Test strategies that deliver

Including articles by:

Paul Mowat
Accenture

Clinton Sprauve
Sauce Labs

Gregory Solovey
Nokia

Janet Gregory

Dorothy Graham

Christoph Preschern
Ranorex

September 2017

# 42
All-in-One Test Automation

- Any Technology
- Seamless Integration
- Broad Acceptance
- Robust Automation
- Quick ROI

Now with Selenium Integration

- Access Selenium with the Ranorex automation framework
- Address Selenium pain points
- Web testing across all major platforms and browsers
- For testers and developers

www.ranorex.com/try-now
Test strategies that deliver

“For sure – the traditional way of software testing is changing.” That was the response of Christoph Preschern, MD and chief operating officer of Ranorex, one of the people we’ve spoken to about the challenges testers now face.

We wanted to look at test strategies that can successfully anchor project delivery. Clinton Sprauve tackles mobile test automation and Gregory Solovey sets out how model driven testing can achieve test completeness.

But we also asked commentators to give their insight into the thinking and organizational dynamics that testers have to contend with. With some 40 years of testing experience, Dorothy Graham has put together an indispensable guide to clearing up misconceptions around testing, automation, tools and people. Janet Gregory also provides her view on what must be in place for teams to transition and thrive in an agile world.

As always, we hope you enjoy the magazine, and do let us know your views.

Vanessa Howard
Editor
The formal process-driven TCoE must evolve or it will no longer have a role to play

by Paul Mowat

Is the TCoE redundant?

The fixed nature of the TCoE, which once delivered efficiencies and constancies, proved its downfall

I have worked as a tester, test architect lead and test manager and over many years, have established and led high-performing TCoEs. Success was built-in and repeatable and yet the very things that provided TCoEs with their triumphs also led to their downfall. Now I believe that only two routes are open to TCoEs and if they fail to adapt, their role is redundant.

Why they mattered

Back in the day, the traditional TCoE was where you went to look for the test capability, it was where the organization stored all the test processes and functions which supported test professionals. A clear advantage of setting up a TCoE was the standardization of test processes and tools and this dovetailed well with testers acting independently of the development team and project managers.

When I began life as a tester, our community was a mixture of dedicated testing experts, ranging from programme and project test managers, to technical testers who either knew the product or domain, or had a level of automation skills or experience of OAT/business readiness, as well as test analysts. Many of the test analysts found themselves in roles which required manual execution and there were certainly a lot of manual testers committed to projects. Clients would ask why the testing budget was so high compared with the development budget and they would question why testing was manual and not automated. Added to this environment were specialist disciplines which meant the TCoE could offer a truly end-to-end service e.g. non-functional testing performance engineering and security testers, along with domain expertise, which provided projects with an optimal mix of skills to deliver the projects.

TCoEs aligned the test goals to the business goals with a set of principles and guidelines, which generally aligned to the quality assurance and SDLC goals. Proven methodologies, deliverables and templates ensured structure and repeatable test process were applied at each test level, along with continuous improvements and metrics gathering, which allowed processes that were over-engineered to be eliminated. As well as removing waste, risk-based testing was implemented to ensure testability of the requirements from the BAs and the business, test optimization using orthogonal array analysis was used to reduce the number of test cases and ensure the right coverage was applied. These measures meant TCoEs continually focused on improvements and the elimination of waste.

The age of theory, not practise

If I cast my mind back to my university days, studying an undergraduate business and management course in Oxford, I remember learning about international business and quality management. Yet lean manufacturing principles had been around since the late 1940s, derived from the Toyota Production System. The 1970s saw the onset of various iterative/incremental lean models which became applied to software development. In the 2000s came Scrum and the crystal method, with the Agile Manifesto published in 2001, and in 2008 we saw the introduction of DevOps. My course covered topics including Deming and lean processes however it was not until eight years (2006) after I entered the world of work that I began to learn about agile.
Why have I raised this? Essentially because the value of driving efficiencies may have been around for many years but it was not classified and accepted at the enterprise level until very recently. It filtered down and gave impetus to TCoEs as there was a desire to set up and centralize capability and reduce costs. So, what could possibly undermine the role of a well-run TCoE?

The tortoise outrun by the hare

With attempts to bring operations and development (DevOps) closer and to dismantle business function walls, IT has changed and delivery is expected to get done and get done fast. Customer expectation levels can be characterized as ‘I want my order delivered within an hour and product quality should be high’. This disruption has resulted in many other changes emerging, for example the rising number of individuals who are moving to contracting and those who are entrepreneurial setting up and running multiple businesses. The pace of technology and the pace of adoption are probably faster than any other time in our history. And these factors contributed to the demise of the TCoE.

Early adoption of DevOps was happening in the early 2000s and you could argue that the likes of Amazon, with their 50 million deployments per year, were commissioning it even earlier. During this same period, around 2002 – 2014, a lot of organizations were building TCoEs mainly because of the reduction in test costs, moving away from cost centres to an established testing service. An organization now had the opportunity to select from a pool of suppliers/resources internationally. There was a big push towards increasing the automation of the regression suite to cope with multiple deployments into test and the test cycle time was reduced due to efficiencies.

However, the fixed nature of the TCoE, which once delivered efficiencies and constancies, proved its downfall. Process driven and with set team member functions/skills, it failed to keep up with the movement towards more frequent delivery.

I recently conducted a sample survey across several client organizations from CMT, FS & products industries where TCoEs were established and have become mature. Here are the reasons given as to why they initially setup a TCoE - note how the reasoning is vulnerable to becoming obsolete or irrelevant within a DevOps/agile environment.

- I wanted a managed service, I knew what I was paying for and I could assess the supplier’s performance against regularly SLAs
- Driving cost down is important thus a commercial model based on either a blended rate or an average day rate with minimum commitment works for me
- My function has a set of core capabilities which I can offer as a service and I can manage non-core specialist capabilities through the contract market or bolt on service
- If I document all the processes and conduct regular audits, it gives me a sense of security because I know the teams are following those processes so the output is of a high standard
- Having access to a pool of career testers who are regularly trained, maintain a view of progress in industry and are motivated helps with the quality and utilization levels

The final victory of the hare?

DevOps moved into the enterprise and is now considered mainstream. The traditional model focusing on validation and verification using static testing is process/test script driven and is much slower when it comes to execution. Manual testing is now a bottleneck; how can a group of manual testers execute a suite of manual tests continually throughout the day to keep up with the speed at which the development team checks code into the production branch?

Also, we have seen quality actively embedded within agile teams and it’s no longer the test department’s responsibility, it has becomes an organizational responsibility - if you don’t get the quality right you are in serious trouble. The days have gone for a role where an individual manually executes a test script, as the role adds no real value to the team and it is too slow. History has taught us that the manual inspection approach comes too late, can miss critical issues and is not cost-effective. To quote Deming “build quality in” and that’s exactly what agile teams do. Agile and more automation through intelligence-led decision making has resulted in increased business-wide agility requirements, increased pressure from dynamic customer demands which in turn drives some organizations and IT to delivery faster, increase quality yet push costs downwards. Themes I hear coined today are ‘test light and test right’, ‘test optimization’ and ‘test smart’ – none of them involve manual testing.

Another aspect to note is the reference to T-shaped resources. On the vertical bar is a team member’s depth of related skills and expertise in a single field, and on the horizontal bar is the ability to collaborate with experts outside of their own expertise, across disciplines and the ability to apply that knowledge. Since agile development teams are cross-functional and self-organizing there is a need to have broad knowledge across the team. Not everyone needs to be an expert I may add but at the same time, the team member should have deep knowledge or expertise in a functional area, discipline or speciality. This is certainly not the case for team members working within a TCoE.
The Agile Manifesto highlighted the obvious benefits that a self-organizing team brings to an organization as opposed to the heavily documented waterfall approach. Face-to-face conversations are seen as more efficient and effective when discussing the requirements when compared to dealing with TCoEs located offshore and driven by documentation. The latter fails to empower a team to get the job done, to allow them to agree a definition of ‘done’, or to encourage team commitment to delivering production-ready software.

Agile practitioners are there to support the organization’s transformation and change whilst TCoEs provide standards and repeatability. In most cases TCoEs adopt the waterfall methodology whilst a recent trend in agile is to become ‘agile agnostic’ because we all know one size does not fit all. Customization of the approach is required, just look at SAFe, it’s a framework left to the individual organization to describe detail based on their setup. I’ve personally signed up to the Agnostic Agile Oath, it basically recognizes that one framework is not the answer, the driver for change is the ‘what’ and ‘how’ tailored to the customer context and the wider strategic vision.

Organizations need to stop looking outwards, there are many examples of good IT delivery like Netflix, Amazon and Spotify however each of these have adopted their own approach so organizations should start shaping their own destiny. If I provide some examples of the challenges faced by the agile teams which I lead, they are simply very different from those experienced in a TCoE.

1. Increasing capacity and throughput of the team to increase speed to market
2. Defining the approach for delivering agile from an offshore delivery centre reducing cost (scaled agile)
3. Identifying and implementing improvements to continuous integration (CI) processes
4. Streamlining the path to live process and developing a continuous delivery (CD) solution
5. Ensuring the code delivered by the teams can be integrated and tested on a per-build basis (i.e. have an automated, continuous integration pipeline that can deal with the output of more than one team)
6. Ensure that the delivery pipeline (the path-to-live) does not hinder the flow of business value into production, using appropriate DevOps approaches and infrastructure

These factors increase in importance as the number of teams grow, and more so again as the work becomes distributed and teams operate in different time zones. Teams must be independent and resilient as time zone shifts and language barriers will hinder communications.

The TCoE is dead, long live the TCoE?
After seeming to write off the TCoE, I do need to add that there will still be organizations that wish to establish a TCoE. I’ve seen evidence of this recently, but in smaller numbers than I have encountered in the last ten years. The need for a TCoE is now dependent on the type of organization, the industry and its level of maturity. Also, the pace at which software needs to be delivered to the customer impacts on whether it is right to establish a TCoE, if you take the defense industry for example, does this sector type need to be deploying production-ready code daily?

The demand for T-shaped resources has been a game changer because team members need to broaden their skills and knowledge to support the delivery. DevOps and agile have pushed for this, in contrast to TCoE where there is distinct separation of skills.

So, as well as smaller numbers of organizations that would still benefit from establishing a bespoke TCoE, I believe there could be a trend in the future where TCoEs transition to become agile TCoEs, especially where agile is being done at scale to communicate key principles and help with the skills shift needed to deliver working software. This type of TCoE will be very different but could still fulfil a vital role in securing quality.

Paul Mowat is a technology delivery lead senior manager at Accenture specializing in test consultancy/advisory as well as complex delivery.
Even though mobile testing is complex, it can be done successfully with the correct strategy.

A global overview by WeAreSocial.com (2017) revealed that more than half of the world’s web traffic now comes from mobile devices. Additionally, Gartner states that by 2018, more than 50 percent of users will go to a tablet or smartphone first for all online activities. So, it is not surprising that today’s software development is based on a mobile-first, even if it is not mobile-only, imperative. But for all these dramatic changes, developers still struggle when it comes to mobile testing. In this feature, I set out the factors to consider when building an effective mobile test automation strategy and weigh up the benefits and disadvantages of popular test automation frameworks.

There are many factors that can make or break the success of a mobile app. For example, consider device fragmentation. Gartner says that in Q4 2016 there were 432 million smartphones sold in the world. Of that 432 million, 352 million run Android and 77 million run iOS. Those are just new phones - there are hundreds of millions of other devices running older firmware. When your app needs to work across most of these devices, it can become a serious burden for testers.

Another factor that contributes to this complexity is the device itself: varying screen and device sizes, resolutions and orientations. Next, we have multiple app types, such as web, hybrid and native. These multitudes of devices operate differently on various device and OS combinations. Finally, you have users all over the world in different regions that must be tested for translations, timezones and targets. These factors make testing with mobile a challenge. The good news is that even though mobile testing is complex, it can be done successfully with the correct strategy.

When building a mobile testing strategy, there are three key areas of focus:

1. Real device testing
2. Emulator and simulator testing
3. Test automation frameworks

By focusing on these three areas, organizations will thrive in the fast-paced world of mobile software development.

Scale continuous testing with emulators and simulators
For several years, the use of emulators and simulators to test mobile applications has been met with some resistance. This
Mobile testing

is based on the perception that if you’re not testing on a real device, you’re not testing at all.

Although real devices give more accurate test results, using them alone is not ideal for continuous testing and continuous delivery. Due to budget issues, some organizations forgo real devices altogether as they’re too expensive, and instead opt for emulators and simulators. But the reality is that effective and efficient mobile app development requires both emulators/simulators and real devices.

An emulator, as the term suggests, emulates the device software and hardware on a desktop PC, or as part of a cloud testing platform. The Android (SDK) emulator is one example.

A simulator, on the other hand, delivers a replica of a phone’s user interface, and does not represent its hardware. It does not run the real device OS; rather, it’s a partial reimplementation of the operating system written in a high-level language. The iOS simulator for Apple devices is one such example.

Emulators give teams the ability to implement parallel testing and test automation via external frameworks like Appium or Espresso. Selenium revolutionized the world of web app testing by pioneering browser-based test automation. Today, Appium is its counterpart for mobile app testing. Appium uses the same WebDriver API that powers Selenium, and enables automation of native, hybrid, and mobile web apps. This brings huge improvements in the speed of tests for organizations coming from the manual world of testing on real devices. Emulators enable parallel testing in a way that can’t be achieved with devices in a lab. Because tests on emulators are software-defined, multiple tests can be run on tens of emulators at the click of a button without having to manually prepare each emulator for the tests. Further, automation is easier with emulators as the tests can be executed without manual intervention, and be controlled remotely.

DevOps, continuous testing and continuous delivery
The speed of release and change demands that mobile development is agile. Mobile requires that continuous testing (the process of executing automated tests as part of the software delivery pipeline to obtain immediate feedback on the business risks) is a key component to the overall regression testing strategy. In order to properly build out a mobile regression testing strategy, it is imperative that the dev/test teams are well equipped with the following:

- **A comprehensive web and mobile testing platform**
  building a test lab from the ground up is timely and expensive. The best option is to use a cloud-based, device lab solution that provides an extensive choice of real devices, as well as emulators and simulators. This should also include parallel testing so that test execution can be done in a shorter amount of time.

- **Highly scalable, highly available solution**
  developers and testers need to ensure that the infrastructure for mobile testing allows the team to expand coverage as the test suite grows. The goal is to decrease test execution time while providing fast feedback, and to ensure that the team spends less time chasing false failures and more time on actual testing and defect remediation.

- **CI/CD optimization**
  in order for the regression testing efforts to keep up with the fast pace of continuous delivery, the process must have a tight integration with the development workflow and mobile app delivery pipeline. The goal is to eliminate any unnecessary manual steps, and promote an automation-first philosophy throughout the testing process.

---

Figure 1: The user numbers driving the mobile challenge

<table>
<thead>
<tr>
<th></th>
<th>Laptops &amp; Desktops</th>
<th>Mobile Phones</th>
<th>Tablet Devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year-On-Year Change:</td>
<td>-20%</td>
<td>+30%</td>
<td>-5%</td>
</tr>
<tr>
<td></td>
<td>45%</td>
<td>50%</td>
<td>5%</td>
</tr>
</tbody>
</table>

Figure 1: The user numbers driving the mobile challenge
Although mobile regression testing can be a challenge, the risks of flaky software can be reduced by making sure your organization has the right strategy in place.

Mobile test automation frameworks
Test automation frameworks are an integral part of mobile test automation. There are many test automation frameworks freely available, but the top mobile testing frameworks are Appium, Espresso and XCUItest. Sauce Labs recently conducted a survey of teams using test automation frameworks for mobile app testing. The survey found that 63 percent of the participants use the Appium framework, while 31 percent didn’t use any test automation framework at all. Since Appium is based on Selenium (the popular open source functional testing framework), it wasn’t a huge surprise that most of the users surveyed were using Appium. But the main question is why is it that over 30 percent of users were not using some sort of testing framework? The answer lies in the implementation, or lack of a mobile testing strategy.

When implementing a mobile testing strategy, it is very important to understand

1) the skillset of the test automation team and
2) the framework that best fits the organization's preferred development methodology.

Tester automation skill set
When evaluating frameworks, it’s important to understand the technical background of your team. Some frameworks take a “black box” test approach, which is typically best for traditional test automation engineers. While other frameworks take a “white box” approach suitable for developers. There are pros and cons to each framework, but let’s explore the top three.

Appium
From Appium.io:

“Appium is an open-source tool for automating native, mobile web, and hybrid applications on iOS and Android platforms. Native apps are those written using the iOS, Android, or Windows SDKs. Mobile web apps are web apps accessed using a mobile browser (Appium supports Safari on iOS and Chrome or the built-in ‘Browser’ app on Android). Hybrid apps have a wrapper around a “webview” – a native control that enables interaction with web content.”

The key advantages of Appium are:

- A single, cross-platform framework for both Android and iOS
- The ability to automate mobile web, native and hybrid applications
- Tests can be run on emulators, simulators and real devices
- Supports any language supported by Selenium, such as Ruby, Java, JavaScript, Python, C#, etc.

Appium is one of the more popular testing frameworks for those organizations that typically have testers who have

---

Figure 2: Emulators, stimulators and real devices cover all use cases
worked with Selenium and are currently making the transition to mobile development. Since Appium is a black box type framework, the tester has less insight into the code, and the focus of testing is limited to what is exposed by the mobile application under test.

**Espresso**

The Espresso testing framework provides a set of APIs to build UI tests to test user flows within an app. These APIs let you write automated UI tests that are concise and that run reliably. Espresso is well-suited for writing white box-style automated tests, where the test code utilizes implementation code details from the app under test.

The key advantages of Espresso are:

- Espresso is that it is super-fast and reliable. It is fully integrated with Android Studio, the preferred mobile development platform for Android apps. It’s based on Java or JUnit, which many android developers are familiar with. Since Espresso is integrated with Android Studio, it’s automatically CI/CD ready, allowing teams to easily incorporate their test with existing continuous integration & delivery tools and frameworks.

- Another advantage of the Espresso test framework is that it automatically provides updates to the latest and greatest features of the Android operating system. This helps keep testing in sync with new features and improvements as they become available. In addition, Espresso’s integration with Android Studio provides backward and forward compatibility. This allows teams to ensure their app works on previous releases of the Android OS.

- A huge benefit for testing with the Espresso framework is automatic synchronization. Many testers struggle with UI testing reliability because of waits for elements to become visible or active. With other frameworks, the developer must write code to manage these issues. Once the object appears, Espresso handles the execution on that object for you. This results in less “flakiness” of test scripts and more reliable regression test suites.

- Espresso, XCUITest and Appium are all useful frameworks for writing tests for mobile applications. But a question to be answered is really what is the best framework for the job. No one tool is applicable to all situations. Espresso and XCUITest are great tools for mobile testing when source code is available and the developer is working in an IDE. Appium is the mobile application testing tool to use when all that is available is the API or APP deployment artifacts. And all come with tradeoffs. Finding the right test automation framework for the right job is a hard determination to make.

**XCUITest**

XCUITest is the automation framework that ships with Apple’s XCode. The XCTest framework lets iOS app developers create and run unit tests, performance tests and UI tests for XCode projects. It provides developers with capabilities similar to those found in Espresso. However, Espresso is dedicated to code written for the Android operating system and XCUITest is dedicated to Objective-C and Swift code that runs under iOS.

XCUITest was created specifically for testing iOS apps and is maintained by Apple. This ensures that developers get the best support from the Apple developer community as well as the latest updates for new releases on iOS. Additionally, the
ability to run unit, UI and performance tests allows more comprehensive testing within a single framework.

If the development of the app is strictly for iOS then the developer has everything at their fingertips. If the app is cross platform using XCUITest it can be a slight disadvantage - for instance, the issue of developer skill set. Tests written for XCUITest can only be written in Objective-C or Swift code. A team that has developers experienced in Java, JavaScript, or any language that Selenium supports must now learn another scripting language and manage two frameworks. This is another reason Appium has become popular with test automation engineers that support multiple mobile app platforms.

Testing frameworks are a key component of mobile application development. With Agile, DevOps, continuous testing and integration, developers and test automation engineers must continue to develop, test and execute at a rapid pace.

Frameworks such as Espresso, XCUITest and Appium make automation easier to scale for the SMBs to the enterprise.

Conclusion
A sound mobile test automation strategy must include test automation frameworks, real devices, and emulators and simulators. Each component is critical to the rapid release of well tested software.

Agile, DevOps and continuous testing are the norm. Mobile software development naturally follows this paradigm. Implementing these components as part of the mobile testing strategies will increase the chances of successful releases and decrease the chances of poor app quality.

Clinton Sprauve has more than twenty years of experience in the software development and quality assurance industry is now the product marketing director for mobile at Sauce Labs.
Requirements coverage – a false sense of security?

by Gregory Solovey

Model Driven Testing is a practical approach to test completeness

1. Test completeness
If you search Google with the query “what is the meaning of complete testing?” it yields some fairly philosophical answers, pretty much along the lines of “completeness is unfeasible.” This is true, but “best or nothing” is not an acceptable approach. “Just do it” seems better, but it is not measurable. A solution may be to use “coverage you can rely on”. The primary coverage that comes to mind is “code coverage”.

A time-honored panacea, code coverage firmly took its well-deserved place in unit test. Nowadays unit test tools exist for most programming languages; they allow you to automate unit test for the objects’ properties/ methods and calculate the code coverage. Code coverage, however, hasn’t proved its usability for functional, integration and business-level tests. This is where the requirements coverage came into play. The allure is the simplicity of the implementation: given a set of requirements and a set of tests, all you need to do is map them and calculate coverage. Good intentions, but we know where those often lead...

This article sets out to highlight the pitfalls of relying on requirements coverage as a measure of test completeness. Model Driven Testing (MDT) and implementation details are presented as a reasonable alternative for achieving test completeness.

2. Requirements coverage
The test tool market responded quickly to the requirements coverage demand and today major test management systems offer requirements coverage functionality, for example HP Quality Center, and some companies build their own in-house solutions. The test cases that belong to a new feature are tagged with requirement IDs. During Test Plan reviews the requirements are analyzed for their coverage and the tests for their completeness. Very frequently requirements are specified in plain text, which makes them subjective and difficult to assess from a test coverage standpoint. There are no formal methods for building test cases from text and, as a result, the test completeness can only be “guaranteed” by insuring 100% requirements coverage. While offering the obvious advantage of simplicity, this type of coverage creates hidden disadvantages for test quality. The requirements coverage creates a false sense of security.

The well-known axiom is that testing can only find errors. Error types are not defined for business requirements expressed in...
plain text. In contrast, error types are defined for formal models described in Unified Modeling Language (UML), an ISO standard. Any business logic can be presented by only a small subset of UML diagram types (such as use case, activity, state machine, sequence diagram) and for each UML diagram type there are known test design methods. Therefore, the test completeness can be simply verified by checking that the right test design methods were applied to the model.

Unfortunately, existing test management systems do not provide modeling management support, therefore making the measurement of the following two important quality metrics challenging:

1. How complete is a single Test Plan (how well a single TP covers the implementation errors)?

2. How complete is a set of Test Plans (how well is an entire system covered by the overall TP set)?

In contrast to test management systems, modeling tools [1] allow you to present a system’s structure and behavior using UML. Some modeling tools include unit test design automation for simple models (single diagram); this is inadequate for the scale and scope of average systems. Most systems are represented by complex UML diagram hierarchies. There are no references to automated test generation for production-scale systems in test-related publications. Therefore, this feature is not useful in support of test completeness.

This article presents a practical proposal for Model Driven Testing, where the test management system is built into a modeling tool, and an approach to support the end-to-end traceability and test quality.

3. Test as a mirror of the model
It is easier to incorporate a test management system into a modeling tool than vice versa. The testware has to be mapped to the system model in order to verify all of its diagrams and their elements.

3.1 System modeling
For the sake of discussion, let’s look at the system structure as a multilayered object, for example: application or business, middleware or platform, HW abstraction, interfaces, DB layers, etc., where each layer consists of independent subsystems, aka as services or components, as shown in Figure 1. In the behavioral view, a subsystem can be presented by a specification/design model, which is presented as a hierarchy of UML diagrams - use case diagrams, activity diagrams, sequence diagrams, state machine diagrams, etc. and as shown in Figure 2. Figure 3 illustrates the combined structural and behavioral views, where a system is described by a set of subsystems, which in turn are presented by a hierarchy of UML diagrams.

A new feature can be described by a subset of diagrams of the affected subsystems, which are going to be edited, updated or deleted during the feature development.

3.2 Testware hierarchy
The central object of any test management system is a Test Plan, which is a hierarchy of test objects. The use of a hierarchy is a means to make the tests modular, reusable, maintainable, and readable, and to reflect the structural-behavioral view of the object-to-test. In my experience, four test layers are usually sufficient, for example let’s call

**Figure 1. Structural view of a multi-layered system**

<table>
<thead>
<tr>
<th>Application/ Business layer</th>
<th>Application 1</th>
<th>Application 2</th>
<th>Application 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Middleware/ Platform layer</td>
<td>Service 1</td>
<td>Service 2</td>
<td>Service 3</td>
</tr>
<tr>
<td>HW abstraction layer</td>
<td>Component 1</td>
<td>Component 2</td>
<td>Component 3</td>
</tr>
<tr>
<td>Interface layer</td>
<td>API 1</td>
<td>API 2</td>
<td>API 7</td>
</tr>
</tbody>
</table>

**Figure 2. Behavioral view of a subsystem as a hierarchy of UML diagrams**

**Figure 3. Combined structural and behavioral views**

**Use case diagram 1**

**Activity diagram 9**

**Activity diagram 3**

**Sequence diagram 9**

**Activity diagram 5**

**Activity diagram 2**

**Activity diagram 5**

**Sequence diagram 7**

**State machine 8**

**Activity diagram 3**

**Activity diagram 3**
3.3 Model-test mapping

We implemented the following approach to integrate a modeling tool and a test management system. The assumption is that the structural view of the system is a set of subsystems and interfaces among them. Each subsystem, in turn, is represented by a behavioral model, as a UML diagram hierarchy.

In addition to its behavioral model, each sub-system has to be associated with a hierarchy of test objects (as in Figure 4). This means that four test hierarchy layers must be put in relationship with several diagram layers.

The first test layer is a set of various test plans for a subsystem, for example regression, sanity, several feature and performance test plans. A regression test plan has to check all subsystem diagrams. A feature test plan has to check only the diagrams that describe that feature’s functionality. A test suite will be created for each diagram that needs to be checked.

A use case is a behavioral aspect of a diagram. A diagram presents a set of scenarios, tasks, procedures, functions, etc. A use case will be built for each behavioral element of the diagram. A use case is a collection of test cases. Each test case initiates a set of actions that produce a system response to be compared with the expected result.

All test cases should be built using known test design methods for the specific UML diagram type [2].

Traceability and coverage

With this background, we can now illustrate the benefits of having a test management system within the modeling tool.

Test completeness can be controlled in two dimensions. The structural completeness means that all subsystems and their diagrams are covered by regression tests. The behavioral completeness means that the test design method appropriate to the particular diagram type was correctly...
The structural completeness can be easily measured in an automatic manner, for example, as a percentage of subsystems covered by regression tests.

The behavioral completeness can be verified only during the test review. There is no automatic coverage calculation that can be used today, because this task is as complex as the automatic test generation. The requirements coverage can still be checked during the review, but the important difference is that reviewers would analyze the test against formal UML diagram types, instead of against complicated, informal business requirements.

In addition to test completeness, the symbiosis between the modeling tool and the test management system can provide the end-to-end traceability and coverage for all product development artifacts, as shown in Figure 5. A modeling tool uses a graphical view (traceability matrix) to present the various artifacts’ coverage, for example the requirements coverage by diagrams. The existing traceability feature can be reused for the traceability of the testware and test scripts. Furthermore, all artifacts should be identified by tags: requirements, system components, diagrams, test hierarchy and automated test scripts.

During each step of product development, starting from customer requests, through requirements, design, test, and finally to automated scripts, it should be possible to measure the coverage of artifacts generated in each process step by artifacts produced in the next step. The coverage can ultimately be cumulated across all system artifacts.

Most of the artifact data exist in the modeling tool/test management system and can be used for the coverage calculation. Only the test execution results, which are available in log files, exist outside and have to be uploaded into the test management system from the CI/CD/DevOps environment.

The modeling tool provides two system views – the “living” view and the release/feature view, and therefore, there are two possible monitoring approaches:

- Monitor the coverage of the entire system and/or its components by existing regression tests (legacy test quality)
- Monitor the coverage progress over time of the new release/features by the new feature tests. This also allows you to compare projects.

A quality dashboard shows various KPIs (such as density and coverage) which are accumulated from the test objects. Here are a few test plan-related KPIs: test-code density, based on number of lines of code and number of tests, model-test density, based on the models’ complexity and number of tests, test automation percentage, model coverage by tests, testware coverage by automated tests, requirements coverage by tests.

**Conclusion**
Just to illustrate the vulnerability of requirements coverage, let's consider the following example. The system (a status display) is presented by only a few requirements:

```
Doc repository: requirements and acceptance criteria
Modeling tool: specifications and Design
Test management system: abstract testware
Source control system: test scripts
DevOps environment: logs and reports

Quality dashboards:
- system components coverage by test
- new features coverage by test

Figure 5. End-to-end traceability process
```
RPREQ_1500: As Application_3 SW, I want Platform SW to control state LEDs during early HW start-up, warm and cold reset, so that state LEDs indicate when module is in resetting phase and initialized.

RPREQ_1747: As Application_3 SW, I want Platform SW to set color and pattern on state and interface LEDs according to received requests, so that I can see a state of a particular unit.

A few test cases can be written to cover the existing two requirements (and report 100% of coverage), but in reality, there should be many more test cases if the model would be taken into consideration.

In this particular example, a model is implemented by the following diagram hierarchy:

- Use case diagram (1 diagram): start-up, cold/warm reset/OFF for various boards
- Activity diagram (7 diagrams): Algorithms/conditions of start-up, cold/warm reset/OFF
- Sequence diagram (4 diagrams): Message exchanges for LED settings

The model associated with these requirements includes more data, such as:

- HW/OS/browser variants and respective state machines.
- Different number of LEDs per HW variants
- Different sequence of colors (RED, GREEN, YELLOW) and color pattern changes for various events (stable, blinking, alternating).

When using formal test design methods, more than 240 test cases are required to find all implementation errors. But testers can get away with writing two test cases to cover the two requirements.

Somebody can argue that the requirements may include more implementation details and provide enough data to build all necessary test cases. Theoretically yes, but in reality, I have never seen such a case.

The only way to deal with test completeness for legacy systems and new features development is to have the system description specified in UML. Maintaining a test repository together with the models dramatically simplifies the traceability, improves coverage, simplifies maintenance, and more important, allows to find all implementation errors during the development cycle.

Gregory Solovey PhD, is test automation architect at Nokia and frequent contributor to Professional Tester.

References:
Anthea Whelan asks Janet Gregory what lessons have to be learned from her experience transitioning traditional test teams into the agile world

It's now been over a decade and a half since the Agile Manifesto was released in 2001. Do you still have to encourage people to go agile?

When I introduce myself, I say I have been working with agile for over 17 years now – I cannot believe how fast that time has gone. I am still working to help teams transition to agile, although it is fair to say how I work with them has changed over the years.

I'd love to hear some of the funniest "reasons" you've been given for not adopting an agile approach?

I'm not sure there are any really 'funny' reasons. The people and organizations fully believe that their teams or their organizations are different than everyone else’s. And it could be that it is different, however, most of the underlying problems I see are shared by almost every organization I go into. Issues such as management not changing their expectations in reporting, or thinking people can be team members on multiple teams and still do a good job for all teams.

The very first page of your book Agile Testing: A Practical Guide for Testers and Agile Teams suggests that agile development will one day fall out of fashion, just as waterfall now seems to have. What do you think might replace it? Why do you think this change will come?

We did say that agile is a buzzword. I still think it will be replaced, but far less slowly than I would have imagined. Phrases like DevOps are being used now to include operations into the picture but I still see that as an extension of agile. Joshua Kerievsky has introduced the idea of “Modern Agile”

Kerievsky has taken the values and principles from the Agile Manifesto and has adapted them. These updated guiding principles are:

- Make people awesome
- Make safety a prerequisite
- Experiment and learn rapidly
- Deliver value continuously
That may sound like marketing slogans but I think what he is promoting is the whole ecosystem of a project must be taken into account. To foster a health environment, people have to feel confident that they can experiment, make mistakes and innovate if organizations are to get beyond delivering working software and deliver value continuously.

A safer, supportive psychological culture will help unlock high performance, I think this may be the next iteration of agile and if we are truly agile, we should be constantly adapting our thinking.

Management-speak is once more peppered with phrases such as "adding value" to production processes. What's your advice to developers and testers who have been told this?

This made me laugh because I use that phrase myself quite a bit. I encourage everyone on a team (programmers, testers, business analysts, …) to ask themselves if what they are doing is adding value to the end product or to someone. If not, stop doing it. One of the biggest hurdles is trying to understand who is getting value out of a process, so I suggest that they start asking the question. If they can't find out who is getting value out of a particular process, it may be obsolete.

In very structured organizations, what's your advice for encouraging coders and testers to adopt a less siloed approach to development? What initiatives could they try to demonstrate the advantages of this approach?

Very structured organizations usually have very strict rules. These are the organizations that have the hardest time adopting agile because they often do not encourage people at the team level to make decisions.

However, individuals can still do small things, like trying to work in a more collaborative manner. For example, if I were a tester on an independent test team, I might start by asking a programmer to “show me” what was developed before I started to test. It would be the start of a better relationship to hear what they have done and ask questions to seek to understand before I start reporting bugs.

Can you recall for us one of the more challenging projects where you had to manage expectations? What differences have you experienced between managing Management expectations and managing those of the team?

I have been working with one organization which was quite aggressive about changing how they worked and wanted to get everyone trained. However, the training department didn’t quite get the idea of ‘whole teams’ needing to be trained together, and didn’t quite get “who” needed what training. They tried, but the implementation of the training left much to be desired. I had to reset expectations, and refused to provide more training unless something changed in their methods of training. That was very difficult for me, but we worked it out.

The biggest difference in setting expectations between management and the delivery teams, is the time I am given. Management will give me an hour of their time, but I get days / weeks / months with the teams. It would be great if management would consider a bit more time learning, as a good investment in themselves.

What irritates you the most - what’s your favourite thing to rant about - regarding the way most businesses implement their feedback cycles/loops?

I’m not sure that is something I rant about too much, but upon more consideration…I would have to say unrealistic expectations. For example, managers often want all the same reporting mechanisms (such as metrics) they have always had, but don’t realize the burden that puts on the team.
The reporting part of the feedback loop needs to be realistic if the teams are expected to work in short cycles. Transparency is critical, but is there a way to do this in a simple manner.

One of the biggest challenges that repeatedly crops up seems to be not losing sight of the big picture, especially when implementing complex requirements. What practical advice do you have for teams who find themselves trapped in this sort of environment?

Complex and complicated features are common, so I don’t think we should think of them as being trapped, but something we have to learn to deal with.

One of the success factors in our first book was to remember the big picture. At that time we gave some general ideas, but in the second book we talk about more concrete practices. One idea I have had success with is defining not only ‘Story Done’, but also ‘Feature Done’ and ‘Release Done’. This makes it clear to the team that there are different levels to think about – especially from a testing perspective.

I often encourage teams to define an extra story when slicing up a feature that might be something like “Complete the feature”. The tasks for the team on this story might be things like exploratory testing on the feature, performance testing, load testing, and user acceptance testing. Note - these tasks do not have to be done only by a tester.

What problems seem to arise uniquely in agile environments, that don’t seem to appear in other methods of managing software development cycles?

I once asked a VP of Development what he thought the biggest differences were between the agile projects and the waterfall projects. I thought his answer summed it up quite well.

In a waterfall project, everything is great until the last 10% of the project and then everything falls apart. So, 90% of the project is no issue for me. On the agile projects, the teams are coming to me every day with small problems I need to deal with.

Of course, I was still confused…. I asked him which one he preferred because I really couldn’t tell from the answer. He laughed and replied, “The agile one”. It was because the small daily issues were ones he could solve easily and keep the customer in the loop in an open honest fashion. The problems at the end of the waterfall projects usually required finesse dealing with the customer involving some half-truths, and usually a delay or something worse.

Testers: don't tie yourselves in “Nots” 

by Dorothy Graham

Here is your handy guide to dispelling the misunderstandings often found in testing

This article is a collection of “Nots” related to software testing, a tour of misperceptions and misunderstandings about testing, test automation and people. Think of it as ready shorthand for the next time you find that you need to clarify what is being said and the impact of what is under discussion. It can save you a huge amount of time and frustration.

**Testing “Nots”:**

- **A test that fails is not a failure**
  - a test that shows a bug is useful, not a failure

- **A test that passes is not a waste of time and resources**
  - a test that shows something working gives confidence (in something)

- **Bug-free software is not the same as quality software**
  - it may perfectly do something no one wants or needs

- **Bug-free software is not possible**
  - you can never test everything, so you don’t know about bugs you haven’t found

- **The quality of testing is not measured by the number of bugs that you find in test**
  - although there is one aspect of test quality that can be measured by the number of bugs: the ones you missed (if/when they are found later)

- **System testing is not a repeat of unit testing**
  - unit testing is on a micro scale, system testing takes a wider view

- **Testing is not just running tests, not just designing tests, not just playing with the system**
  - all of these are part of testing, and more; testing is more than any one of these

**Test Automation and Tools “Nots”:**

- **Test (execution) automation is not testing**
  - this automation is a form of running tests, but test execution is only one aspect of testing - the other aspects are deciding what to test, deciding how best to test it, implementing a test, evaluating test results
Capture/replay is not test automation
- what you capture when you “record” a test is input only (not a test, part of a test), everything you typed is “hard-coded” into the test, the script is tool-based and environment-specific, and there is no evaluation of test results unless specifically added, but this is more than capture. (Chapter 2 of the book I co-authored with Mark Fewster, Software Test Automation, covers this in more depth)

Test execution tools don’t assess pass/fail
- yes, that’s what they report, but actually they assess “match” or “no match” to their stored “golden version” of expected results. We assume that a match is a pass, but the expected results could be wrong

Coverage is not “running all my tests”
- this is “test completion”. Test coverage is the percentage of some element that has had a test run through it. (See my EuroStar webinar about coverage )

“100% coverage” is not the same as “tested thoroughly” (or “tested enough”)
- coverage, even 100%, is only of whatever aspect has been exercised. There are many more aspects, coverage says nothing about whether or not the software actually passed those tests, coverage only needs one test to “tick the box”, and it looks only at what is there, not what is missing (see webinar mentioned above)

People “Nots”:

People are not inter-changeable units, a team is not a collection of such units
- each person has unique abilities and characteristics. The best teams are where everyone is valued, all have complementary skills and personalities, and there is open and honest communication

A training course does not make you a skilled tester
- a training course (including ones with certificates) can remove a “bottom layer of ignorance” about testing by imparting knowledge about testing principles and techniques. Skills are only developed through practice and experience. Knowledge without skill can be a good starting point; skill without knowledge may be inefficient or even dangerous

Telling someone about something is not communication
- “telling” is one way - from you to the other person. Communication is two-way, with feedback, comments, questions and verified understanding from the other person

All testers should not be test automators
- the skills of the tester are different to the skills needed for test automation. To work directly with the test execution tools, you need to be a good developer. Although many testers are happy and skilled to do both, not all are, and those who love testing and don’t want to write code, should not be forced to, or be de-valued if they don’t, in my humble opinion

And one final “Not”:

“X” is not a panacea, where X = certification, coverage, exploratory testing, test-driven design, agile, skill, techniques, management, test process improvement, DevOps, CI/CD (and more)
- enough said? ■

Dorothy Graham provided training and consultancy in software testing for over 40 years, and has been involved in many testing conferences and events. She is co-author of four books (two on test automation) and the wiki: TestAutomationPatterns.org. She holds two Testing Excellence Awards.

www.DorothyGraham.co.uk
Christoph Preschern answers Anthea Whelan’s questions on how software testing is changing and what skills should not be overlooked

The debate about whether testers should also be coders has been continuing for a while. What's your own view on this?

Testers are increasingly concerned about this issue. Recently, I attended a vendor presentation at one of the biggest testing conferences in the US. One statement made during this presentation was that you should learn to program today; otherwise, you won't have a job tomorrow. I think that isn't really fair. This is not motivating – instead, it's scary. It's true that these days more and more tech skills are required in the testing space.

During my career as a developer and tester, I saw many developers become frustrated because they knew that they were not the best in their job, so they were relegated to bug fixing. That’s not exciting work. What could be more fulfilling is becoming an automation expert and serving testers with a robust test automation framework. In addition, someone who has already been part of a dev team could help improve communication between testers and developers. I personally think that it is important to recognize that testing, especially on the system and integration level, still requires traditional testing skills and business knowledge about the user's perspective. Someone who has good coding skills and is fully focused on automation during the SDLC, might not have a deep understanding of what it means to create a good test case.

Do you still have to deal with code-heavy automation purists? How do you handle them?

Things are very different now than in the beginning of Ranorex. Especially within the first two to three years, we offered just an Automation Component (API) and a Spy tool. Back then we were focusing on those “early-adopters” of shift left testing.

Today, with our growing feature set, users can decide where and how to use the Ranorex core test automation engine. This core automation engine today is also the base for everything the Ranorex Recorder does. It’s true that capture & replay has its limitations, but, especially for testers with no programming skills, it’s the easiest way to get started with automated testing. In addition they can learn scripting step-by-step, since every single line of code generated by the Recorder is easy to access and to edit.

There are quite a few Ranorex users who are working in their existing dev-environment, implementing powerful and sophisticated automation steps. While our code editor is great for quick coding actions, we don't want to limit a purist who prefers Eclipse or Visual Studio. It is more important for us,
that this group can continue using the environment that they prefer, and leverage our automation APIs there. The beauty of this approach is that test modules created this way can easily be used by other team members, whether or not they have a VisualStudio license.

Now that HP seems to have completely bowed out of the testing arena, should businesses that are focused on testing be concerned?

For sure – the traditional way of software testing is changing. Tests which usually have been managed with classic TM tools are now shifting to CD/CI environments. Just look at the explosive growth in Atlassian’s tool set within the last five years. The services of testing professionals are especially needed now to help with this transition from traditional to agile test management.

A lot of organizations are struggling with this transition. Apart from the tools used to support testing, businesses should not forget about the foundation of the software testing profession: a tester’s perspective and skills, such as system-level understanding, and hunger to break a system under test. These are needed now more than ever. Why? Everyone is talking about automation. This is really great! Especially for tool providers like Ranorex – but, to be honest: everyone knows that the best automation engineer can rarely become a good tester. A test automation expert is more like a developer. He loves the challenge of getting things automated that others can’t.

Testing has seen its fair share of fads and trends over the years. Where do you think software testing trends will take us in the next decade, especially when compared to the last ten years?

Automated testing combined with easier access to virtualization and cloud infrastructure allows IT organizations to gain much faster feedback about their system’s quality than ever before. Execution will become much more important. We are already seeing an increased demand on test execution infrastructure. One of our biggest clients in the US, Symantec, is using hundreds of Ranorex runtime licenses for parallel test execution in the cloud. It may seem counter-intuitive, but the more convenient and consumer-friendly software products become – the more complex they are to test. Back in 2011, Ranorex was one of the first tools in the market supporting automated testing on mobile devices. Then, and now, it is still one of the biggest challenges for automation tool providers to deal with closed systems like Apple.

Definitely the next big thing is how to test the world of IoT. We believe that automating integration and system-level tests in this field is only possible by mocking or simulating microservices in the backend.

What are some of the issues that your own development team has encountered in delivering an interface orientated around the non-coder? Which were you able and unable to overcome?

Providing a powerful, flexible and at the same time, easy-to-learn test automation tool is challenging. Although simplicity is very much needed in order to become productive quickly with automated testing, many experienced testers out there don’t trust shiny tools. Perhaps they burned their fingers with capture/replay tools. Because of this, the Ranorex core test-automation philosophy is always to support both approaches. Our motto is, “APIs first, tools second”. I think we have done a very good job with supporting both audiences with their day-to-day test automation challenges in the past, and building the bridge between both extremes will become more and more important to these cross-functional teams.

What currently insurmountable problems within the field of testing would you most like to see resolved, soon?

This is hard to say. Maybe more design patterns or standards for testability? It’s disconcerting to see that even modern software tool providers like SalesForce or big brands like Apple are not taking “design for testability” into consideration in order to make test automation easier. A second thing is more education in regard to software testing at universities and colleges. Universities tend to focus on software development. This makes it challenging to grow solid software testing people – especially in the field of automation.

What are some good automation framework testing habits to develop? For instance, if you could only pick two pieces of advice, what would they be?

Consider testability for automation on every layer of your system under test.

Make sure your automation framework – or at least the reports generated by it – are available and easy to understand for PMs, POs, TMs. Business owners have to understand what automation is doing for them: for example, which issues have been identified or how much faster the organization has been able to release product updates to the market.

-------------------------------------------------------------------------------------------------

Christoph Preschern is managing director and chief strategic officer at Ranorex

-------------------------------------------------------------------------------------------------
What's your big idea?

With thousands of subscribers, make sure that your solution is showcased.

Be part of the conversation.

Be part of the future of testing.

Advertise in Professional Tester.

Essential for software testers.