Building Stakeholder Confidence Through an Automated Testing Solution

Rebecca Long

rebeccal@stcu.org

Abstract

How do you build stakeholder confidence in your software and also help minimize time spent on manual User Acceptance Testing (UAT) efforts? By getting stakeholders involved in the development of an automated testing suite, you build a foundation of understanding in both what is being tested and how the software is being tested. This in turn also allows the stakeholders to be more familiar with the implementation of the software. Involving stakeholders builds confidence in the software, the test coverage of features and gives them a clearer understanding of why less manual tests during UAT are required.

Building a system to meet this need requires good architecting to allow for stakeholder involvement, easy scalability, portability, and minimal maintenance. At Spokane Teachers Credit Union (STCU), we designed an automation test system called Browser Bot that uses SpecFlow to allow business owners to write the tests, Selenium for driving the browser, a custom library to dynamically pull test data based on test criteria, and TeamCity to run the tests on a Selenium Grid. The partnership between the software department and stakeholders to create this system has already begun to build the desired understanding and confidence in our systems from stakeholders. Using my team at STCU as an example, I will identify some of the common problems, solutions we used and the results we achieved. By following a similar approach, you can also get positive stakeholder participation while meeting the testing needs of your software team.

Biography

Rebecca Long is a quality assurance engineer at Spokane Teachers Credit Union in Spokane, Washington. She has passionately ensured the delivery of stable software with a delightful user experience through effective and efficient testing methodologies for the last decade. Her quality centric approach streamlines the software development process and ensures many problems are mitigated before they arise. Additionally, she also co-runs SpoQuality, a quality assurance user group based in Spokane.

1 Introduction

Companies are constantly pushing for software to be developed faster, tested in less time, and rushed out the door into production as quickly as possible. Management often believes that it's more valuable to get user feedback immediately with whatever can be released instead of solidifying an application before shipping it. The rise in popularity with the agile mentality seems to be lending to this misconception among business and project managers who may be jumping on this bandwagon before fully grasping the true intention behind the agile development lifecycle. As more managers fall in line with the falsehood of rushing software out the door, or as less managers push back on this idea, the harder it is for software teams to stand their ground to maintain good software practices. Yes, sometimes software does need to be rushed out the door. Yes, sometimes there is not enough time to fully regression test. Yes, there will always be bugs that escape to users. The key is balancing these inescapable realities with a best attempt at risk guided efforts toward avoiding pitfalls.

2 The Problems

At STCU, our software team ran into the following problems as we grew in both the size of our agile team and the amount of software we produced. Some of these problems were due to natural growing pains of a team and some of these problems were from the push to rush software out the door quickly.

2.1 Relying on Stakeholders as Bug Testers

It has been tried and proven by other companies – the best example being Netscape -- that having your end users as a main source for testing resources does not work out well (Spolsky 2000). Having your end users be your testers produces real world scenarios and real world problems, this is true. But the problem comes into play when your users get too early of access to your software and gain this perception that the software is only full of bugs. Lots of bugs. This damages your team and / or company's reputation with your users. If your users have alternative software to switch to, you could be at high risk of losing them.

One application built by my team is called Toolkit and is for employees to better serve our members with all their financial needs. Toolkit is a complicated Web application that connects multiple internal credit union (CU) systems and external vendor systems into a single point of contact with the goal of minimizing the number of applications employees need to use. Since our user base for this application is employees, our stakeholders and unofficial product owner for this application is another department within the CU. This department performs all UAT for Toolkit and manages communication to staff for new features and changes. As Toolkit began to take off in popularity and the vision of what it could do to help employees expanded, our backlog of features grew and along with it the pressure to push it out the door faster. This led to our internal quality assurance (QA) spending less time on regression testing and the software team relying heavily on UAT to find bugs or missed requirements. As the main source of testing started to shift to the UAT group, they began to find more and more problems that should have been caught prior to it being handed over to them. This led to stakeholders pushing back on tight deadlines in order to allow them enough time to do deeper testing which should have been completed earlier in the process. Toolkit has become such a core application for employees that if we ship a new release with a major (or even sometimes a minor) bug in it, it can cause a lot of turmoil for staff resulting in an increase of help tickets and phone calls from frontline staff to back-office support staff.

2.2 Release Tracking

Our team also suffered from some disorganization internally with tracking features and bug fixes. While we use JIRA (Atlassian 2016) for issue tracking, for a long time we did not have it setup to easily connect the user stories and bug reports to code and builds. Inside JIRA we mostly used good versioning but it wasn't consistent and we struggled to know when our manual versioning of issues lined up with the actual code releases.

Unfortunately, this led to the occasional big miss in our release notes that our stakeholders rightfully did not appreciate. Without reliable ways to track anything back to a point in the code base, it was essentially a guessing game at what was being included in a release. Stakeholders were regularly finding items during UAT that were not expected and they always noticed when promised items were missing. This poor ability to track items back to specific releases also made it hard for internal QA to know what to test and look for before shipping anything to UAT or production.

2.3 Environment Setup

Toolkit interfaces with multiple third party systems to centralize where employees need to go in order to service members. This includes systems like SharePoint to more complex systems such as our core credit union software as well as external vendors that manage credit cards, credit checks, check ordering, and electronic signatures. It is also connected to other applications we wrote that manage debit card ordering and rates for various account types. My software team only manages the Toolkit application itself, the associated two application database and the other connected internal applications. The majority of applications and vendors Toolkit interfaces with are managed by other teams or by the vendor themselves. There are many potential points of failure, making it all the more important to have solid testing practices and good test coverage before releasing changes into production.

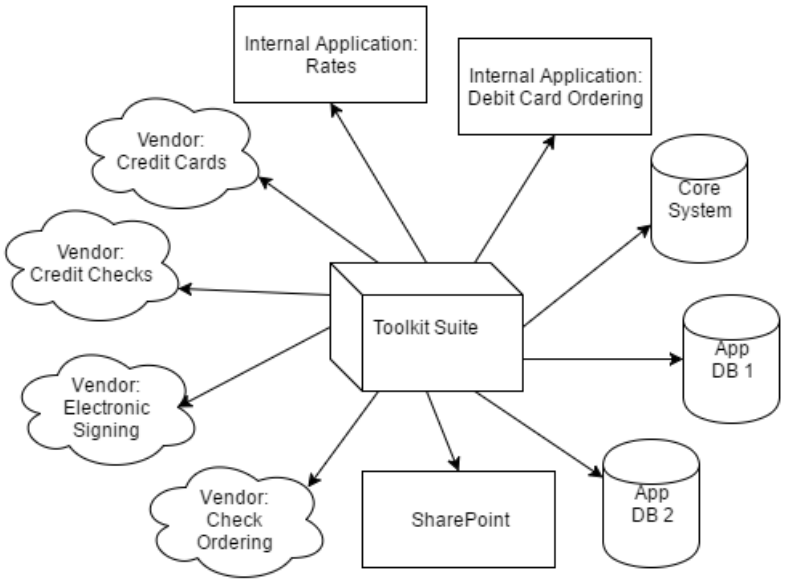

Figure 1: Toolkit Environment

2.4 Stakeholder Confidence

As might be expected, these problems led to our stakeholders, and also our users, losing confidence in both the application itself and the team's ability to produce reliable software. Stakeholders are not guaranteed to be technically savvy and thus can have a harder time understanding or sympathizing with problems with software releases. As stakeholders lose confidence in the software, the trust between them and the software team quickly erodes.

For us this meant that the stakeholders became more demanding as a means to compensate for what they felt was our shortcoming. They demanded more detailed documentation on everything included in a release no matter how small. They demanded ever increasing lengths of time for them to conduct their UAT sessions before they would sign-off on any release to production. Stakeholders recognized our testing was insufficient and would double-check every item in the release notes in addition to performing a full regression test of their own. This was both time consuming and a major duplication of effort given

that both departments were now essentially performing the same tasks on the same software before it was released.

Our department began to recognize the negative effect on our stakeholders when we left the majority of testing up to them and did begin shifting testing back to our software team. However, at this point the damage had already been done where stakeholders had seen too many bugs that should have been caught earlier and had seen too many post-release issues that weren't caught by anyone. They had already lost confidence in our software testing and release process.

Their response of demanding more UAT time for them to recheck all our work before a release slowed down our release cycle time. Slower release cycles is the opposite of shipping software faster and more frequently and caused extra tension between departments as we each pushed for what felt like opposing goals – greater testing versus shipping software faster.

3 Solutions

While my team did a multitude of process improvements to address the problems outlined in this paper, one big push has been in creating an automation system that can help elevate some of the testing pressure jointly felt by our stakeholders and our internal QA. The automated system design had at its core the goal of involving stakeholders so that they could be invested in the testing and gain a deeper understanding of what is being tested. The hope was that this involvement would increase their confidence in our software, our processes, and ultimately our team. This would also help rebuild the trust between our departments.

3.1 Technical Considerations

In building any automation system, you will have unique technical considerations to keep in mind. This is based on your specific infrastructure and resource availability. In the next section I describe the things we had to take into consideration when designing our framework.

3.1.1 Complex Dynamic System

The main part of our infrastructure includes the core vendor system which all our applications integrate into. This system has a nightly process and is not easily setup or mirrored to new environments. Because of the overhead with maintaining any test environment for our core system, our software team was not granted an isolated, dedicated test environment for our needs.

Due to the complexity of the financial world and our systems, the data needed to properly test is not easily replicated in test form so we often have to fall back to having to use copies of real data for many use cases. The test environment we are assigned to use is also used by other departments regularly which results in data moving around a lot without our knowledge. Test data you are using today is not guaranteed to still be in place and setup the same tomorrow.

This setup can create hang-ups in regards to any automated testing. We only have a couple of dedicated test accounts to use which is the case in most test environments. These accounts only cover a couple of common scenarios leaving huge gaps in test data needs. Some test data can easily be created on demand for more simple testing needs. Again, the more complex scenarios need to be using copies of real world data. Given we are a security minded institution, we will not commit into our code base any actual test data or member data that is classified as personally identifiable information (PII). This means that our existing automated tests are frequently breaking because the shared test accounts are used and require regular maintenance. Scaling out a new automation test system needed test data that would be available in a more reliable manner without risking member information.

3.1.2 Maintainability

Due to our own limited QA and development resources, it was not feasible to build a new automation system that would require a high level of maintenance. This meant we needed to be sure that test data was dynamic and not hard coded anywhere so that it could be guaranteed to work. The system needed to be smart enough to skip tests if test data was not able to be found or created on the spot for a given scenario. The tests picked for automating needed to take into account how frequently the application is updated and changed. The features to automate tests for were prioritized based on the likelihood of the feature being updated to minimize the immediate need for maintenance.

3.2 Automation Architecture

The following automation system, named Browser Bot, was designed as a solution to the major concerns described in this paper.

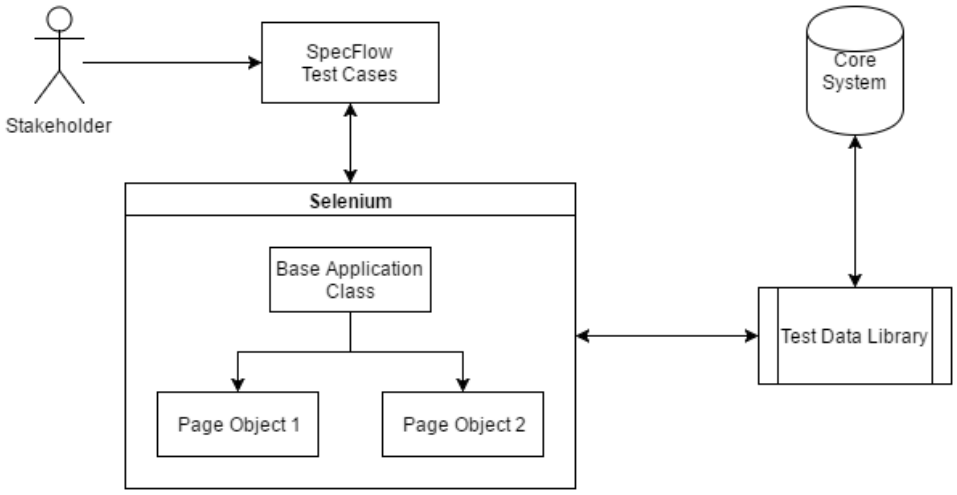

Figure 2: Browser Bot Architecture

3.2.1 SpecFlow

We picked SpecFlow to be the entry point into the automation framework. It is a tool that "aims at bridging the communication gap between domain experts and developers by binding business readable behavior specifications to the underlying implementation" (SpecFlow – Cucumber for .NET 2016). With the main goal in mind of building stakeholder confidence, SpecFlow provided an easy way to pull business people more into the software process. Stakeholders are now able to write the tests themselves using their knowledge of requirements, business processes, and real world scenarios. Besides generating an investment from stakeholders in the testing process and building confidence in the system, this partnership also leads to better test scenarios and greater test coverage of the application.

Tests are written using the Gherkin language in the Given-When-Then format (Cucumber 2016). This is a "structured format for expressing scenarios with example data, including pre- and post-conditions" (Gorman and Gottesdiener 2012). It simplifies the process of project stakeholders communicating requirements using business domain language. Involving them in this process generates an investment on their part into the software development cycle and ultimately results in greater confidence in the software being produced.

```
Feature: BuildVerification
    Verify that the build is stable enough to run other regression/acceptance tests

Background:
    Given Phoenix is in Day Mode

@BVT
Scenario: Load Toolkit
    Given I want to use Toolkit
    When I navigate to toolkit
    Then Toolkit loads without error(s)

@BVT
Scenario: Load Member into Toolkit
    Given I have Toolkit open
    When I search for a Personal member
    Then the member should be loaded into Toolkit
```

Figure 3: SpecFlow test case examples using Gherkin syntax.

3.2.2 Selenium

Since our team produces 99% Web based software, Selenium was a natural choice for an end-to-end software automation framework (Sundberg 2011). Selenium pairs with SpecFlow to run all the tests written by stakeholders in the browser (Heppinstall 2012). Selenium is not always the fastest or most efficient automation tool. It runs at the UI level vs at a lower level that can test specific units or functions in the code itself. This can make it harder to trace back bugs that are found due to the number of layers involved in digging into the application to find the code with the problem in it. However, for the purpose of essentially duplicating user and manual testing efforts with the bonus of having a nice visual that can be shown on demand to business people, Selenium is perfect. Business people can see the tests being run automatically. They can see the work that would take them hours or days to perform being done in a fraction of the time. This is a wonderful way to wow the stakeholders as well as demonstrate that the work they would normally do is already being done for them and done faster. Since they wrote the test cases themselves, they can have confidence that what they feel is important to test is being done before shipping the software.

In alignment with the goal of maintainability, the Page Object Model was used when designing out the Selenium part of this architecture. This model creates a page object for each Web page of the application. This page object is then responsible for owning and finding the needed elements contained within it. Using this design aids in creating “a clean separation between test code and page specific code such as locators (or their use if you’re using a UI Map) and layout” (SeleniumHQ 2016). It also provides simplicity when the user interface (UI) ultimately changes:

> *“The benefit is that if the UI changes for the page, the tests themselves don’t need to change, only the code within the page object needs to change. Subsequently all changes to support that new UI are located in one place” (SeleniumHQ 2016).*

3.2.3 Test Data

In an attempt to solve multiple test data problems with a single solution, we created an internal NuGet package to dynamically pull data out of our core database system. Our core system does not officially support direct SQL connections and rather must be accessed through a proprietary XML based language which enforces business logic and data integrity. This adds a layer of complexity to querying the core database for test data. Placing the code to pull back dynamic data based on test criteria in a central location streamlined the efforts for acquiring the data. It also allowed it to easily be used by both Browser Bot and any unit or integration tests created by the rest of the software team.

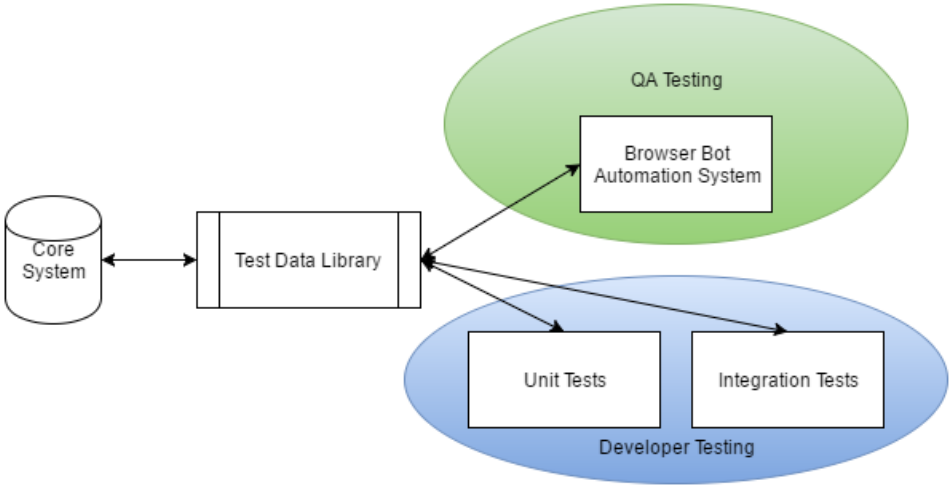

Figure 4: Central library to pull test data from core system can be used by anyone on the team, not just QA.

3.2.4 Build System Integration

Our team uses TeamCity, custom F# scripts and FAKE for running tests and publishing software to any environment (JetBrains 2016) (Hanselman 2014). We added the ability to run Browser Bot tests via these custom scripts. Due to UI based end-to-end tests taking longer to run, a special build in TeamCity was created to run Browser Bot tests on demand without interfering with any of our other builds or day-to-day tasks.

3.2.5 Selenium Grid

A Selenium Grid was setup on internal virtual machines to support running the SpecFlow tests (Colantonio 2014). The Selenium Grid server / hub was setup to live on the build server with TeamCity and stands ready for whenever the TeamCity Browser Bot build is run. Nodes for the grid were setup on virtual machines, each supporting a variety of browser options.

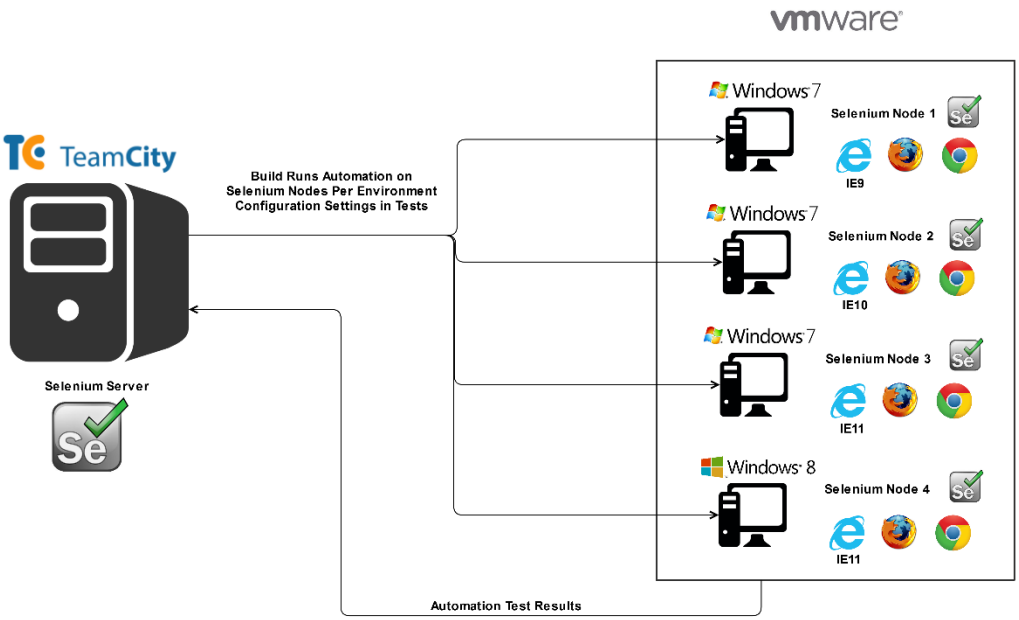

Figure 5: Selenium Grid Infrastructure

After completing the build, both a SpecFlow / Nunit report and a Pickles report is generated. The SpecFlow / Nunit report is more technical and helpful to the development team to troubleshoot errors and failed tests. Pickles "is a Living Documentation generator: it takes your Specification (written in Gherkin, with Markdown descriptions) and turns them into an always up-to-date documentation of the current state of your software - in a variety of formats" (Pickles 2016). This report is great for providing to stakeholders the following: what tests are being run, how those tests are written, what they mean, and what the test result was for them. It's very end-user and business friendly.

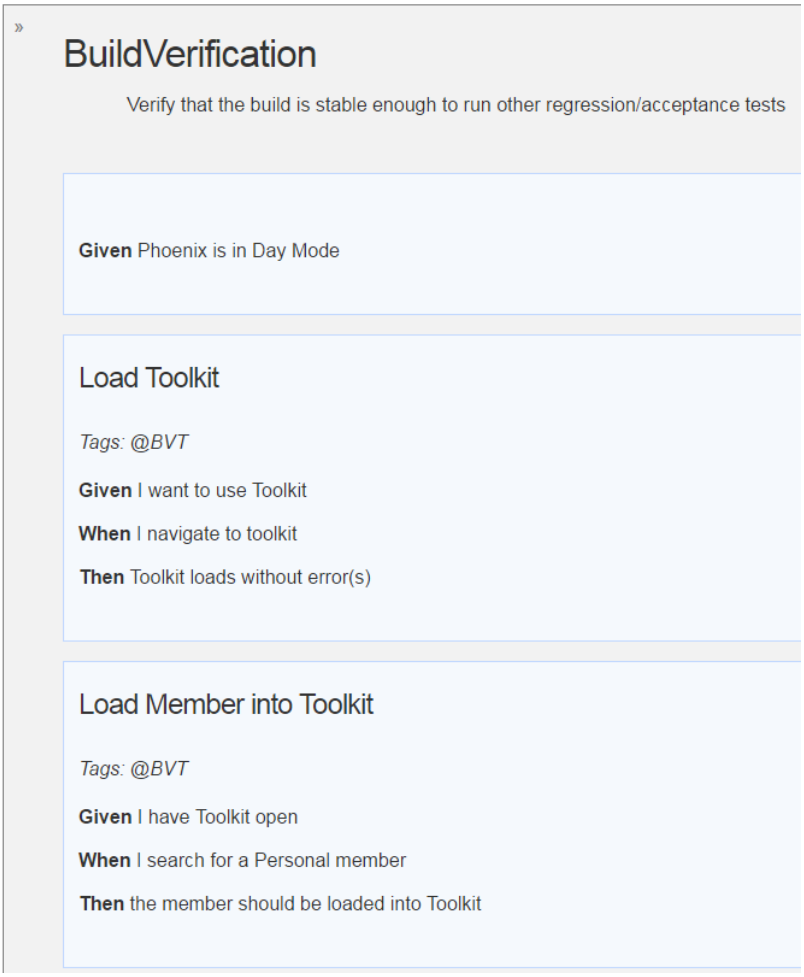

Figure 6: Pickles report generated after running SpecFlow tests through TeamCity

4 Results

While this system is still in its infancy stage on our team, we have already seen positive results from the partnership created out of it. During the initial development of the system, a representative from both the software team and the stakeholders were meeting weekly to learn the automation framework tools and educate each other. The software team was teaching stakeholders about the technical setup and infrastructure of the software. The stakeholders were teaching the software team more about the business rules and real world examples of how the Toolkit is used. The greatest educational piece was in learning how to better communicate with each other; building that mutual trust and understanding that fosters a stronger relationship between teams.

This trust was immediately seen during a post-release fire impacting live users. Previously a problem in the live environment would have caused panic from stakeholders and misunderstanding of the depth of

the problem. Following the start of the work our departments were doing together for the automation system, the reaction was drastically different. Our stakeholders acted with calmness and even helped think through what could be the issue so that the problem could be resolved quickly. This experience helped reinforce the efforts being put forward with the automation system. While there are more efficient ways to automate software than going through the UI, the side effects of picking this path have far outweighed any cons of this solution.

5 Future Work

Going forward our plan is to continue to expand this system. This will include adding tests. Better reporting also needs to be implemented so that it includes integration with our logging system Seq, screen capture on error and even greater Pickles formatting (Seq 206). We will be investigating possible solutions for the Selenium Grid machines using vendors such as SauceLabs since the maintenance of the virtual test machines can often detract from the usefulness of the framework (Gochenour 2016). The goal is to get all our applications using this automated framework and building that same trust relationship with all our stakeholders.

6 Conclusion

Any team who desires to ship software quickly and frequently and has struggled with building confidence in their stakeholders or who have lost stakeholder confidence can benefit from creating a similar system. Building an excellent partnership with your stakeholders and including them in the development of an automated test suite can boost their confidence in your test coverage and the quality of your tests. Less time will be needed during UAT for manual testing efforts helping your team to deliver high quality software with a quick release cycle. Your stakeholders, users, management, and development team will be grateful for the increased trust and strengthened relationship that is created between teams. The benefits are huge and the reward is better quality software!

References

Atlassian. 2016. “JIRA Software.” Atlassian. https://www.atlassian.com/software/jira (accessed July 31, 2016).

Colantonio, Joe. 2014. “Selenium Grid – How to Easily Setup a Hub and Node.” Joe Colantonio: Automation Awesomeness, entry posted October 7. https://www.joecolantonio.com/2014/10/07/selenium-grid-how-to-setup-a-hub-and-node/ (accessed July 31, 2016).

Cucumber. 2016. “Gherkin.” Cucumber GitHub Wiki, entry last edited June 10. https://github.com/cucumber/cucumber/wiki/Gherkin (accessed July 31, 2016).

Gochenour, Phil. 2016. “C# Test Setup Example.” The SauceLabs Cookbook, entry last edited April 5. https://wiki.saucelabs.com/display/DOCS/C%23+Test+Setup+Example (accessed July 31, 2016).

Gorman, Mary and Ellen Gottesdiener. 2012. “Using ‘Given-When-Then’ to Discover and Validate Requirements.” Success With Requirements, entry posted December 9. http://www.ebgconsulting.com/blog/using-given-when-then-to-discover-and-validate-requirements-2/ (accessed July 1, 2016).

Hanselman, Scott. 2014. “Exploring FAKE, an F# Build System for all of .NET.” Scott Hanselman Blog, entry posted March 14. http://www.hanselman.com/blog/ExploringFAKEAnFBuildSystemForAllOfNET.aspx (accessed July 31, 2016).

Heppinstall, James. 2012. “Behavioural testing in .NET with SpecFlow and Selenium (Part 1).” James Heppinstall: On Development, entry posted September 24. https://jamesheppinstall.wordpress.com/2012/09/24/behavioural-testing-in-net-with-specflow-and-selenium-part-1/ (accessed July 31, 2016).

JetBrains. 2016. “TeamCity.” JetBrains. https://www.jetbrains.com/teamcity/ (accessed July 31, 2016).

Pickles. 2016. “What does Pickles do?” Pickles: Living Documentation. http://www.picklesdoc.com/ (accessed July 31, 2016).

SeleniumHQ Browser Automation. 2016. “Page Object Design Pattern.” Selenium Documentation, entry last edited April 15. http://www.seleniumhq.org/docs/06_test_design_considerations.jsp#page-object-design-pattern (accessed July 31, 2016).

Seq. 2016. “Machine data, for humans.” Seq. https://getseq.net/ (accessed July 31, 2016).

SpecFlow – Cucumber for .NET. 2016. “Getting Started.” SpecFlow – Cucumber for .NET. http://www.specflow.org/getting-started/ (accessed July 3, 2016).

Spolsky, Joel. 2000. “Top Five (Wrong) Reasons You Don’t Have Testers.” Joel on Software, entry posted April 30. http://www.joelonsoftware.com/articles/fog0000000067.html (accessed July 1, 2016).

Sundberg, Thomas. 2011. “Testing a web application with Selenium 2.” Thomas Sundberg Wordpress Blog, entry posted October 18. https://thomassundberg.wordpress.com/2011/10/18/testing-a-web-application-with-selenium-2/ (accessed July 31, 2016).