PROFESSIONAL TESTER

Essential for software testers

Cool tools to transform testing

43

December 2017

Including articles by:

Evgeny Tkachenko
EPAM Systems

Huw Price
Curiosity Software Ireland

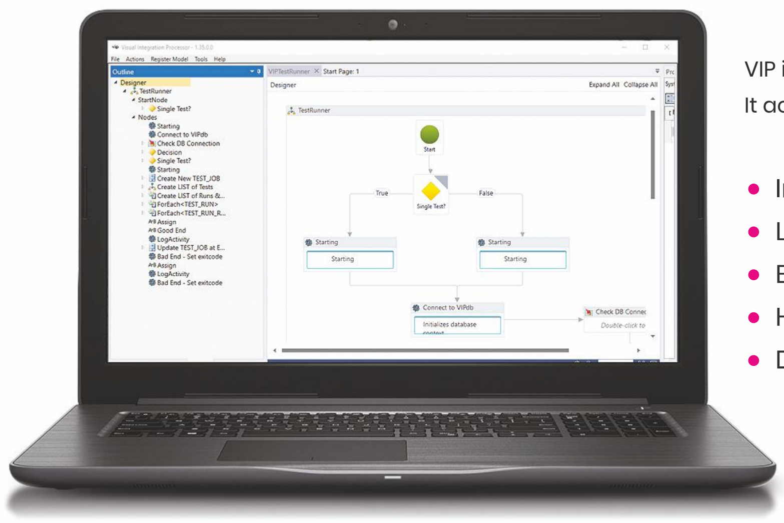
Justin Watts
Loblaw Digital

Johan Steyn

Viv Richards
Vizolution

Visual Integration Processor

Where have you been all my life?



VIP is a lightweight framework to help you build better DevOps.
It acts like DevOps glue, so people and tech can work together seamlessly.

- Integrate, monitor and control your software factory
- Let your systems talk to each other with ease
- Build dependency maps as you test
- Harvest traffic to drive AI
- Drive everything using ChatOps

Just add imagination

curiosity SOFTWARE IRELAND

curiositysoftware.ie

PROFESSIONAL TESTER
Essential for software testers
Cool tools to transform testing
# 43
December 2017

Editor Vanessa Howard
editor@professionaltester.com

Managing director Niels Valkering

Art director Christiaan van Heest

Publisher Jerome H. Mol

Sales advertise@professionaltester.com

Subscriptions subscribe@professionaltester.com

Editorial board
Professional Tester would like to extend its thanks to Lars Hoffmann, Gregory Solovey and Dorothy Graham. They have provided invaluable feedback reviewing contributions. The views expressed in Professional Tester are not those of the editorial team.

Professional Tester is published
by Professional Tester Inc

We aim to promote editorial independence and free debate: views expressed by contributors are not necessarily those of the editor nor of the proprietors.

© Professional Tester Inc 2017
All rights reserved. No part of this publication may be reproduced in any form without prior written permission. 'Professional Tester' is a trademark of Professional Tester Inc.

How cool tools can transform testing

It is always interesting to hear from people with a long pedigree in the software industry, particularly when they take a new direction and offer a fresh perspective on testing.

Huw Price was VP at CA Technologies, has founded five companies over the last 30 years, and he believes that recent developments in the vendor marketplace and open source mean that it is now possible to simply chain together components to quickly assemble solutions. He sets out a step-by-step guide for building what he characterizes as truly reactive DevOps.

Problem-solving dominates the rest of the issue too. From avoiding classic test automation pitfalls to building a visual testing framework, our contributors are meeting everyday challenges and creating new approaches so better outcomes are within reach.

If that deals with today, Johan Steyn looks at the skillsets testers may need tomorrow. A timely reminder as we reach the end of the year and look ahead to the next.

As always, we hope you enjoy the magazine and the team at Professional Tester would also like to take this opportunity to wish all of our readers a happy and prosperous 2018.

Vanessa Howard
Editor

12 reasons why test automation can fail

by Evgeny Tkachenko

How to identify and avoid the common factors that hold back test automation success

If you have thousands of automated tests, it means nothing if those tests fail to check what is needed.

During my career as a QA consultant, working across web, desktop and mobile projects, I've encountered common reasons why attempts to introduce automation failed. Here, I set out what they are and what can be done to avoid them.

Stumbling blocks at the outset

1. Insufficient budget

Test automation (TA) consumes time and money. Some TA projects fail before they begin because of cost restraints. I do hear remarks such as: "Budget shouldn't be an issue if companies stick with open-source solutions for their automation needs", but test automation is not just about which tool you use. To succeed you need skilled, experienced automation engineers, continuous delivery/continuous integration systems, machines/servers, an environment dedicated to automated tests (their own device test stand or cloud solutions, like Sauce Labs, AWS Device Farm, etc.) and all of these demand investment. Costs can't be avoided and a long-term perspective is essential if organizations are to evolve.

2. Wrong tool

It takes a lot of time to evaluate a relevant automation tool, but it is worth the effort. It helps to ask the following questions:

- Does the organization have the necessary skill sets available?
- What is your budget?
- Is it suitable for the project environment and technology you are using?
- Is the tool version stable?
- Which testing types (load testing, functional testing, etc.) does it support? Choose the tool according to the testing types of your application needs.
- Does the tool support easy interface to create and maintain test scripts?
- Does the tool support a data-driven paradigm? When choosing an automated testing tool, check which data formats it can use, such as text files, XML files, database tables, and others.
- Does it provide good reports?
- Does it integrate with your other testing tools like project planning and test management tools?

Managers (who have never dealt with test automation and do not have QA engineers who are experienced in automation in their team) usually choose the easiest way to start a test automation - buy a “record and playback” testing tool. It is a seductive option because you don't need to have programming-skilled engineers to start automating your tests, you get fast first results, it usually includes logging and reporting functionalities, runner, etc., you don't need to build it by yourself and your first automated tests are soon ready.

But maintenance can be very expensive. Tests captured by the tool are usually fragile and even minor changes in the application under test (AUT) can require that all tests need to be updated or re-recorded. This happens because tests are often dependent on the precise placement of UI objects and may be affected by screen resolution. With such tools, you are mostly tied to use its own runner without remote execution, parallelization, configuration, data driving, etc.

I have witnessed several failures where automated testing began with an expensive “record and playback” testing tool. The project appeared to begin well - a smoke testing suite which consists of 18 UI automated tests was ready in a week, the run took only 25 minutes instead hours of performing them manually. It was an "epic win". After a month they had more than 100 automated tests. But soon they started noticing that they spent a lot of time updating existing tests to make them reflect changes in the AUT. Two months later, when they had more than 250 automated tests, about 50 of them were constantly failing because the QA guys could not sustain the pace needed to develop new tests and keep the old ones in good shape. As a result, they had to stop using this tool and throw all automated tests away as it is almost impossible to switch from a commercial testing tool to an open source one without losing everything you have built.

3. Wrong team

Test automation is not a silver bullet, where it helps is:

- When there are many repetitive tests
- When there are complex validations (data in databases, big data testing, API, etc.)
- When there are frequent regression testing iterations
- When you have a large set of test cases (automated scripts run much faster than manual executions)

I was asked a million times: “How many QA engineers do I need for my project if I have X developers?”. There is no easy formula. It depends on the complexity of the project, software development methodology, tools and technologies used. But based on my experience, I can safely say that if you work in a "true" agile world (with requirements analysis and automated tests written in advance) then you need to have a ratio 1:3 (QA:Dev). An ideal situation is when you have a ratio 1:2 but I have never had such "luxury" and I always had to optimize the efforts and change a QA-DEV process to fix this performance bottleneck (testing usually is a "bottleneck").

An ideal set up would be one senior QA automation engineer responsible for building or/and supporting of a testing framework, writing low-level methods (bricks) which other engineers can use to write automated tests on a higher level. I call this position “developer in testing”.

Two junior/middle-QA-automation-engineers-transformers who are not squeamish to do a test design, analyze requirements and test something manually. They can do everything and could be owners of any product stories and they lead them throughout the whole life cycle from a requirement analysis stage, test design, writing and performing automated tests and up to validating a feature in the production environment.

What you want is to eliminate misunderstandings amongst those who analyze requirements, those who test manually and those who automate. People who take responsibility for the success of the user story or functionality. I call these guys, able to write automated tests using the bricks which have been prepared, “Swiss Army testers” or “QA transformers”.

You also need a QA Lead who builds a test automation plan and adjusts it accordingly. This person has to be technically-skilled and have experience in introducing test automation. They need to truly know the project, its dependencies, and architecture, to consult with others and help with test design, requirements analysis, and even a code review. They should lead and drive the automation process, I call the position "Lead driver" or “QA Optimus Prime” because they have to lead and guide other “transformers”.

This QA team of four people can successfully handle a workload produced by 10-12 developers. A 16-people team (12 Dev + 4 QA) does not look like a scrum, as it should not exceed 9 people. For instance, if we have 16 people on the initiative, a scrum requires splitting into two or three teams. But it works in Kanban because we don't have any limits on team size as well as in Scrumban. Also, this QA team of 16 people could support several projects and act as one huge Scrum team.

Based on my experience, it is not a good idea to ask developers write GUI and integration tests. QA engineers should have specific analytical skills and have to have a good expertise in test design. Developers should be responsible for unit tests only, of course, they could help QA engineers make the AUT testable. If the product is huge and it has a lot of dependencies on third party services - there should be a separate QA team which should analyze requirements and help developers to avoid introducing bugs by preparing test data and discussing acceptance tests and write automated tests and maintain them.

Test automation is not a "fire and forget" operation. Automated testing needs to be continuously updated to manage changing components (controls) and feature details. And especially in case of GUI automated tests, it can take a lot of DEV efforts.

4. Bad planning and unrealistic expectations
A good plan must describe what will be automated, when and by whom. It must cover all types of automated tests: unit, integration and functional. Organizations cannot invest into introducing automation thinking it is the bug-finding silver bullet. Yes, automation can free up time spent in regression testing and re-testing, and help focus on requirement analysis or exploratory testing, but to gain real advantage you have to gain a sufficient test coverage at first place. If you don't share this vision with stakeholders, you could be in trouble. The plan should contain milestones, including smoke testing suite, sanity testing suite, functionality covered, where the "biggest pain" is, regression testing suite, new features. You can always ask a product owner about functionality priorities and create a risk matrix which helps to determine where to start. The main goal of this procedure is to identify where we can benefit from automated testing.

As well as helping to secure success, listing these milestones and reporting on progress will help to get rid of another problem - lack of visibility.

How we maintain automated tests

5. Automated tests which are poorly written
There is no single right way to build a testing framework, but there are several wrong ones. "Quick and easy wins," can look seductively cheap, but not when they are impossible to maintain. Try to avoid copy-pasting and duplicating code and creating preconditions for tests via UI, especially via UI of other projects, make your tests independent from each other.

UI automated tests are dependent on the AUT. The UI of your application changes over time and these changes affect the test results and you have to update your scripts to keep your tests "green", otherwise, your tests will not find controls to interact with and will result in a false fail. Sometimes this procedure of reviewing failed tests turns into a nightmare, especially if locators are spread out throughout the testing framework, or if your tests rely on location coordinates to find the control.

To avoid this mishap, in the requirements analysis phase, automation testing engineers should specify on the mockups which locators (IDs or special parameters) developers should add for which controls. This provides a QA with an ability to write UI automated tests before implementation, and the fact that locators will not be affected by DOM changes of web pages and will make the tests more stable and easier to maintain. Manage the changes in advance instead of reacting to them post-factum.

6. Hard-coded data
Do not hard-code input data. Instead of specifying exact values of parameters, use scripts (or API calls) which find or generate an appropriate test data in a database. It will make your tests flexible, it is easier to keep scenarios up-to-date and make tests independent from the environment you use. Also, it helps to avoid the "pesticide paradox" in your application - the phenomenon where the more you test, the more immune the software becomes to your tests - just as insects eventually build up resistance. Do not use the same data for testing all the time. If you have forms which you fill out with data, for instance, a name field, instead of entering something like "Bob", "John" or "Test" use libraries (like javafaker or build your own to generate random data) and specify randomizer.name().firstName().

7. You have a lot of automated tests, but a full run takes too long
I have taken on several projects where automation failed because the test runs took too long (in one instance, it took 3 weeks). Every morning began with the launch of different bunches (suites) of tests and analyzing the results, every day was Groundhog Day. If those tests found issues then it was impossible to repeat the whole regression suite after they were fixed (due to lack of time) and as a result, some bugs were leaking to the production environment.

Make your automated tests independent from each other, do not use the same data (users, accounts, organizations) and make it possible running tests in several threads simultaneously. Also, do not duplicate steps of automated tests which are not related to the functionality - in other words if you need to prepare preconditions for your automated tests do it in the “cheapest” way.

For instance, if we write tests for a cart functionality, then all steps (methods) which describe how to find an item to purchase have to be located in separate test cases for the searching functionality. If you test cart functionality, don't cover tests searching functionality again because it should be already covered by independent separate automated tests, go directly to the item using URL address or even generate an order using back-end (API) and go to the cart. Of course, you will need to have at least one end-to-end scenario automated which covers the whole purchasing flow, but most of the automated tests should be focused on one particular functionality and use preconditions which do not duplicate (overlap) other scenarios. On top of that, if you need to change something in the searching scenario then you don't have to look through all functionalities where you could use these steps (methods) - you just need to change them in one place (in tests for this functionality).

If you find (during requirements analysis) that it is impossible to automate the testing of a functionality due to some kind of restrictions in the system you can always ask a product owner to include a task for developers to implement something (like handler, dummy data or other workarounds) to overcome this barrier and make it testable. For example, we have a requirement which says: "We have to implement a registration functionality in our web application". This type of page contains a captcha which protects it from brute-force attacks. That means it will be impossible to automate verification of this functionality. But we can ask to add an additional task to provide an ability to bypass it for automated tests in testing environments exclusively. It could be an environment property which disables the captcha or it could be a static key you can use to put in the captcha field to pass the validation.

A lot of teams start test automation with the part of the application they spend most of the time on - UI. But running of this type of automation is time-consuming. A run of 100 integration (API) automated tests takes seconds while a run of 100 UI automated tests can take hours. Also, UI automated tests are less stable and costlier to maintain than integration ones. Be focused on unit tests (should be a developer's responsibility) and integration test coverage, adopt a “pyramid testing” strategy. And finally, if you do need to have a lot of UI automated tests then parallelize your automation run.

How we use automated tests

8. Poor test coverage

Poor test coverage, or test coverage which grows too slowly, can't sustain the pace needed to implement new functionalities of the AUT. A tight deadline encourages us to say: “We will test it manually now and we will cover by automated tests later.” This “later” never comes because other deadlines soon arrive and this technical debt accumulates. Postponing test automation risks not demonstrating its effectiveness but a fail before it even begins.

9. We have a lot of automated tests, but we don't run them often enough

Run your automated tests as often as possible to make sure that the AUT and your tests are in good shape - especially if you must integrate with other services not under your control. You should know precisely when a problem started appearing in your app and without frequently launching automated tests it is almost impossible to do so. Projects with automated tests but without test automation are at risk. Automated tests are just scripts which help you avoid manual testing while test automation is all about automating the process of tracking and managing the different type and level of tests. Automated tests alone are almost useless without proper application.

10. We have a lot of automated tests, we run them but we do not analyze the results

A lot of teams have a culture of ignoring failing tests or a red continuous integration run. Often it is a result of a lack of trust in the tests. I see it almost all the time - teams with “that test sometimes fails when nothing is wrong and there's nothing I can do” and soon people are ignoring real failures.

The best way to fix this situation is to make your tests trustworthy again. If the test is bad - fix it or delete it. If the CI system has a bug fix it (or find a new CI). If there is a broken feature fix it.

11. We have a lot of automated tests, and they pass every time

Poor coverage of real-world cases is the most common and obvious reason why test automation fails. It is obvious that the purpose of automated tests is not to find new defects but rather to find regression bugs after implementing a new feature. Despite that, there are a lot of cases in which regression problems slip through to the production environment. When we start relying on automated tests we have to be sure that they cover every important aspect application functionality:

- All services respond properly (integration tests)
- There is no performance regression (performance tests)
- A layout is not broken (automated visual testing)
- The application behaves according to requirement (functional UI tests)

But even if you have everything from the list above covered it does not guarantee that you will not miss a bug. If you have thousands of automated tests, it means nothing if those tests fail to check what is needed. You will never reach success with test automation if you are not good at manual testing. Not having an understanding what you should test means not being ready to automate your testing process. If you automate rubbish – you get nothing but automated rubbish.

It is always a good idea to have your test cases reviewed by a business analyst or product owner prior to automation, they will be able to provide feedback and help determine what is the most important aspect and what should be covered. It is better to not have automated tests at all than have some you can't fully rely on. It will lead you to the "fake" confidence in your automated tests and, as a result, you won't perform manual tests for this functionality and miss defects.

12. Test automation = automate (manual tests)
Even after introducing test automation, QA teams have been known to carry on writing test cases in the same way as before without making any changes. First of all, you need to find how to prepare preconditions. Also anyone who writes test scenarios for automation should understand what really should be tested in the scope of the current test case and what could be skipped or simplified. It is also vital to include all validations which make sense and exclude all unnecessary ones.

Some people try to embrace everything by validating everything on every screen of the application (including text, a location of an element in the DOM, size, and style of elements) in one functional automated test scenario. A good test scenario for automation should be short and specific, for instance, if you want to check what controls look like on the page use an automated visual testing tool, do not include it in the functional automated tests. The test case must not contain more than 15 steps (actions + validations) otherwise It will be difficult to maintain, and the test could fail before reaching the functionality it was intended to test due to some problems in previous steps.

On the other hand, you should include all validations which matter for this functionality. One team, which thought it had 100% test coverage, missed a bug which cost the organization thousands of dollars. The QA team had designed automated tests for each part of the functionality. They had separate automated tests for searching, cart, checkout and reporting. To test the cart functionality, all orders were generated via API as well as for the reporting functionality, all tests were independent and fast. But during refactoring of code, a bug was introduced which caused the wrong behavior: on one of the steps purchasing flow (adding an item to the cart or checkout or sending an order to be processed) an additional fee was added to the total price if a user did that through UI. As a result, a lot of users have been overcharged. This issue could be found only if the team created at least one automated end-to-end scenario to check that the price of the item is the same on the search results page, in the cart, on the checkout page (if no taxes) and in the orders report. It is not good when your automation consists mostly of end-to-end scenarios but you can't afford to not have them at all, you should always find a balance.

Conclusion

There could be other reasons for failure like wrong programming language used, an unstable testing environment, lack of time, lack of support from general management, etc., but all of them overlap with those I have set out in this article. All projects are different and the approach for each should be customized.

The main key to avoid test automation fails is to recognize that test automation is just like product development. It requires the same kinds of preparation, planning, and investigation. Then it has to be designed, managed and properly maintained for the long-term because the test automation project is going to go through exactly the same lifecycle as your application under test ■

Evgeny Tkachenko is QA lead at EPAM Systems, with a background in automation testing and test management on complex projects in the telecommunication and online entertainment industries.

How to harness cool tech

by Huw Price

A guide to building a truly reactive DevOps strategy

Testers should be using this cool tech and they should be able to drag it into their worlds quickly and easily

It seems that every company is currently working on DevOps initiatives, whether they are shifting left, shifting right or doing the hokey cokey. There have been enormous changes in our industry yet the reality is that most organizations are struggling to move faster. Even gains made in speed are not guaranteeing improvements in quality. So, what can be done to ensure that testing cannot only keep pace with demands but reshape DevOps?

So, where are we today?
Over the last ten years I have worked with numerous enterprises and have spent a considerable time implementing quality initiatives and driving a higher level of automation in testing. What is clear is that core problems have remained the same throughout this period: bad requirements, not enough of the "correct" automation, poor and misunderstood testing coverage and very little dependency mapping between software components.

If we look at four categories, 'people, processes, tools and politics', there have been some notable changes. First, on a positive note, as far as people are concerned, testers are now seen as "critical modellers" and are being brought in earlier to help design testable systems. Looking at processes, agile practices are everywhere but I'm afraid the procedures have taken over from the spirit of agile. Politically, in-sourcing is back and working in more connected teams is becoming more common, which is no bad thing.

So that leaves tools. Have they really helped drive better systems? If they have, I am afraid that it is more by luck than judgement. Software vendors tend to focus on their silo and as an afterthought try and connect it to other systems. Some of the consultancy vendors have started to build integrated end-to-end solutions but they tend to be rebadged as consultancy projects and pushed out as software frameworks.

And where do we need to be?
As the testing industry has limped along the software world has changed dramatically and over the last three years, it has exploded into life. With the introduction of Git sharing, vendor marketplaces and open source there is now some very cool tech out there to help build the nirvana of a truly reactive DevOps strategy.

Testers should be using this cool tech and they should be able to drag it into their worlds quickly and easily. But, as the saying goes, everyone is always too busy chopping down trees to buy a chain saw and I appreciate that stopping a release to improve processes is difficult. The cost of changing processes far outweighs the cost of the software, so it is necessary to produce a step-by-step plan with as little disruption as possible.

It can be done and here I'll break down the approach into a few phases. These can be done in parallel but like all good strategies, all steps taken must recognize the end goal and how each phase builds towards that goal.

Expose what you have
If you look at most tasks performed by development teams, a lot fall into repeatable categories, probably 90% of the routine

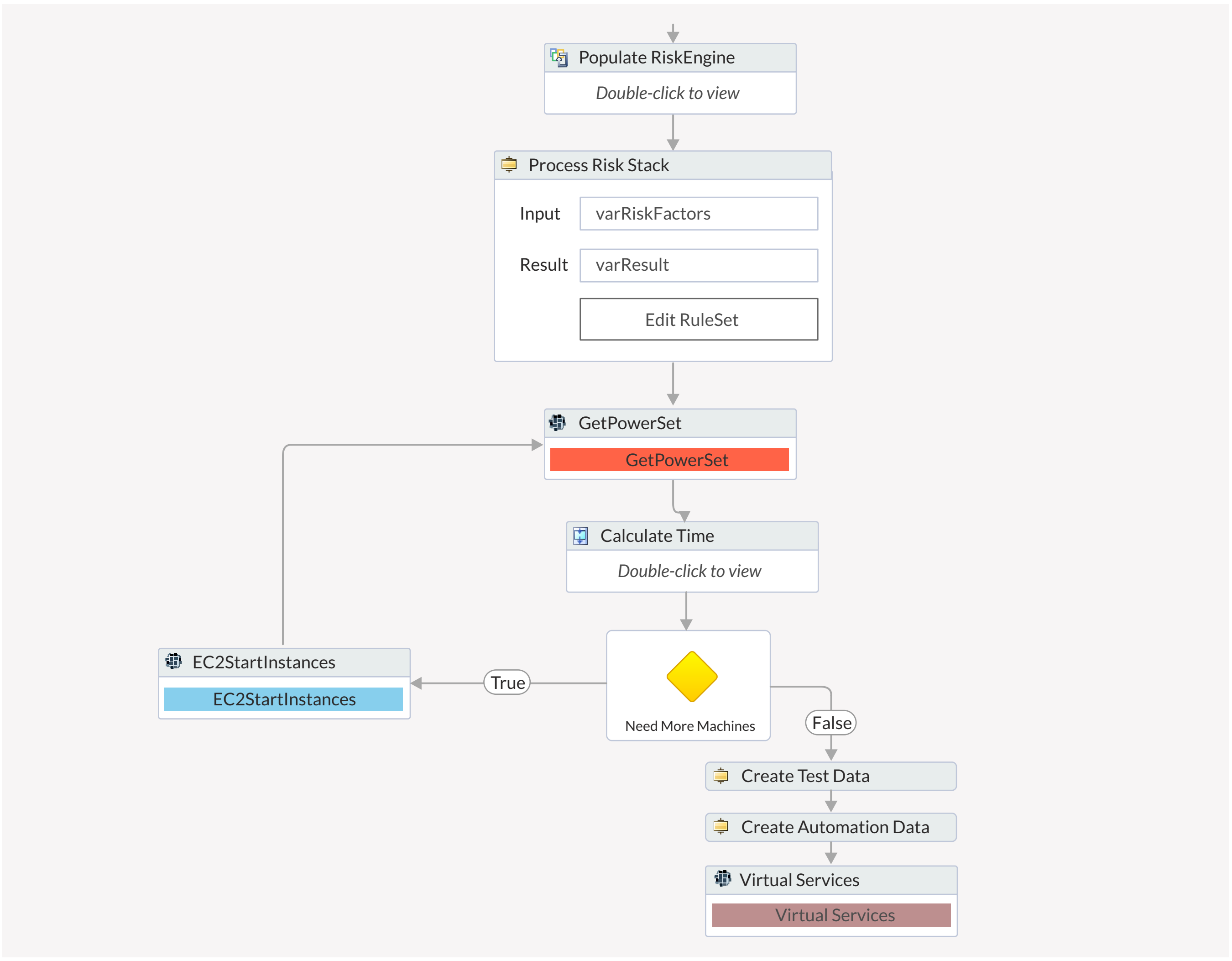


Figure 1: VIP and dynamic provisioning

tasks could be initially exposed and run via a Chatbot. Products like Slack are now omnipresent and as the Slack team point out "context switching" i.e. jumping between products can waste up to 20% of your time.

As new tech processes get created by anyone in the team expose them to Slack and get others to use them. Fire off jobs directly from your primary communication tool and let the tech do the work in the background.

Start taking control

Now you have these processes more widely used and under a common tech stack you can start "gathering" up the information and pushing meta information to command and control systems. For example, you requested a set of regression tests through Slack and some have failed, the developers are automatically notified, and a new environment has been spun up automatically, ready for any fixes to be tested in.

So far you haven't really done much beyond adding in some control and reducing context switching but you can see one action getting a result, which then implies another is starting to bring efficiencies. Over time it will become normal to look for existing jobs and define actions linked to a process. You are now starting to join up process and tools.

Learn from the past

The chances are that an effect has been seen before, for example your payments take on program may start rejecting customer files and marking them as invalid. Has this failure happened before and more importantly what was the root cause of it? In this case it was one of the rule configuration files had changed and some of the combinations had not been tested. This had happened before, so we have a clue as to what might have gone wrong. If we turn this on its head, why don't we automatically monitor if the core files have changed and, first, let people know it is about to happen in the next release then, second, invoke a more rigorous set of automation tests?

In other words, look for root causes not effects and set up automated alerts looking for potential problems before they occur. Linking these problems to a risk factor can then influence which tests are run. Start to think in terms of types of risk as well, this is an important change to the way teams

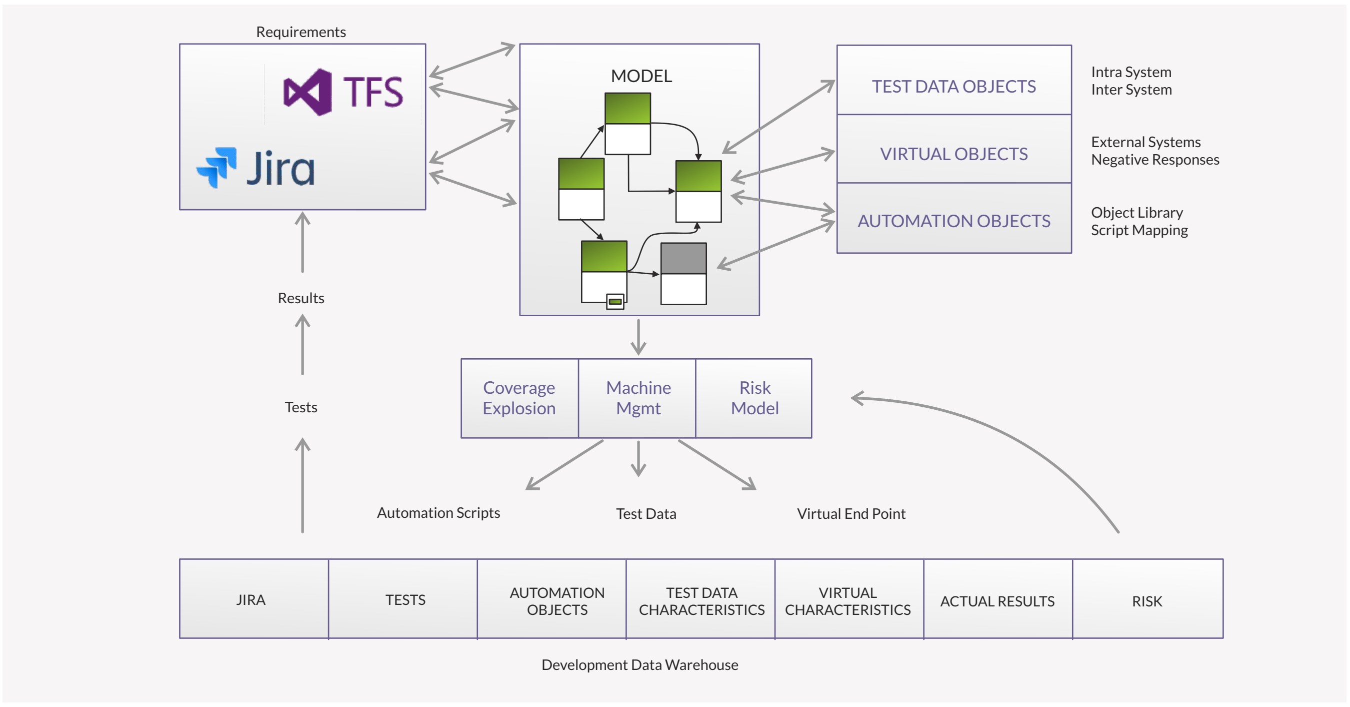


Figure 2: Meta data harvest

think about deploying software and can be used to turn what was once subjective into objectively driven improvements.

Artificial intelligence

There are almost daily briefings from software vendors about how AI is going to revolutionize the software industry and their integrated software using the latest AI will take away human decisions and drive magical speed and quality. The reality is that AI is only as good as the data that is fed in, what meta data is gathered, how each factor is rated and whether the information flows back so AI can "learn" from the past. In a nutshell rubbish in, rubbish out.

The phased approach - where we gradually bring structure to what was previously random - is a sensible way to introduce change and it can then feed into some of the cool AI tech available.

Monitoring

It pays to start monitoring stuff. And, as ever, start with the easiest - you may already have good tools to track code check-ins, link them to specific programmers and your code quality engine. Joining them up is easy as they already have integrations.

Once you have the basics done think about the data that is flowing between the systems. Is it meaningful, in other words, does it contain enough evidence to inform a decision? Over time, will the meta data that you gather be able to monitor a threshold that could inform an action. Spend some time looking at what happened in the past and see if you have enough information to be able to detect a similar type of event before it happens.

Once you look at the problems using a more analytical approach, you can start applying this philosophy everywhere and start spreading out from the basic integrations. Think about how information flows up and down in your development world and whether it can be tracked effectively.

Technical integration and workflows

It is said that necessity is the mother of invention which is why we've been working on VIP, a general integration tool that can glue together DevOps tools, and Figure 1 is an example.

Virtually all testing and development tools have a reasonable API layer that allows data to be read and added easily, what they all lack however is an inbuilt dependency map to help you track anything more than the basics. Few of them take advantage of the new cool tech and they are for the most part difficult to integrate into an AI strategy. This is what a workflow engine like VIP addresses.

In order to harvest information flowing between tools you need to take control of the feed and pass through structured information such as: test ids, stories, users, tags, defect ids, release numbers etc. This information can be passed on down to the next system or, better still, pass the information into a DataMart as it flows through your processes. In effect you are building in enough meta data that other parts of the system can track back to the original code and requirement, as set out in Figure 2.

This connectedness and link building will build a rich enough repository to be able to make informed decisions. The data gathered must also include details on all parts of testing, including the test data characteristics, virtual responses, actual results and expected results.

Use testing to build dependency models
At the core of any clever DevOps strategy is the ability to track dependencies. If I change this line of code, what am I going to break, what is the risk to my system and every other dependent system? The problem of technical debt and a lack of a clear data flow model is massive, and every company struggles with a clear map of dependent code.

A good technique is to use testing to help join the data flows together. If you know the input and output of one process under test, then the resulting output can be the input to another test later on. Mapping this into your DataMart helps you bring the threads together, allows you to predict the effect of a change and lets the AI engines build more accurate sets of test cases.

Reactive automations
A good intermediate goal is to think about not only increasing the level of automation but also to get it to react to changes in the requirements, so a change in a screen, API, or configuration rule should result in a new set of automated tests, data and virtual services - all of which are created automatically.

This sounds like hard work; the reality is you have to do it anyway so spending time linking the automation to meta logic is really the only way to go. The meta logic needs to map business terminology (which is how requirements are worded) into test assets, including test data (both input and reference)

Cool tech can transform testing
As you build up your software factory, initially from a few components on the shop floor, and start building up sets of good data flows that gather rich, relevant and accurate information then you can start linking in some of the cool tools around. Good examples we have added are:

- Using sentiment monitoring (linguistic emotion AI) on team chatter to warn of delays and provide roll ups on the status of threads.

- Linking combinations and permutations to the test definition. Before the test runs it automatically expands the test data to include pairs, triples etc to expand initially to the time and machine power available.

- Getting cleverer with actual results. Testers rarely test all the way down to a specific result.   In effect tests could be said to: possibly, probably or have definitely worked. By tracking actual results over time, you can use clever textual pattern analysis to look at results for similarities and give a success score that can inform go / no-go decisions on a release.

- Data tools. There are many that can be easily incorporated into test dev frameworks, for example: Snapshot roll back, roll forward; fast comparing databases before and after to discover what has actually happened not what you think has happened; synthetic data engines are now very powerful and can be used to prep data for automation runs; data cloning allows real cases to be exploded, masked and moved from production to provide more realistic testing of defects and edge cases.

- Coverage and model engines are now much easier to incorporate and provide great flexibility, for example adding in genetic testing algorithms, constrained triples and model-based tests is now straightforward.

- Adjusting the depth of testing to levels of risk, if you know that a change to this module has caused a defect before (ask the DataMart) then set the risk to high and let the automation mutate and test more rigorously. Rule engines such as Nrules and Reactive LINQ queries which have typically been used for business event handling are perfect for driving the amount of testing, interpreting the results and gradually learning from the subjective skill of the tester.

In summary
It is now time to start driving the agenda if we are to keep up with the relentless and ever-expanding demand on testers. The growth of the API means that complexity increases exponentially, and current processes will struggle to keep up. Think of version compatibility testing between APIs that can be called in any order and any one could fail; how do you test that? In a nutshell it's time for testers to use their analytical skills to drive new testing agendas ■

Huw Price has 30 years of experience as a software inventor and software entrepreneur and is the founder of Curiosity Software Ireland.

Don't compromise on tool selection

by Justin Watts

Hard lessons learnt about what works best when deploying test automation software

When you have a quarter of a million results to sift through it's very difficult to understand what is happening using traditional reporting techniques.

In the last year Loblaw Digital, the development shop of Canada's largest retailer, Loblaw Companies Limited, has made a step-change in its testing scale and performance. We recognised that there were bottlenecks in testing and we set out to examine processes and tools that would help scale our output. Going through the migration process has taught us a few lessons about what works best when deploying test automation software which I'm happy to share.

The challenge
Loblaw Companies Limited is a publicly traded company operating in over 2,300 locations and turning over CAD $46 billion per year. Loblaw Digital creates and delivers omni-channel experiences across physical and digital mediums which include online grocery offerings, e-commerce, loyalty, financial services and pharmacy products. We operate within multiple businesses including Click & Collect for groceries, the award-winning Joe Fresh site for clothing, beautyBOUTIQUE.ca for cosmetics, Shoppers Drug Mart for prescription medications and PC Optimum for loyalty.

Our team
We are a 100-strong technology organization and so like many of comparable size, we are constantly generating code, there is a need for rigorous end-to-end testing and that responsibility falls to our test engineers. Test engineering is supported by our internal tooling engineers and together these groups form the engineering productivity (EP) team. Engineering productivity is an ideology taking aim at traditional QA which shifts the focus from checking quality to generating confidence.

While refined ideology can be wonderful, it does not change the fact that as our businesses grow, our team needs to increase capacity and support running more tests. In order to scale-up, we went looking for the right tools and processes to seamlessly manage, automate and interpret the results.

Defining the requirements
Loblaw Digital uses much of the Atlassian stack including Jira, Bamboo, Bitbucket and Confluence. We needed automated testing to meet the demands of large-scale agile development. We also needed processes that would remove bottlenecks and help scale testing. We thought long and hard about what we needed and established that we wanted the following features from our test management software:

- The ability to assign tests to a user because it's important to know who is going to work on what and what work is on everyone's plate.
- First class support for automated and manual testing. Automation shouldn't cover everything!
- An extensible API that would allow Loblaw Digital to use test data in other systems as well as integrate other systems with the test management software.
- Clear visibility of what was going on at all times. Particularly, visibility into the health of a given sprint at any given moment.

We then spent nearly two years reviewing test management software that would integrate with Jira and failed to find what we needed. Regarding integrated automation, feature set and extensibility, nothing the team evaluated could deliver what was required. We got to the point where we more or less gave up looking and toyed with the idea of producing our own tool and leaving it alone.

Yet only a few weeks later we came across a possible solution that would supply answers in all the areas that had been lacking in other products. So we decided to investigate. That product is Adaptavist Test Management for Jira (ATM) and, on first examination, it looked to be a completely fresh take on testing and it met our requirements. We subsequently implemented ATM over the next nine months and the results have been very positive.

Along the way, we made sure we got the features we'd identified as critical for our ideal test automation tool. And we learned a few things about what you need for a successful automated testing deployment.

Automated testing can't perform without the processes that support it
The Loblaw Digital team is seriously busy, generating an average of 20 concurrent builds throughout the day. End-to-end testing of those builds is on a massive scale since the supported browsers, languages, devices, and user agents multiplied by our available brands results in 384 permutations of any given test. As you might imagine, firing test results back to a test management solution in real time is no small feat.

In addition, we need results to be automatically organized and pushed to each seam and sprint segment without the interface getting cluttered or messy. We needed a tool that would meet all of these needs and one which would support our test creation, management and results processing at any timeframe or scale required.

Loblaw Digital is now running around 250,000 tests a day, which is critical to increasing confidence and speeding up the delivery of new features to production. Speed matters – you need to be able to make regression period – the pause where code is certified for production – we are now 2.5 times quicker but we believe we will soon make it five times quicker.

The tool should help you with planning
As noted above, at any one time the team is delivering a large number of features with the work on each split into what we call seams. The code produced in each seam must come together as a single product to go into production. We needed a tool to help us plan and track this, to be able to look at how a seam is tracking throughout a sprint, get a holistic view of its health, and plan accordingly. It's kind of unreal to get that granularity while also being able to obtain a bird's eye view.

Your tool should act as an orchestrator – what you want to avoid is investing in a dedicated resource looking across seams if you are to be successful. We now have processes which fundamentally change how we create, run and look at tests which allows us to focus on more important things.

Visibility of sprint health is crucial
We knew we wanted a tool that would give us visibility of sprint performance as well as traceability and increased coverage. We wanted to be able to quickly identify a problem and say: "We don't have tests for three tickets in this sprint. Check the tool and get someone on it." What you should have is the ability, at any time, to see whether sprint tests have been executed and how the results are trending. As we start to gear up for release, we can now generate a confidence score and make hard calls if need be.

You need powerful, usable reporting
When you have a quarter of a million results to sift through it's very difficult to understand what is happening using traditional reporting techniques. We needed a powerful reporting tool which can, for example, render results on an X/Y plane to find patterns of execution. Imagine test cases are your Y axis, and permutations are your X – you can easily identify if a given test is failing over many environments, or if a given environment is failing over many tests.

The value of a solution that shares your work with the whole company
Look at whether test reports and statuses can easily be shared with anyone who has access to Jira, or whatever planning and collaborative tool you use, so that it positively impacts on your entire company. Developers, product managers and other stakeholders across the business benefit from getting visibility of results. Make sure it also informs everyone about the scale and impact of what test engineering does, giving them more confidence in your work its value.

A way to make testing enjoyable
Finally, the right tool should make my team and the teams we interact with happier. It's probably not a word that often gets associated with testing but, quite honestly, testing is now less of a slog – and that increases morale. Getting the tools and processes right actually makes creating and running tests a pleasure and nothing we've used in the past has done that. When people see a sea of green in a report, it makes them feel good. In the long term I think this will have important benefits for staff retention and the continuity of what the team can deliver.

Honestly, the best test management tool that I can imagine is the one that people don't realize they're using. We waited a long time to find the right tool but the big lesson is knowing what you want and holding out against all the not-quite-right options until you find the perfect fit ■

Justin Watts is senior manager, test engineering at Loblaw Digital.

The testers of tomorrow (today)

by Johan Steyn

In an excerpt from his new book, Johan Steyn sets out what testers will need to be 'future proof'

The tester of tomorrow is a real leader. Where many in her trade like to work in the shadows, she operates in the trenches with her team.

There is a momentous shift taking place in the world of digital technology. Industries and careers that offered sanctuary to many professionals for many decades are disrupted in ways that we may never be able to grasp. Although the news media and industry forums have been shouting this news into our ears for a long time, many of us are oblivious to the dramatic impact of and speed at which we are approaching the cliff of innovation.

We are entering a new technological world, a world where only the brave will survive. Who are those brave souls?

They have the foresight to understand the massive impact of what is already happening to our world, and have taken the needed steps to survive the coming tsunami. Tsunami is the right word to use here. When a tsunami approaches, we cannot do much to stop the destruction about to hit our homes. But we can heed the warnings from scientists and prepare accordingly. A tsunami moves with great speed and is usually unexpected. As meteorological technology advances, we will have more time to organize when the warning bell sounds. But we will never have enough time. A tsunami wave moves faster than we can imagine.

The DevOps tsunami
Tsunami is the word I have been using for a long time to describe the changes in our digital world and technical careers. Some months back, I published an article on LinkedIn called The DevOps Tsunami which caused quite a stir among my peers. Resultantly, many software quality professionals from a global spectrum contacted me to express their views.

My sincere belief was that my description of the tsunami would echo what many others in our industry already knew and experienced. But I was surprised by the amount of resistance and criticism that filled by Inbox. Many who made contact expressed a belief that DevOps and the resultant impact on software quality management were just a fad – another buzz word like agile or scrum – and that it would soon disappear like the sound of a jet plane passing by. They expressed a “been there – done that” view: they have seen the many changes hitting our technological world but have experienced little change in their daily lives as testing practitioners. There are always new tools at our disposal, new buzz words and new trends. But many are still conducting software testing in a manual way, and they seem to be quite happy with that.

The status quo
This comfort zone of the status quo was built on personality cults and empires that were carefully manufactured in our corporate environments over the years. These cult leaders may have been good testing professionals in their hay-day. But over time, have they climbed the corporate ladder, nestled in a comfortable career where change and innovation were the enemy, and where like-minded minions filled the ranks of the teams they managed.

They have managed to become the go-to software guys in their corporate divisions and are the holders of the keys to quality. But to justify their existence, they keep their stakeholders – especially those with the funding on which their kingdoms depend – at ransom. Concepts like automotive innovation, cognitive technology and even the expertise of vendor partners are avoided at all costs. Innovation, the reuse of assets and the employment of disruptive thinkers are not welcomed. These things will cause their houses built on sand to crumble.

What does the tester of tomorrow look like?
The clarion call goes out to the software quality and testing community. What we desperately need TODAY is an army of the “testers of tomorrow”. The call goes out to those testing professionals who embrace the coming tsunami with all the change and uncertainty it brings. Nothing would have prepared you for this.

First of all, it is a testing professional with good technical skills. This is not someone who is bound to a specific tool, framework or methodology. This adaptable tester allowed himself to be exposed to a variety of the tools of his trade. Exploration, hunger for growth and innovation is the name of his game. The tester of tomorrow is a real leader. Where many in her trade like to work in the shadows, she operates in the trenches with her team. She drives by her example of commitment and dedication and she sees the strengths in her team not as threats, but as those essential elements that will make her successful, too. She is always keen to promote others and to give praise where it is due.

The tester of tomorrow is a commercially savvy leader. He understands that software quality management and testing is a means to an end. He always and foremost takes into account the business objectives of his customers and stakeholders. He spends time and effort with his team to ensure all are aligned with the business goals of their organization, and aligns their testing approach and planning to these. He is measured and measures his team on the successful realization of business goals through software quality management.

The tester of tomorrow is a shrewd political navigator. She knows that both her and her team's success rely on her political capital within her organization. She makes sure that she is connected to the relevant influencers and that she has their ear. She knows that gossip and second-hand information within the corridors of the workplace can scuttle her success. She knows how to promote herself with skilled manoeuvring, and she always ensures that the achievements of her team and the credit due to them is visible to her stakeholders. She recovers from failures gracefully, knowing how to dust herself off and tackle the failure with ownership to exceed expectations.

The tester of tomorrow is a reader and a learner. Learning never stops for this leader. He is on the cutting-edge with technological advances and innovation because he attends conferences, participates in webinars and spends time reading. He is not a lazy information gatherer. He is also well connected with his peers in the world of software quality. He is a voice worth listening to, a thought-leader.

The tester of tomorrow lives and breathes software quality management. She is not merely a tester at the end of the cycle. She is not seen as the "stepchild of the SDLC". Her voice and influence are heard from the very outset of a new project or feature being planned. Her peers welcome her opinion and shape their planning around her guidance. She embodies "shift-left" as she skilfully practises her craft throughout the software development and release process.

The impossible dream?

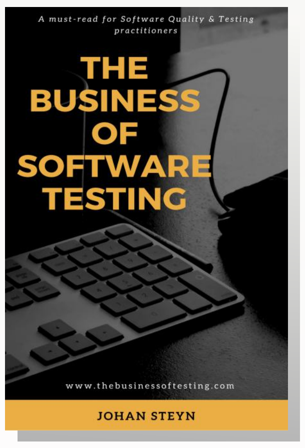
What I have just described may seem like a far cry from the reality that most quality professionals experience. The growth of a plant in a pot is restricted by its environment. Most organizations – whether end-users of software services such as banks, or even the supposed experts like global vendors – are not aware of or prepared for the tsunami. Your career ambitions as a tester of tomorrow may not be realized where you currently work. Many organizations still see software testing as a necessary evil to be avoided at all costs, or at least as a grudge purchase like short-term insurance.

Traditionally, our peers in the software world looked at testers as second-hand citizens. Testing was seen for those who did not "make the cut" to become developers. One would never be able to entice a hard-core developer into a career of software testing. The tsunami will force a change here. As we wake up to the tsunami-hit world around us, and as the actual role of software quality is recognized in a world moving at a fast pace that introduces massive risk, the tester of tomorrow will find her real place.

I see a world where those hardcore, weirdo pony-tail developers can be enticed to focus on a career in software quality management. In this world, their technical and development skills will make them the ideal candidates to test software.

Dear reader, welcome to a brave new world! Will we find you sinking or swimming as the tsunami hits? ■

Johan Steyn is senior manager of enterprise quality assurance at Nedbank and author of 'The business of software testing'. For more info visit Johan's web site www.thebusinessoftesting.com
The opinions expressed here are the author's own. It does not necessarily reflect the views of any of the organizations he is currently or has previously been involved with.


A must-read for Software Quality & Testing practitioners
THE BUSINESS OF SOFTWARE TESTING
www.thebusinessoftesting.com
JOHAN STEYN

Spot the difference: automating visual regression testing

by Viv Richards

Visual testing adds value to tests by automating some of the checks normally carried out by manually testing

Using a visual approach can reduce the amount of lines of code required for an automated test dramatically

This feature explores how visual testing can add another tool to your belt as well as highlighting issues which teams may face when implementing such a framework.

Most now accept that test automation has specific advantages for improving long-term efficiencies but when it comes to implementation, there are many hurdles and it can stop progress in its tracks.

Whilst our visual framework is still in development, and we are still facing many challenges due to the broad range of products we offer and the way in which we develop and test them, I'd like to share our experiences to date and the ways in which we've been able to work around them to start building a solid and valuable visual test framework.

How valuable are the tests?
Figure 1 shows a basic example of a type of test I've seen written. The tester set out with good intentions and invested time to ensure that should the form change, they would be able to capture the change using their test.

Late one Friday evening the support team got a call, the customer could no longer use the application as the send button was not on the screen. The support person immediately checked the tests to check for any failures but was confused when they noticed that all the tests had passed… what had happened?

In Figure 2, you can see that the issue was that the send button had rendered on the form so the asserts all passed, a css change had been made and so the send button had become hidden behind one of the input text areas.

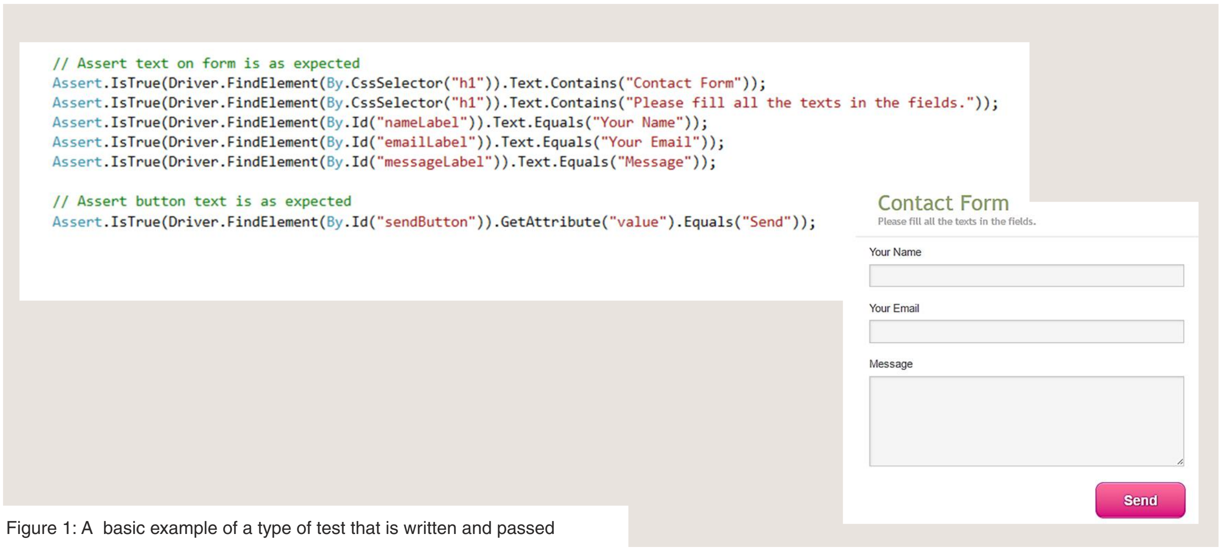
How good are you at spot the difference?
Before we go much further, how good do you think you are at finding differences? Below there are a number of differences between the two images. Once you think you've found them all continue reading. See Figure 3.

How many differences did you find?
They're flagged in Figure 4 but it is often difficult to know where they are and to know when to stop checking, and to have confidence to know we've found them all.

Can we add more value to our tests?
Given the spot the difference example, we asked ourselves if there is a way in which we can add more value to our tests by looking at ways to automate some of the checks we'd normally carry out by manually testing?

What if you could simply assert your base image of an element or web page matches a snapshot of an element or web page based on how it currently looks?

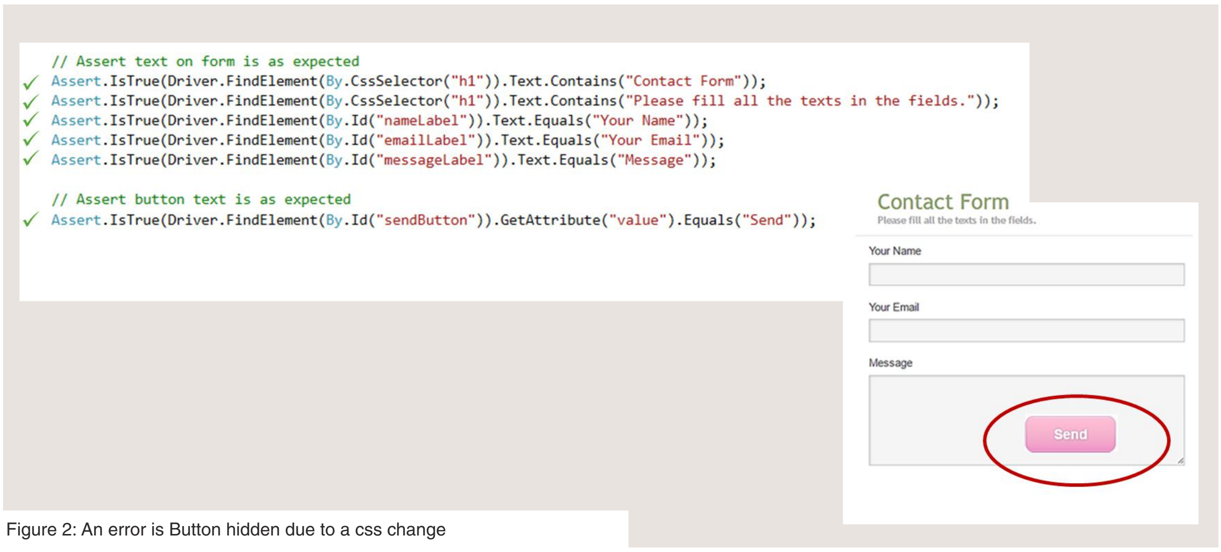
Visual testing
When looking at visual testing, there are many options out there, again similar to test automation tools it depends on where, what and how you need to test. Do you want it free,



```
// Assert text on form is as expected
Assert.IsTrue(Driver.FindElement(By.CssSelector("h1")).Text.Contains("Contact Form"));
Assert.IsTrue(Driver.FindElement(By.CssSelector("h1")).Text.Contains("Please fill all the texts in the fields."));
Assert.IsTrue(Driver.FindElement(By.Id("nameLabel")).Text.Equals("Your Name"));
Assert.IsTrue(Driver.FindElement(By.Id("emailLabel")).Text.Equals("Your Email"));
Assert.IsTrue(Driver.FindElement(By.Id("messageLabel")).Text.Equals("Message"));

// Assert button text is as expected
Assert.IsTrue(Driver.FindElement(By.Id("sendButton")).GetAttribute("value").Equals("Send"));
```

Figure 1: A basic example of a type of test that is written and passed



```
  // Assert text on form is as expected
✓ Assert.IsTrue(Driver.FindElement(By.CssSelector("h1")).Text.Contains("Contact Form"));
✓ Assert.IsTrue(Driver.FindElement(By.CssSelector("h1")).Text.Contains("Please fill all the texts in the fields."));
✓ Assert.IsTrue(Driver.FindElement(By.Id("nameLabel")).Text.Equals("Your Name"));
✓ Assert.IsTrue(Driver.FindElement(By.Id("emailLabel")).Text.Equals("Your Email"));
✓ Assert.IsTrue(Driver.FindElement(By.Id("messageLabel")).Text.Equals("Message"));

  // Assert button text is as expected
✓ Assert.IsTrue(Driver.FindElement(By.Id("sendButton")).GetAttribute("value").Equals("Send"));
```

Figure 2: An error is Button hidden due to a css change

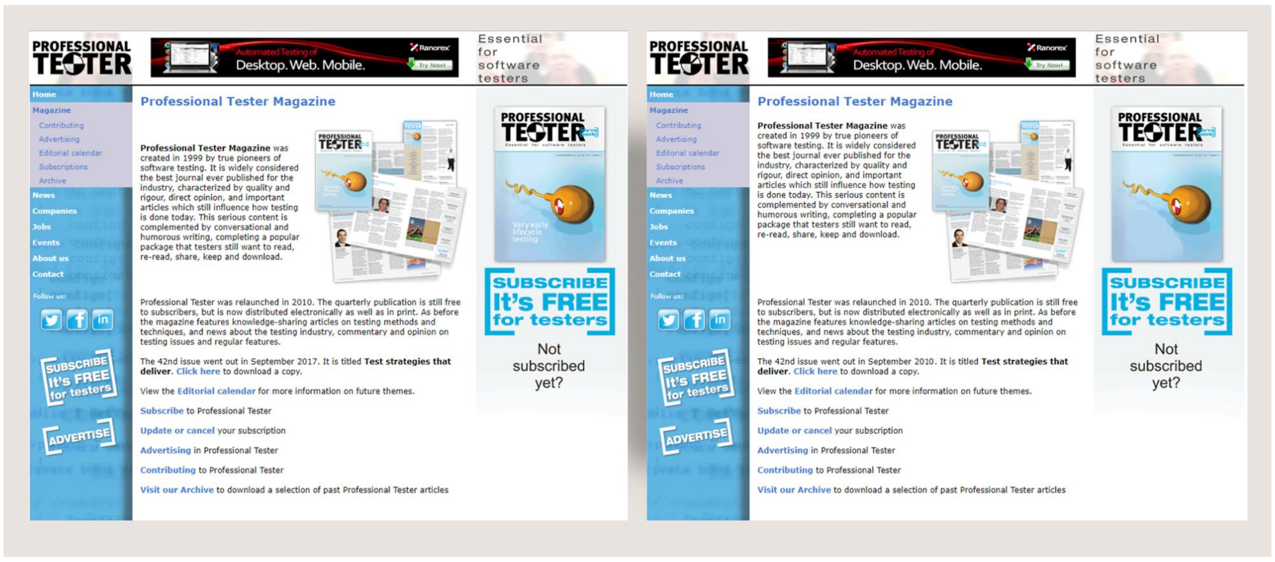
can you afford to pay? Do you need a GUI or are you comfortable to write some code? How do you want to be alerted of differences when they are found?

When looking at visual testing at my current employer, we were unable to simply use another offering. Many options didn't work straight out of the box, or contained lots of bloat which would just add an extra overhead to the maintainability of the tests. There had been a massive investment in our automation frameworks and lots of investment which had been made in upskilling developers/testers in C# and Selenium and possibly looking at new languages or frameworks could mean a steep learning curve. The main issue for us was that we needed it to fit in with our current automation framework, and so we had to create our own visual testing framework.

Reliability
Whilst developing our framework, it quickly became apparent that the rendering from different browsers would often cause tests to fail. For example, if we were to take a base image of our homepage in a Chrome browser and then run our test in Firefox to ensure the homepage hadn't changed, the test would fail due to the browser adding additional padding to some elements. The pink boxes in Figure 5 indicate areas where differences were found by our framework.

Execution speed
When comparing a visual test of the contact form compared to the test in Figure 6, where we asserted each element's text, we noticed the visual tests executed far quicker. The test framework simply navigates in a web driver to the desired page, takes a snapshot which is held in memory and then byte by byte compares the image to the base image stored locally or wherever you'd prefer.

Maintainability
Using a visual approach can reduce the amount of lines of code required for an automated test dramatically, as shown in Figure 6. As an example, a previous test I'd been asked to write was 500+ lines, when writing this using the visual



PROFESSIONAL TESTER

Automated Testing of Desktop. Web. Mobile.

Essential for software testers

Home
Magazine
Contributing
Advertising
Editorial calendar
Subscriptions
Archive
News
Companies
Jobs
Events
About us
Contact
Follow us:

SUBSCRIBE It's FREE for testers

ADVERTISE

Professional Tester Magazine

Professional Tester Magazine was created in 1999 by true pioneers of software testing. It is widely considered the best journal ever published for the industry, characterized by quality and rigour, direct opinion, and important articles which still influence how testing is done today. This serious content is complemented by conversational and humorous writing, completing a popular package that testers still want to read, re-read, share, keep and download.

Professional Tester was relaunched in 2010. The quarterly publication is still free to subscribers, but is now distributed electronically as well as in print. As before the magazine features knowledge-sharing articles on testing methods and techniques, and news about the testing industry, commentary and opinion on testing issues and regular features.

The 42nd issue went out in September 2017. It is titled **Test strategies that deliver**. Click here to download a copy.

View the Editorial calendar for more information on future themes.

Subscribe to Professional Tester

Update or cancel your subscription

Advertising in Professional Tester

Contributing to Professional Tester

Visit our Archive to download a selection of past Professional Tester articles

SUBSCRIBE It's FREE for testers

Not subscribed yet?

PROFESSIONAL TESTER

Automated Testing of Desktop. Web. Mobile.

Essential for software testers

Professional Tester Magazine

Professional Tester Magazine was created in 1999 by true pioneers of software testing. It is widely considered the best journal ever published for the industry, characterized by quality and rigour, direct opinion, and important articles which still influence how testing is done today. This serious content is complemented by conversational and humorous writing, completing a popular package that testers still want to read, re-read, share, keep and download.

Professional Tester was relaunched in 2010. The quarterly publication is still free to subscribers, but is now distributed electronically as well as in print. As before the magazine features knowledge-sharing articles on testing methods and techniques, and news about the testing industry, commentary and opinion on testing issues and regular features.

The 42nd issue went out in September 2010. It is titled **Test strategies that deliver**. Click here to download a copy.

View the Editorial calendar for more information on future themes.

Subscribe to Professional Tester

Update or cancel your subscription

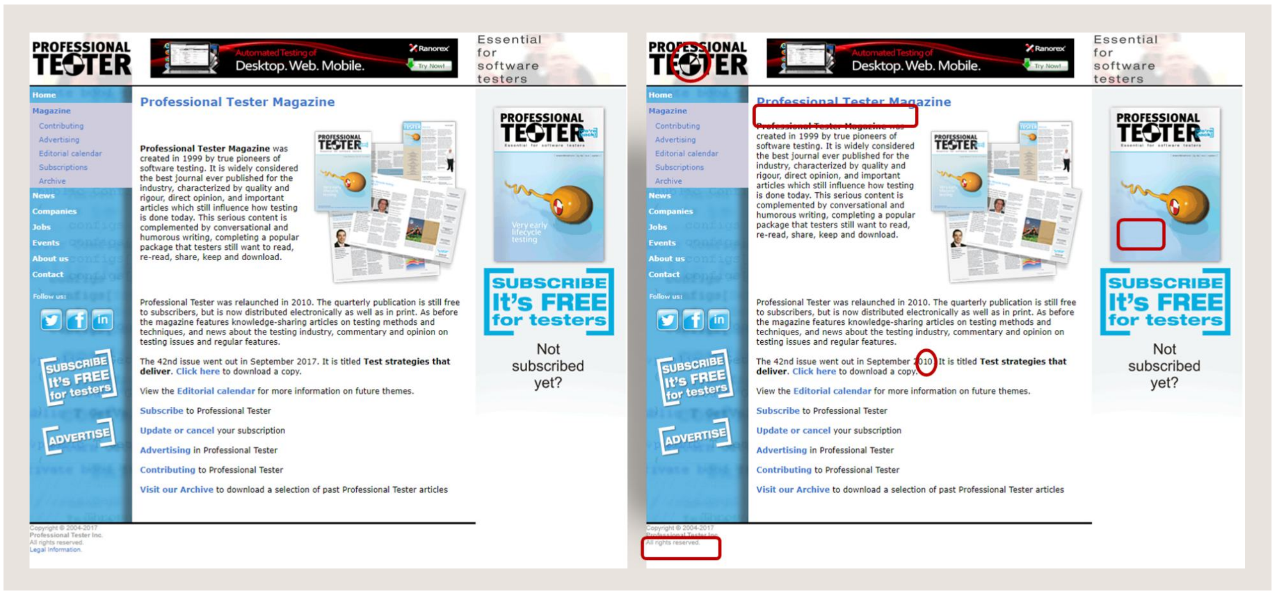
Advertising in Professional Tester

Contributing to Professional Tester

Visit our Archive to download a selection of past Professional Tester articles

Not subscribed yet?

Figure 3: How good are you at spot the difference?



PROFESSIONAL TESTER

Professional Tester Magazine

Professional Tester Magazine was created in 1999 by true pioneers of software testing. It is widely considered the best journal ever published for the industry, characterized by quality and rigour, direct opinion, and important articles which still influence how testing is done today. This serious content is complemented by conversational and humorous writing, completing a popular package that testers still want to read, re-read, share, keep and download.

Professional Tester was relaunched in 2010. The quarterly publication is still free to subscribers, but is now distributed electronically as well as in print. As before the magazine features knowledge-sharing articles on testing methods and techniques, and news about the testing industry, commentary and opinion on testing issues and regular features.

The 42nd issue went out in September 2017. It is titled **Test strategies that deliver**. Click here to download a copy.

View the Editorial calendar for more information on future themes.

Subscribe to Professional Tester

Update or cancel your subscription

Advertising in Professional Tester

Contributing to Professional Tester

Visit our Archive to download a selection of past Professional Tester articles

Copyright © 2004-2017
Professional Tester Inc.
All rights reserved.
Legal information

The 42nd issue went out in September 2010. It is titled **Test strategies that deliver**. Click here to download a copy.

Copyright © 2004-2017
Professional Tester Inc.
All rights reserved.

Figure 4: Errors can be easily missed

framework this was just four lines of code; one line to specify the base image, one line to specify the URL to compare, one line to do the compare and an assert, and that's it!

Feedback

One of the fantastic things with the visual testing framework, and specifically the one we designed, was that whenever a test would fail, a copy of the original image is created and when differences are found, a pink box is drawn to quickly identify the areas of change.

Accuracy?

During development of the testing framework, and whilst running spikes using various other visual testing tools, they were all pixel perfect. The frameworks were able to detect a single pixel difference, as seen in Figure 7. However, after a few months of testing and asking other team members to help out, the tests started to fail. We found that within our visual framework, we had to start allowing a certain amount of tolerance during comparison. A common issue we encountered was that the colour would sometimes be slightly off (a slightly different shade) depending on the machine taking the base image, or taking the comparison image. Whilst this wouldn't be a problem on the machine which would always run the tests, it would cause issues when developers would run the tests locally. So, we had to introduce a tolerance for the 256 different intensities. Whilst this no longer was a single pixel perfect we found it accurate enough to still enable us to assert that layouts were as expected, as well as checking that wording was correct and elements were being rendered as expected.

Base images

When initially creating the framework, we decided it would be a good idea to manually take base images and should we need to test a specific browser, we would take a base image

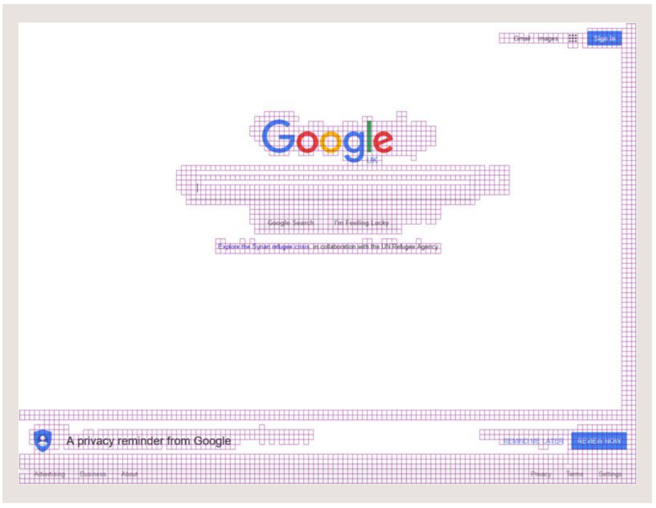


Figure 5: The pink boxes flag areas where differences were found by our framework

of it i.e. the homepage in each required browser. This, however, became quite difficult to manage and so we changed our approach. Instead, we had the framework automatically check which web driver we were using, check if a base image already existed for the page and browser, and if not to create it automatically and tell us that it had been done. Now when we run a test in any browser and the base image doesn't exist, it creates it for us. We have a helper if we want to override the base image, or we can simply just delete it and when we run the test, the base image will be created automatically for us. This reduces the need for developers to manage the base images.

Screen resolutions

During early stages of the visual testing framework, I was asked to demonstrate the framework to the team, but all did not go to plan. I'd created the base images on my local machine but then for the demo had a machine with a smaller resolution. When I ran the tests they all failed as the browser had rendered slightly differently to what was expected because of the resolution. A work around for this was to set a height and width for the web driver and not to set it maximized. In this way, whenever the web browser was open to take images it would always be displayed in the expected size rather than depend on the screen resolution of the machine the test was being run on. See Figure 8.

Storage

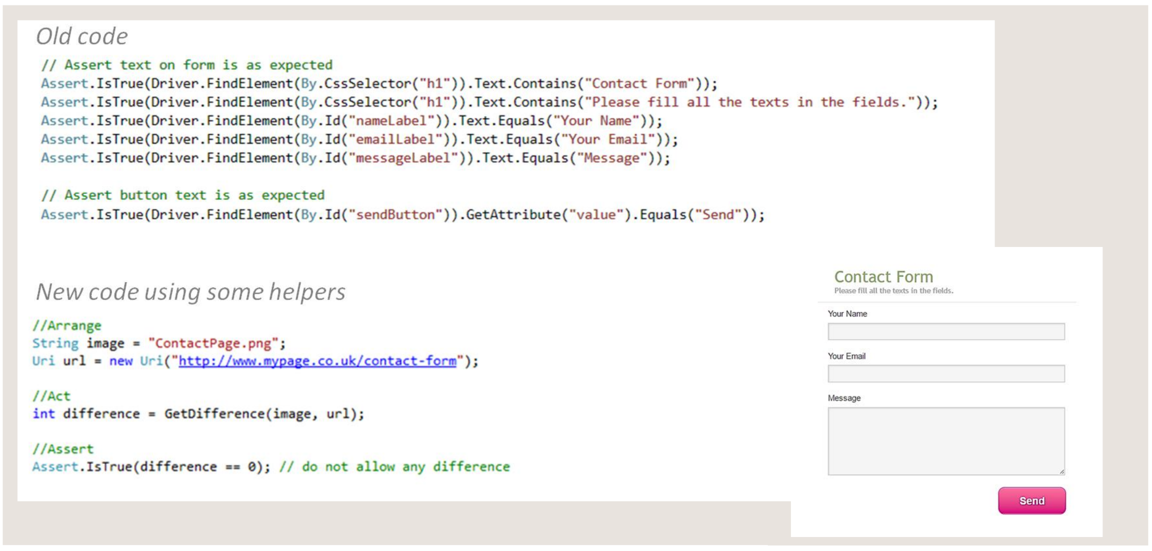
When running the tests, depending on the screen resolutions you set, the number of pages you are testing and the number of browsers you are testing in, the base images can quickly start to fill up the disk on the computer running the tests. Whilst we are currently only running the tests locally, and saving the images on a local disk, a database or online storage longer-term would be a preferred option.

Dynamic content

One of the big challenges for us has been dynamic content, often our web pages display a logged-on client name or perhaps a news feed. One of the ways we've been able to get around the issue is to create a helper which simply blankets over the dynamic elements, as shown in Figure 9.

Visually checking documents?

Another quite interesting challenge we've had, is to investigate the ability to visually check documents. We deal with a large number of items and it can be very difficult, under tight timescales, to visually check formatting, spelling, layouts, colours on a multi pages document. You can use a third party



Old code

```
// Assert text on form is as expected
Assert.IsTrue(Driver.FindElement(By.CssSelector("h1")).Text.Contains("Contact Form"));
Assert.IsTrue(Driver.FindElement(By.CssSelector("h1")).Text.Contains("Please fill all the texts in the fields."));
Assert.IsTrue(Driver.FindElement(By.Id("nameLabel")).Text.Equals("Your Name"));
Assert.IsTrue(Driver.FindElement(By.Id("emailLabel")).Text.Equals("Your Email"));
Assert.IsTrue(Driver.FindElement(By.Id("messageLabel")).Text.Equals("Message"));

// Assert button text is as expected
Assert.IsTrue(Driver.FindElement(By.Id("sendButton")).GetAttribute("value").Equals("Send"));
```

New code using some helpers

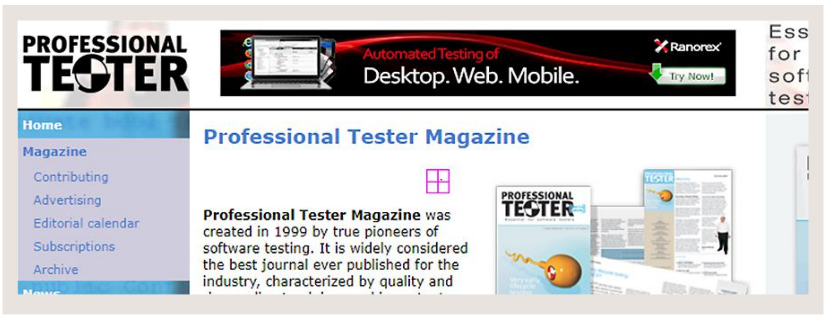
```
//Arrange
String image = "ContactPage.png";
Uri url = new Uri("http://www.mypage.co.uk/contact-form");

//Act
int difference = GetDifference(image, url);

//Assert
Assert.IsTrue(difference == 0); // do not allow any difference
```

Contact Form
Please fill all the texts in the fields.
Your Name
Your Email
Message
Send

Figure 6: Visual testing can dramatically reduce the amount of lines of code required for an automated test



PROFESSIONAL TESTER

Automated Testing of Desktop. Web. Mobile. Ranorex Try Now!

Home
Magazine
Contributing
Advertising
Editorial calendar
Subscriptions
Archive

Professional Tester Magazine

Professional Tester Magazine was created in 1999 by true pioneers of software testing. It is widely considered the best journal ever published for the industry, characterized by quality and

Figure 7: Even a single pixel difference can be identified

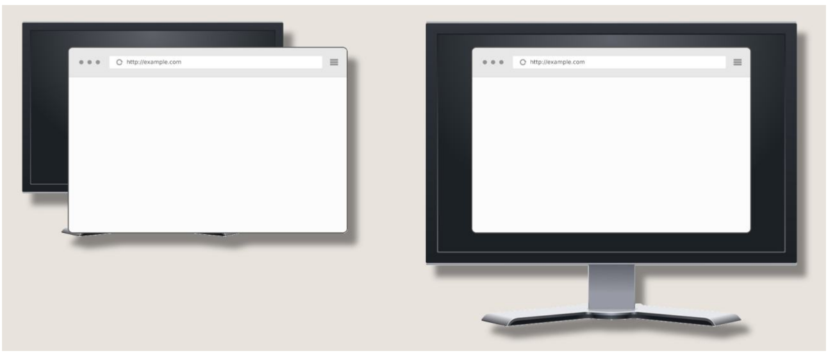
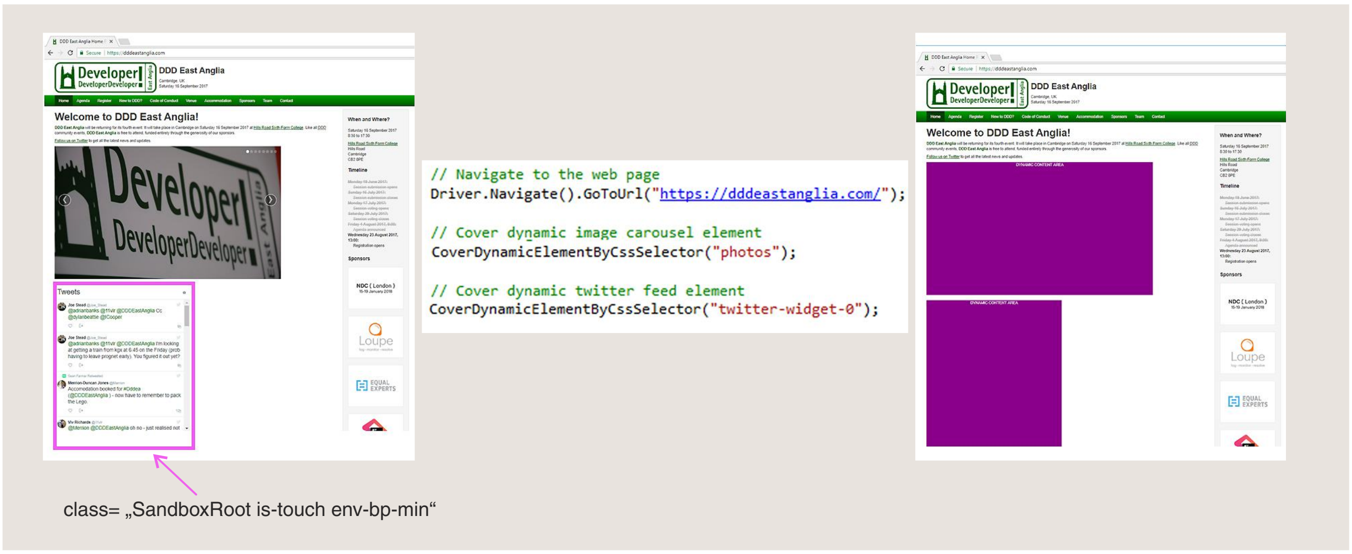


Figure 8: Setting a height and width for the web driver ensures images will be displayed in the expected size



```
// Navigate to the web page
Driver.Navigate().GoToUrl("https://dddeastanglia.com/");

// Cover dynamic image carousel element
CoverDynamicElementByCssSelector("photos");

// Cover dynamic twitter feed element
CoverDynamicElementByCssSelector("twitter-widget-0");
```

DDD East Anglia
Welcome to DDD East Anglia!
Tweets
class= „SandboxRoot is-touch env-bp-min“
DYNAMIC CONTENT AREA
DYNAMIC CONTENT AREA

Figure 9: Dynamic content can be blanketed

such as Aspose to convert a PDF page or pages to images and then run them through the visual testing framework to quickly check for differences.

An exact copy?

Even with the ability to cover dynamic content, it's not always desirable to cover all elements of a page. What if you only want to check a small portion of your page, perhaps just a button to ensure it still matches the customers set branding?

By using visual testing, you can not only test that a whole page matches what you expect it to but it is possible to check an individual element. By specifying an element by CssSelector or ID you can take a base image of that individual element and then run tests to check for changes.

Want to find out more?

Whilst there are many fantastic free and paid for visual tools and frameworks out there, you can pick up a free copy of the framework we are developing over on GitHub and let us know what you think. https://github.com/vivrichards600/AutomatedVisualTesting ■

Viv Richards is a test engineer at Vizolution, a blogger and a community bumble bee. He is a CodeClub volunteer, organizes South Wales' largest agile and developer conference (SwanseaCon) and is co-organizing DDD Wales.

Happy
NewYear
With thousands of subscribers, make sure that your solution is showcased in 2018.
PROFESSIONAL TESTER
Essential for software testers