What's the point of Test Automation?

Introduction
In this paper we cover the high level fundamentals of test automation, discuss the primary problems test automation solves and take a look at the steps required to implement this within your development process – all aimed at improving software delivery.

Before we start...
Discussions around the area of test automation are often overly complicated and rife with misunderstandings. Terms like BDD, TDD, agile testing, acceptance testing and functional testing are flung around together, and they regularly merge to become one concept in the minds of many. Understanding test automation becomes simpler however once you understand where it fits as a discrete idea within the world of agile development. We also learn that getting the most out of test automation requires implementing changes to the overall delivery process.

Let's look at some of the basics of the agile approach, and how it relates to test automation and then review the benefits and implications of implementing an automated in place of manual test regime:

Iterative delivery
This is a core concept of agile development. This concept moves away from massive specification documents and long development phases, and introduces a more streamlined approach that is reminiscent of a production line. Requirements are pushed through in small groups, developed by the team and released at regular intervals. With this in place, testers are able to become involved much earlier in the process. They can test the output of each individual requirement, rather than being forced to wait until the entire delivery process is complete.

Continuous integration
This is a process that constantly takes the output of the development team and performs build and deployment jobs automatically. This means that as soon as a piece of code is considered done by the team and pushed into the codebase, that change is made available to test. Adding test automation to this process provides immediate test feedback.

Test Driven Development (TDD) and Behaviour Driven Development (BDD)
TDD and BDD are enhancements to an automated test process. Within this paper we will be discussing automation within the context of an agile delivery team, with a focus on areas generally referred to as acceptance or functional testing. For simplicity, we will disregard BDD and TDD as concepts that can be introduced separately.

Gradual improvements
Gradual improvements can be made to any existing process without the need for a “big bang” approach to change. When teams struggle to adopt an agile development process, they generally do so because they attempt complete change to what they consider “proper” agile. Change should be managed, and new processes and tools only introduced when the benefit is properly understood and measurable.

Cost and speed
The financial and process benefits of test automation will appear in the enhanced code quality and speed of software delivery as a whole. The creation of a more rapid, iterative development process will greatly reduce the testing team in terms of headcount, with a similar combined upfront resource cost due to the different set of skills required.

The problem with manual testing...
Before making any change, it is important to consider the problems you are trying to address. Common challenges faced by teams that rely heavily on manual testing:

Test feedback
This is often overlooked as a problem during a traditional manual test based approach. Any time gap between development and testing can introduce exponentially larger delays to the delivery process. If a defect is returned to the development team after some time has passed, additional problems will arise. For example, the person who wrote the code may have either left the team, or simply forgotten the specifics of the work they did.

Fix rejection
Fix rejection or retest failure is a further cause of delay in delivery; a developer returns a supposedly corrected defect to the test team for review, only for it to be subsequently rejected for not addressing the problem. Without an automated verification (a failing test which now passes), the likelihood of this occurring repeatedly is much higher.
Continuous Integration Pipeline

Regression testing
This verifies that existing functional capability has not been negatively affected by new features. This critical testing phase is usually the first to suffer when deadlines loom.

As defects are raised and addressed by the development team, true regression testing requires that the entire regression pack is run again. Given the time this would take, retesting is not normally performed outside the area directly impacted by the fix.

Suitability
The suitability of the tests is not guaranteed by any process, but traditional manual testing falls particularly short of this goal. Most organisations will either produce a test plan based on the initial design documents used as an input to development, or produce them after the fact. These tests will be out of touch with changes discussed and implemented during development, leading to more wasted time when defects are passed back to developers that turn out to be related to expected behaviour. This is assuming the test approach is designed to capture a failure to meet requirements, in addition to obvious defects (broken user journeys, styling issues etc.), but this is not always the case.

Choosing the right tools
Whilst this is an important step, having a well structured test framework in place makes the choice of tools is less critical than might be assumed. Tools should be selected based on the nature of the system under test - taking into account the primary development language being used as well as the interface for the system. The framework should readily support the swapping and replacement of specific tools as desired and without impacting the related tests.

Structuring the test framework
Automated tests are easy to maintain and extend within a well structured test framework. The main goal of a well-structured framework is to ensure that test scenarios are kept clearly separated from the interaction code. Interaction code should be held to high quality expectations and be structured to avoid code duplication. A good example of this in action is the page objects pattern often used when testing web application user interfaces (see figure 1)

The figure 1 diagram shows a framework that ensures test scenarios can reuse common code, meaning that the effort to update the automation framework is in line with changes to the system under test. In other words, simple changes to the application require only small changes to the test framework.

The path to test automation…
The exact steps taken will depend upon the starting point and exact nature of your business. However, a high level approach to implementing a successful automation strategy can still be discussed.
Upskilling on the move
You can up-skill whilst developing software by having the developers and testers working together within the test framework. When moving from manual testing to a more automated approach, testers will need to be introduced to writing code gradually.

Using the framework in figure 1 as an example, developers will produce page objects which expose simple methods for interacting with the UI. Automating the test scenarios would then be a case of utilising these methods to interact with the system under test. Over time the testers will be able to extend and modify the interaction code and eventually take ownership. Depending on the starting point in the organisation, this could either be a simple training exercise or require recruitment.

Co-locating
The co-location of teams encourages developers and testers to work harmoniously together. Initially this simply requires them to be in the same geographical area. The eventual goal is to have a single unified development team; developers and testers working together to deliver requirements. Without this unity, testers lack the required support and level of communication to produce automated tests in line with iterative development.

Shared ownership of quality
This is another fairly fundamental idea of agile development with functional and performance testing no longer falling within the remit of a dedicated testing team. As teams merge into a single development team, developers will take on greater responsibility for code quality with their creation of unit and integration tests as they deliver code.

At this point the skills exchange becomes two way as testers will be able to share their experience to construct appropriate test cases.

Appropriate levels of testing
This ensures that tests are efficient and maintainable.

The largest set of tests should be at the unit level, verifying that any given piece of code generates the correct output from all expected inputs.

Integration tests should be a smaller set of test that focus on key examples of communication between different areas of the code base, or between applications in more distributed systems.

UI tests (or functional tests) should be the smallest set as they will be the slowest running and most brittle. High-level functional tests represent the first time that the application will be fully tested from end to end, from the interaction point (e.g. the UI) down to the underlying data structures. Functional tests will also catch any failure to meet business requirements, rather than purely technical flaws.

Changing the way your organisation assures quality is bound to come with challenges. Some of them can be addressed simply by planning the adoption and taking the right first steps, but some will require work throughout the process of change.
Barriers to culture change...
Changing the way an organisation operates in order to assure quality will come with challenges. These can be addressed by planning the adoption carefully and taking the right first steps. Some challenges will require work throughout the entire process of change.

Skill requirements
This is the most obvious barrier. It is almost inevitable that finding people with the ideal balance of testing and development skills will be a challenge. Taking appropriate steps in advance as mentioned in previous sections will mitigate but not remove this problem.

Testing iteratively delivered software
This requires a mental shift for testers from a manual background over and above the increased skill requirements. Test personnel need to limit their testing to the scope of the requirement rather than testing the system as a whole. Doing so while effectively testing integration with dependant functionality takes time to grasp.

Company culture
The organisation’s culture will repeatedly impact the success of test automation. This can be addressed in part by encouraging team ownership of software quality and breaking down the silos that exist within many organisations. This only helps within the context of the development team itself.

An appropriate culture must exist at all levels of the company from the CEO down. Business stakeholders must be willing to engage with the development process to ensure clarity of requirements and give fast feedback on desired capability when reviewing defects. Members of the development team must be empowered to support themselves without being entirely dependent on external support/operations teams.

Finally, automated tests must be regularly reviewed and maintained, and should never be considered “done”.

Going the next step...
Further enhancements to maximise the benefits of Test Automation include:

Behaviour Driven Development (BDD)
BDD is a widely used methodology that offers many benefits when correctly implemented. It is often incorrectly viewed as an integral part of automation, whereas BDD is actually a natural language layer that sits on top of test automation. It allows for requirements to be expressed as test cases which can be understood by non-technical stakeholders. This supports the idea of verifiable requirements, as well as producing a clear test report and documentation of system behaviour.

Performance testing
Performance testing is often overlooked in the development process. With a good continuous integration pipeline it is easy to include automated performance tests that monitors systems’ performance under every-day conditions. This is then extended to demonstrate the impact of both substantial load and how the system behaves in a range of failure scenarios.

Parallel test execution
This makes your test automation more valuable by further increasing the speed of feedback. A first-pass automated suite will often run test cases consecutively, which becomes problematic as the suite of test cases grows. Enabling parallel execution allows you to scale almost indefinitely to support any size of system. Considering this in advance leads to ensuring that test cases are able to run independently, without cross dependencies such as shared test data.

About the Author
Tristan McCarthy is test practice lead for OpenCredo with substantial experience of ensuring the delivery of high quality, usable software.

An advocate of cross-skilled agile teams, he is constantly seeking to extend the remit of a tester beyond the traditional role. With his interest in new testing tools and techniques he has built up expertise across a wide range of technologies. Particular focus is given to BDD tooling, web and desktop automation tools, and the creation of custom testing frameworks.

About OpenCredo
OpenCredo believes great consultancy is founded in deep and broad real-world experience. Our highly capable and experienced people provide applied knowledge of the latest technologies and best practices to all our blue-chip clients. These capabilities stretch across the entire spectrum of the development process from architecture through software engineering to DevOps.

Transparent, objective driven and pragmatic, OpenCredo acts as a trusted advisor to its clients. It is a company founded on excellence in software engineering, and our approach is to lead and support through deed and example - delivering tangible value, not slide decks or fluff.

OpenCredo believes great consultancy is founded in deep and broad real-world experience.