb.Worksheets["CrossPlatformTestingConfig"];


            var testCasesUsedRange = testCasesSheet.UsedRange;
            var testCasesHeadersDictionary = ExcelManager.GetColumnNamesAndIndices(testCasesUsedRange,
              testCasesSheet);


            var crossPlatformUsedRange = browsersControlSheet.UsedRange;
            var crossPlatformDictionary = ExcelManager.GetColumnNamesAndIndices(crossPlatformUsedRange,
              browsersControlSheet);


            //The buildName represents a single unique time where all of the tests are ran for some application.
            //Caution, BrowserStack removes any special characters and replaces them with a space, therefore, don't put special characters in the name
```

```
            //or the buildId will not be found in BrowserSTack
            var desiredTags = string.Join(" ", parameters.Tags);
            allTestData.Metadata.Build = string.Format("app:{0}. env:{1}. tags:{2}. user:{3}. time:{4}",
              parameters.App,
              parameters.Environment.ToUpper(), desiredTags, Environment.UserName, _timeStamp);
            ExcelManager.NumberOfRowsForCrossPlatformTesting = crossPlatformUsedRange.Rows.Count;


            var dataRetreiver = new ExcelDataRetreiver(browsersControlSheet, crossPlatformDictionary);
            //For every single row in the CrossPlatformTestingConfig sheet, we will perform the actions
            for (var i = ExcelManager.StartingRow; i < ExcelManager.NumberOfRowsForCrossPlatformTesting; i++)
            {

//ConfigureBrowsersAndPlatformsBasedOnExecutionMode(browsersControlSheet, crossPlatformDictionary, i, _smokeTestMode);
              if (parameters.SmokeTestMode || parameters.LocalTestingMode)
                //if we're running smoke tests or local tests, we only need to run through 1 iteration of the tests
              {
                //set rowCount to 1 above the starting row so that only one iteration occurs
                ExcelManager.NumberOfRowsForCrossPlatformTesting = 4;
                //get only the first row of the spreadsheet which will signify the default os/browser combination
                allTestData.Metadata.Browser = dataRetreiver.RetreiveValue(i, "browser");
                allTestData.Metadata.Version = dataRetreiver.RetreiveValue(i, "version");
                allTestData.Metadata.Os = dataRetreiver.RetreiveValue(i, "os");
                allTestData.Metadata.OsVersion = dataRetreiver.RetreiveValue(i, "os_version");
                allTestData.Metadata.Resolution = dataRetreiver.RetreiveValue(i, "resolution");
              }
```

```
                else
                {
                    allTestData.Metadata.Browser = dataRetreiver.RetreiveValue(i,
"browser");
                    allTestData.Metadata.Version = dataRetreiver.RetreiveValue(i,
"version");
                    allTestData.Metadata.Os = dataRetreiver.RetreiveValue(i, "os");
                    allTestData.Metadata.OsVersion = dataRetreiver.RetreiveValue(i,
"os_version");
                    allTestData.Metadata.Resolution = dataRetreiver.RetreiveValue(i,
"resolution");
                }

                //None of those should be null
                if (string.IsNullOrEmpty(allTestData.Metadata.Browser) ||
                    string.IsNullOrEmpty(allTestData.Metadata.Version) ||
                    string.IsNullOrEmpty(allTestData.Metadata.Os) ||
                    string.IsNullOrEmpty(allTestData.Metadata.OsVersion)) continue;

                _log.InfoFormat(
                    "Iterating through every single row of the spreadsheet to find the test
cases that have an Execute flag.");
                for (var j = 3; j <= testCasesUsedRange.Rows.Count; j++)
                {
                    var executeColumn = "execute" + parameters.Environment;
                    var cellValue = ExcelManager.GetCellValue(testCasesSheet,
testCasesHeadersDictionary, j,
                        executeColumn);
                    //sometimes there may be formatted cells in the spreadsheet that do
not contain any data
                    //we may iterate through them, but should not do any further action if
there are no values
                    if (string.IsNullOrEmpty(cellValue)) continue;

                    allTestData.External.TestCaseName =
ExcelManager.GetCellValue(testCasesSheet, testCasesHeadersDictionary, j,
                        "classname");
```

```
                var strOfTagsFromExternalSource = ExcelManager.GetCellValue(testCasesSheet, testCasesHeadersDictionary, j,
                  "tags");
                string[] tagsFromSpreadsheet;
                if (strOfTagsFromExternalSource != null)
                  tagsFromSpreadsheet = strOfTagsFromExternalSource.ToLower().Split(',');
                else
                {
                  throw new Exception(
                    string.Format("There are no tags set up for this test:{0}. Please add tags.",
                      allTestData.External.TestCaseName));
                }
                //check to see if the tags in the test cases sheet were called for in the
                //execute file. If they match up, then those test cases will be ran
                var tagsMatch = tagsFromSpreadsheet.Any(x => parameters.Tags.Contains(x)) ||

parameters.Tags.Contains(allTestData.External.TestCaseName.ToLower());
                _log.Debug($"row:{j}. cellValue:{cellValue}. tags retrevied from external source:{strOfTagsFromExternalSource}. tagsMatch:{tagsMatch}");


                //if the execute flag is equal 'y' then that means that we are going to execute this test
                if (cellValue.ToLower().Trim() != "y" || !tagsMatch)
                  continue;
                allTestData.External.NameSpace = ExcelManager.GetCellValue(testCasesSheet, testCasesHeadersDictionary, j,
                  "namespace");



                //All of the optional parameters that don't necessarily need values
                var testLevelType = ExcelManager.GetCellValue(testCasesSheet, testCasesHeadersDictionary, j,
                  "testlevel");
```

```
                //if this is an 'api' leve test case, then we dont need it iterating through the regression browsers
                //because that is not relevant. Therefore it will only do 1 loop, on 1 browser, through all the test cases
                if (!string.IsNullOrEmpty(testLevelType) && testLevelType.ToLower() == "api" || tagsFromSpreadsheet.Contains("smoke"))
                    ExcelManager.NumberOfRowsForCrossPlatformTesting = 4;


                allTestData.External.TestCaseDescription =
                    testCasesSheet.Cells[j, testCasesHeadersDictionary["description"]].value !=
                    null
                        ? Convert.ToString(
                            testCasesSheet.Cells[j, testCasesHeadersDictionary["description"]].value)
                        : "Description was not specified in the test suite control";


                bt = CreateBaseTesterInstance(allTestData);
                bt.LocalTestingMode = parameters.LocalTestingMode;


                var testPass = false;
                //Execute the test case
                try
                {
                    //Initialize the Driver
                    allTestData.Metadata.Project = "WI_" + parameters.App.ToLower() + "_" + parameters.Environment;
                    _log.InfoFormat("--Running Automation Check.");
                    _log.InfoFormat("------App:'{0}'.", parameters.App);
                    _log.InfoFormat("------Test Case Name:'{0}'.", allTestData.External.TestCaseName);
                    //Set up envs
                    //TODO, if Test set up fails, then we should not update the previous session with our results
                    bt.TestSetUp(parameters, allTestData);
                    bt.RunTest(parameters.Environment);
                    testPass = true;
```

```
                }
                catch (WebDriverException e)
                {
                    //"Failed to start up socket within 45000 ms. Attempted to connect to the following addresses:"
                    Notifier.MyException = e;
                    Notifier.SetEmailToGroup(parameters);
                }
                catch (WebException e)
                {
                    Notifier.MyException = exHandler.HandleWebException(e);
                }
                catch (InvalidOperationException e)
                {
                    //The browser/OS was not compatible for BrowserStack and Driver was never initialized.
                    //This failure is not critical because the applications are still working and only the QA should be notified
                    //Or it could be the "Not able to reach BrowserStack..." message.
                    Notifier.MyException = e;
                    Notifier.SetEmailToGroup(parameters);
                }
                catch (AssertFailedException e)
                {
                    Notifier.MyException = e;
                    Notifier.SetEmailToGroup(parameters);
                }
                catch (Exception e)
                {
                    Notifier.MyException = e;
                }
                finally
                {
                    _testResults.Add(string.Join(".", Guid.NewGuid(), parameters.Environment, parameters.App, allTestData.External.TestCaseName), testPass);
```

```
                    bt.CleanUp(Notifier.MyException);
                    exitCode = Notifier.LogUpdateAndEmailFailures(allTestData.External.TestCaseName, parameters, allTestData, bt);
                    Notifier.ResetExceptionMessage();
                  }
                } //End of the for loop that treverses the entire DB to count all the test cases
              } //End of the For loop for the CrossPlatformTestingConfig sheet
            } //End of the try block where all the main logic lives
            catch (ArgumentNullException e)
            {
              _log.FatalFormat(
                "Error creating a BaseTester class as a result of missing classes. Not found:'{0}'",
                allTestData.External.NameSpace + "." + allTestData.External.TestCaseName);
              exitCode = ExitCode.BrowserStackSessionRetreivalFailure;
              //TODO move into SetEmailToGroup
              _emailTo = ConfigurationManager.AppSettings["qaEmailGroup"];
              Notifier.SendFailureNotification("Error creating a BaseTester class as a result of missing classes.",
                "Error creating a BaseTester class as a result of missing classes.", e, parameters, allTestData);
            }
            catch (COMException e)
            {
              exitCode = exHandler.HandleCOMException(e);
              Notifier.SendFailureNotification("COMException", parameters.App, e, parameters, allTestData);
            }
            catch (Exception e)
            {
              Notifier.SendFailureNotification("Uknown Exception", parameters.App, e, parameters, allTestData);
              exitCode = exHandler.HandleGenericException(e);
            }
```

```
            finally
            {
               try
               {
                  ExcelManager.ReleaseExcelObjects(ref wb, ref xlApp, ref testCasesSheet, ref browsersControlSheet, ref workBooks);
               }
               catch (COMException e)
               {
                  exitCode = exHandler.HandleCOMException(e);
               }

               KillAllProcesses("EXCEL");
            }

         }  //End of the else loop that handles all of the logic for if the args != empty
         Console.ForegroundColor = ConsoleColor.Cyan;
         Console.WriteLine(Environment.NewLine + Environment.NewLine + "---------- Automated Check Results ----");
         Console.WriteLine("Application:{0}", parameters.App);
         Console.Write("Environment:{0}", parameters.Environment);
         Console.WriteLine(Environment.NewLine);

         if (_testResults.Count != 0)
         {
            var testCount = 1;
            foreach (var test in _testResults)
            {
               Console.ForegroundColor = ConsoleColor.Green;
               if (test.Value == false)
               {
                  Console.ForegroundColor = ConsoleColor.Red;
                  exitCode = ExitCode.TestFailed;
               }
```

```
                Console.WriteLine("{2}. Automated Check:{0} - Result:{1}", test.Key, test.Value, testCount);
                testCount++;
            }
            Console.ForegroundColor = ConsoleColor.Cyan;
            Console.WriteLine(
              "Docs for checking status of tests are here: http://sdlc.devcentral.equifax.com/confluence/display/WAL/How+to+execute+the+automated+functional+tests+through+Posh");
          }
          else
          {
            _log.WarnFormat(
              "The framework did not find any tests to run. Check to see if any tests actually exist for your Tags. App:'{0}'. Env:'{1}' that you passed in through command line.",
                parameters.App, parameters.Environment);
            exitCode = ExitCode.NoTestsWereExecuted;
          }
          Notifier.SendHeartbeat(parameters.Environment);
          _log.InfoFormat("{1}Exiting Framework. Result of batch run:{0}", exitCode, Environment.NewLine);
          return (int)exitCode;

        }



        /// <summary>
        /// Will kill the process that is specified
        /// </summary>
        /// <param name="processName">the name of the process such as Excel</param>
        private static void KillAllProcesses(string processName)
        {
          foreach (var proc in Process.GetProcessesByName(processName))
          {
            proc.Kill();
```

```
            }
        }


        private static BaseTester CreateBaseTesterInstance(AllTestData allTestData)
        {
            // var assemblyQualifiedName = nameSpace + "." + className + "," + nameSpace
+ "," + DEFAULT_ASSEMBLY_QUALIFIED_SUFFIX;
            var assemblyQualifiedName =

$"{allTestData.External.NameSpace}.{allTestData.External.TestCaseName},{allTestData.
External.NameSpace},{DEFAULT_ASSEMBLY_QUALIFIED_SUFFIX}";


            var typeDynamic = Type.GetType(assemblyQualifiedName);
            BaseTester bt;
            if (typeDynamic != null)
            {
                bt = (BaseTester)Activator.CreateInstance(typeDynamic);
                bt.TestName = allTestData.External.TestCaseName;
            }
            else
            {
                throw new ArgumentOutOfRangeException(
                    "When trying to create the dynamic class, we did not find a reference of this
project in the Main method." +
                    "Please Add a reference of your project from which you are running a test to
the Main method of the Program.cs class." +
                    "Located in this project IXI.AutomatedTesting.DriverScript");
            }


            return bt;
        }


        /// <summary>
        /// A method to decide who should receive the email based on different
parameters.
        /// For example, if it's Dev or Test, then BJ should not be receiving any of these
failures.
```

```
        /// If there are certain Exceptions that are not related to actual test failures but due to BrowserStack, then BJ should not receive those emails.
        /// </summary>
        /// <param name="parameters"></param>
        /// <param name="type">Represents the exception that was thrown because sometimes we want to send an email to a special group based on some exception.</param>


        private static CommandLineParameters FillInCommandLineParameters(string[] args)
        {
            var parameters = new CommandLineParameters();
            parameters.AcceptSsl = false;
            parameters.LocalTestingMode = false;
            for (var i = 0; i < args.Length; i++)
            {
                switch (args[i].ToLower())
                {
                    case "-app":
                        parameters.App = args[i + 1];
                        break;
                    case "-env":
                        parameters.Environment = args[i + 1].ToLower();
                        break;
                    //-debug switch will override all
                    case "-debug":
                        parameters.DebugMode = true;
                        Notifier.SetEmailToGroup(parameters);
                        break;
                    case "-tags":
                        var tagsLine = args[i + 1];
                        parameters.Tags = Array.ConvertAll(tagsLine.ToLower().Split(','), p => p.Trim());
                        break;
                    case "-localtesting":
                        parameters.LocalTestingMode = true;
```

```
                    break;
                case "-acceptssl":
                    parameters.AcceptSsl = true;
                    break;
            }
        }

        //run through all of the tags, if one of them says Smoke test, then that means that IXI-Bugs needs to be notified
        //TODO update query to the same one as the one in Excel using the .contains
        foreach (var tag in parameters.Tags)
        {
            if (tag.ToLower() == "smoke")
            {
                //set the smoke test mode to true so that we only run through one iteration of a default browser
                parameters.SmokeTestMode = true;
                Notifier.SetEmailToGroup(parameters);
            }
        }

        //TODO add error handling to make sure that things like app, env and tags are passed in to let the user know to pass them in
        _log.DebugFormat("Tags list- app:{0} -env:{1} -debug:{2} -tags:{3} -acceptSsl:{4}", parameters.App, parameters.Environment, parameters.DebugMode, string.Join(",", parameters.Tags, parameters.AcceptSsl));
        return parameters;
    }
  }
}
```

Criticize it all you like, I know it's bad. It was many years ago when I was learning C#.

Regardless, if I did this today with better programming patterns, it would still be a waste of time.

It gets better:

Here is how I data drive my tests today:

```
namespace Web.Tests.BestPractices
{
    [TestFixture]
    [TestFixtureSource(typeof(CrossBrowserData), "LastTwoOnLinuxFirefoxChrome")]
    [Parallelizable]
    public class LogoutFeature : BaseTest
    {
        public LogoutFeature(string browser, string version, string os) :
            base(browser, version, os)
        {
        }
        [Test]
        public void ShouldBeAbleToLogOut()
        {
            var loginPage = new SauceDemoLoginPage(Driver);
            loginPage.Open();
            var productsPage = loginPage.Login("standard_user", "secret_sauce");
            productsPage.Logout();
            loginPage.IsLoaded.Should().BeTrue("we successfully logged out, so the login page should be visible");
        }
    }
}
```

This even shows how the test case looks. But really, all of the logic for how to read test data for all of my tests is in this single line of code:

```
8       [TestFixture]
9       [TestFixtureSource(typeof(CrossBrowserData), "LastTwoOnLinuxFirefoxChrome")]
10      [Parallelizable]
11      public class LogoutFeature : BaseTest
```

One line of code to read test data

Concise, isn't it? From 100s of lines of code, to one (honestly, it's not a 100% apples to apples comparison, but it's close enough).

Thanks so much NUnit testing framework and all of the Developers that spent years maintaining this code ☺

Without you, I might still be stuck writing my own code to do the same thing ☺

PS. Don't get me started on all the logic required to parse and manage strings from an Excel sheet.

3. More complexity by introducing external teams

If you've done test automation for some time, I think that you will agree with me that the success criteria that I defined above is not easy to meet?

Even doing test automation without KDT, with simple Page Objects, most of us struggle to even achieve one of the KPIs.

It gets worse:

Imagine trying to do automation where you have to teach manual testers how to use your Excel spreadsheet. Not that there is anything wrong with manual testers.

However, the problem is that now we need to add more communication, which means more potential for miscommunication aka problems.

It's just another unnecessary variable.

Codeless test automation tools have been failing for decades, why are we trying to recreate them?

4. KDT forces you to create your own test runners

Because we want to read a spreadsheet and turn those keywords into functions using reflection, we can't use the beautiful testing frameworks that have been developed for us by really smart people from JUnit (Java), NUnit (C#), Mocha (JS) and so on…

Seems like a waste of time to me, how about you?

5. KDT forces your tests to be written with interaction commands in mind

The problem here is that Keyword Driven Testing encourages your tests into an improper mindset of ClickButton(), SendKeys(), ClickElement()…

Your automated tests should be written through user actions versus element interactions:

Login() versus SendKeys(), SendKeys, ClickButton()

Anti-Pattern: Giant BDD Tests

Description is coming soon…

However, you do not want to have large BDD tests with many “When” and “Then” because it means that you are testing too much. It means that your tests will not be atomic. See Atomic Tests Pattern at the top.

Anti-Pattern: Imperative BDD Tests

> "The biggest mistake BDD beginners make is writing Gherkin without a behavior-driven mindset. They often write feature files as if they are writing “traditional” procedure-driven functional tests: step-by-step instructions with actions and expected results. These procedure-driven tests are often imperative and trace a path through the system that covers multiple behaviors. As a result, they may be unnecessarily long, which can delay failure investigation, increase maintenance costs, and create confusion."
>
> Automation Panda, https://automationpanda.com/2017/01/30/bdd-101-writing-good-gherkin/

Anti-Pattern: Large Classes

> "Every class in your test automation project should be no longer than 200 lines of code."
>
> -Nikolay Advolodkin

Ultimately, this is a software programming problem. But developing test automation is a programming exercise, so the same concepts apply.

Having small classes (less than 200 lines long) provides the following benefits:

Classes are more likely to follow the Single Responsibility Principle

It’s easier to understand what a class is doing. Think GoogleHomePage class versus WebApplicationManager class

As a result of the first two points small classes are easier to maintain

What if I don’t want to have too many classes?

I believe this is a fallacy. I have never encountered anyone who complained of having too many classes or files. Even if it’s possible, seems likely an exception, rather than the rule.

CHAPTER 4
RED FLAGS

This section is a collection of automation techniques that I have seen with my clients that cause them a lot of problems. I can't quiet classify them as "anti-patterns" because they are not widely accepted as such in the automation community, by the automation experts. However, I do believe that they are on the brink of being bad practices that you should strongly reconsider.

Using BDD tools for UI automation

It all really starts with a simple question…

What is Behavior Driven Development?

> "Behaviour-Driven Development (BDD) is a set of practices that aim to reduce some common wasteful activities in software development:
>
> -Rework caused by misunderstood or vague requirements.
>
> -Technical debt caused by reluctance to refactor code.
>
> -Slow feedback cycles caused by silos and hand-overs
>
> BDD aims to narrow the communication gaps between team members, foster better understanding of the customer and promote continuous communication with real world examples.
>
> https://docs.cucumber.io/bdd/overview/"

Now that we understand that, let me ask…

Did you see the word tool or tools used a single time?

No, we didn't.

This implies that BDD is NOT a tool, it is a set of practices.

In fact, Aslak Hellesoy, the creator of Cucumber says,

> "If you think Cucumber is a testing tool, please read on, because you are wrong.
>
> There is a process to follow that involves many roles on the software team.
>
> This process is called BDD. It's what came out of that clique I mentioned. BDD is not a tool you can download
>
> The World's most misunderstood collaboration tool, Cucumber.io"

I'm fortunate in that I'm a Solutions Architect (SA) and I get to talk to dozens of new customers and hundreds of automation engineers every year.

Here's where it gets bad:

The common problem that I encounter and my fellow SAs…

is that almost nobody uses BDD as a set of practices.

No practices are implemented to remove all the waste and technical debt.

Instead, tools such as Cucumber, Serenity, SpecFlow are used to write automated tests and then we claim that we are "doing BDD".

It gets worse:

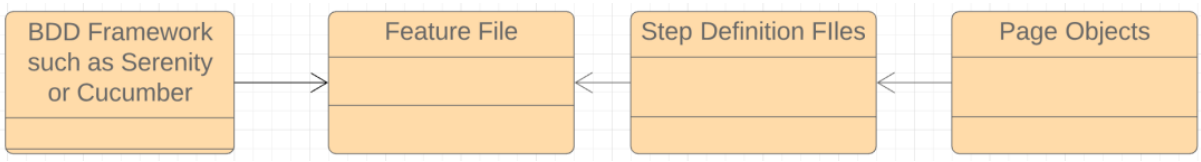
The problem is that using BDD tools adds an extra layer of complexity and dependency to test automation code.

If we aren't using the BDD process for the actual advantages then all we are really doing is adding extra complexity for no benefits

These are the problems that I see when a BDD tool is used for automation without actually following the BDD practices:

PROBLEM1: BDD TOOLS CREATE MORE DEPENDENCIES

Let's take a look at a diagram that shows all the dependencies that are added when using a BDD automation framework (I didn't include all the other dependencies such as test runners and so on as they're not related to BDD).

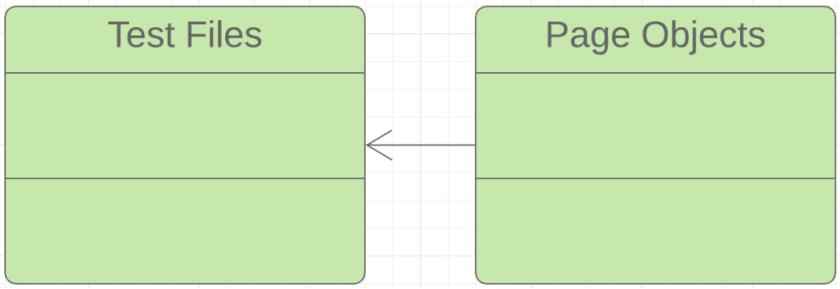


bdd

When you add a BDD tool to your automation suite, you have the BDD framework such as Cucumber used by Feature Files. Feature Files use Step Definitions and Step Definitions use Page Objects.

Hold on to that thought for a second…

What if you don't use a BDD automation tool?



Without a BDD tool

By not using a BDD tool, we can remove two extra dependencies.

Dependencies in software development are almost always bad. We want to limit the number of dependencies because each one is a chance for something to go wrong.

Most of the software development design patterns focus on dependency management…

Think Single Responsibility Principle or Open-Closed Principle.

> "In software development, we strive towards having our modules doing less while limiting the number of dependencies."
>
> -Nikolay Advolodkin

So why are we adding extra BDD tool dependencies to our automation if we aren't using the process?

BDD tools help to create more readable code:

Yes, this is true. I would agree that a test written in good Gherkin syntax is very readable.

However, is it that big of a difference when compared to a non-BDD test like this:

```
[Test]
public void ShouldNotBeAbleToLoginWithLockedOutUser()
{
    _loginPage.Open();
    //Although I would likely never test a login through the UI. This is just a small example
    var productsPage = _loginPage.Login("locked_out_user", "secret_sauce");
    productsPage.IsLoaded.Should().BeFalse("we used a locked out user who should not be able to login.");
}
```

I don't believe it's that drastic of a difference.

Is it really worth it to take the risk of extra dependency management for slightly more readable tests?

It gets worse:

The other problem that seems to happen with majority of the BDD tests is that they don't follow actual BDD best practices.

PROBLEM 2: BDD IS BEING DONE BY THE WRONG TEAM!

I recently read an interesting article that pointed out another problem of using BDD Tools for automation. As most organizations currently do BDD, it's the automation engineers that are simply converting manual tests to BDD tests, right?

What that ultimately means is that the completely wrong team is writing BDD.

The developers who are designing the code aren't writing the test specifications. Even though they are most familiar with the system.

The BAs aren't writing the specifications either because they view BDD as an automation tool. Even though the BAs are the ones that are the closes to the client.

It gets worse:

The BDD specs are being written by the isolated team that rarely communicates with the client and doesn't have an intimate knowledge of the system being tested.

How does that make any sense?

PROBLEM 3: PARALLEL NIGHTMARES

As far as I've seen with BDD automation frameworks such as Cucumber or SpecFlow, parallelization with them is a problem.

You can only parallelize at the Feature File level. Which means that you can only run as many tests in parallel as you have feature files. This is actually a major problem if you are trying to scale your test automation.

More to come soon…

PROBLEM 4: POOR GHERKIN IS CODED

More to come soon…

PROBLEM 5: UNNECESSARY OVERHEAD

Every technology brings overhead with it because you need to learn and answer some of the following questions:

- How does it work?
- What are the best practices?
- How do I use it?
- How do I install it?

…

Here's a real example of some of the rules that you will need to learn if you want to do BDD correctly (there are many more than this):

> "Write all steps in third-person point of view
>
> Write steps as a subject-predicate action phrase
>
> Given-When-Then steps must appear in order and cannot repeat
>
> …
>
> https://automationpanda.com/2017/01/30/bdd-101-writing-good-gherkin/"

The most important question that we need to ask ourselves is

Why are we trying to answer all of these questions? For what purpose?

For slightly better readability… Because our boss told us so…

Those don't seem like valid reasons to spend extra time mastering another tool.

ABOUT THE AUTHOR

> Nikolay Advolodkin
>
> Test automation leader and passionate instructor

Nikolay Advolodkin is a seasoned IT Professional, Test Automation Expert, and Quality Assurance Innovator whose dedication to innovation and progress has earned him the reputation as a strategist in the information technology space.

Throughout the span of his technical career, he has not only cultivated extensive experience, he has received extensive acclaim for his continual success. Most recently, he was a contributing author to Continuous Testing for DevOps Professionals. He was named one of 33 Test Automation Leaders to follow in 2017 by TechBeacon.com. And according to Udemy.com, he is one of the top Selenium WebDriver Instructor across the globe, educating 50,000+ students on the ins and outs of test automation from 120+ different countries.

Growing up, Nikolay discovered his passion for computers early on in his life. After successfully hacking a TI-83 calculator in the 8th grade, he knew IT would be a lifelong interest.

Currently, Nikolay proudly serves as the CEO and Test Automation Instructor at UltimateQA (www.ultimateqa.com). Furthermore, he is a frequent Contributor at SimpleProgrammer.com and TechBeacon.com. He was also a Speaker at multiple different conferences.

When he isn't teaching people how to be automated software testing masters or revolutionizing the test automation world as we know it, Nikolay Advolodkin enjoys exercising, self-development, and travel. He is also an avid entrepreneur with an unquenchable thirst for knowledge.

COPYRIGHT