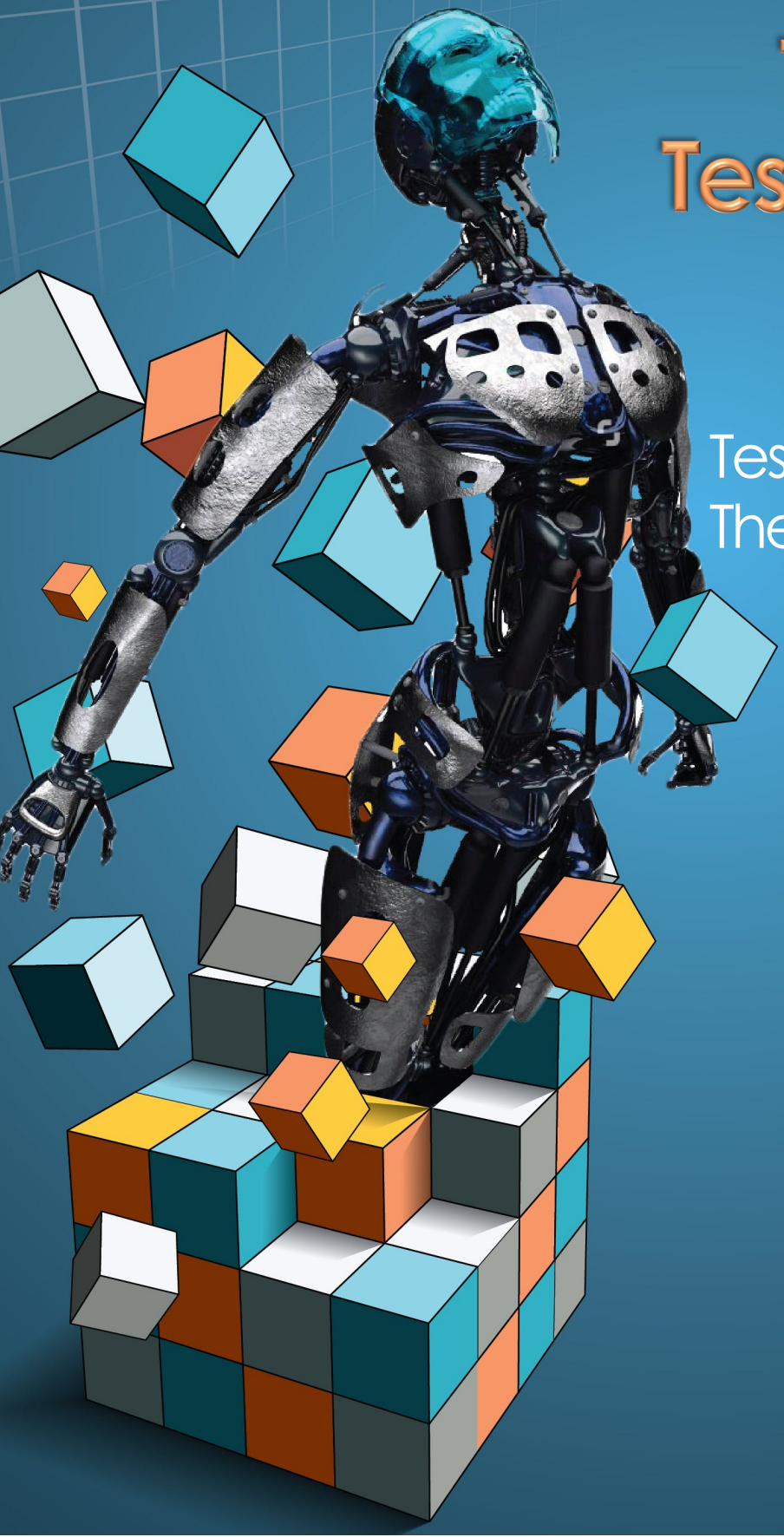
SHOWCASING THOUGHT LEADERSHIP AND ADVANCES IN SOFTWARE TESTING

LogiGear Magazine



The Rise of Test Automation in Agile

Testing Agility in the Cloud: The 4Cs Framework

Sumit Mehrotra

Implement UI Testing without Shooting Yourself in the Foot

Gojko Adzik

Get Automated Testing "Done"

Hans Buwalda & Subu Baskaran

Cruise Control: Automation in Performance Testing

Tim Hinds

Test Automation: The backbone of Continuous Delivery

Ranjan Sakalley

DECEMBER 2015
VOL IX
ISSUE 4

Editor in Chief
Michael Hackett

Managing Editor
David Rosenblatt

Deputy Editor
Joe Luthy

Worldwide Offices

United States Headquarters
Silicon Valley
4100 E 3rd Ave, Suite 150
Foster City, CA 94404
Tel +1 650 572 1400
Fax +1 650 572 2822

Viet Nam Headquarters
1A Phan Xich Long, Ward 2
Phu Nhuan District
Ho Chi Minh City
Tel +84 8 3995 4072
Fax +84 8 3995 4076

Viet Nam, Da Nang
346 Street 2/9
Hai Chau District
Da Nang
Tel +84 511 3655 333

www.LogiGear.com
www.LogiGear.vn
www.LogiGearmagazine.com

Submission guidelines may be found at:
http://www.LogiGear.com/magazine/issue/news/2016-editorial-calendar-and-submission-guidelines/

Every year, LogiGear Magazine devotes one full issue to Test Automation. We could do more than one, and perhaps even that would not be enough.

The problems around automation have become increasingly complex. And now, automation is much more integrated into the software development process. For over a decade teams have been faced with "do more with less": do more testing, do more automation, and do it all with less staff. Then Agile/Scrum came along and we had to automate it faster. As the XP practice of continuous integration (CI) caught fire, our automation suites – smoke tests and full regression suites – got integrated into the autobuild development process, which in most cases was out of our control. Other people and tools are now running our automation and reporting back results – not by us kicking off automation when we choose to, but whenever a build takes place.

Today this process is moving at an even more extreme pace and further away from us. We see CI moving onto virtual machines and DevOps running our automation all the time (continuous testing), on all kinds of environments.

Many teams are still struggling with getting automated test into their current sprints, or Sprint +1 (getting new functionality automated, but only in the sprint following that function's development). Some teams struggle just to get more tests automated in their development cycle at all, and end up settling for adding new automation after a release, because they just do not have the time. This is not OK. If this is your situation, you need to fix it. It may not be an easy fix, but not fixing it has a negative impact on development.

What do we have to do?

- First, automate more and automate faster. With shorter cycles, you need automated tests, or you will never reach levels of coverage acceptable enough to have confidence in your product. Yes, automate faster.
- You need a framework with reusable and low maintenance functions.
- Finally, choose effective methods. We all know the idea that tests need to be low maintenance. But how do you do that? When you have a big suite of tests and some break – and not because of application bugs – how do you unbreak the test suite to run again? Simply automating step-by-step test scripts is a surefire formula for failure. Instead, choose a more sophisticated method for developing tests, like Action Based Testing.

Our tests have to be effective at validating functionality and finding bugs or breaks. And they must be efficient – suites should do this in the minimum number of tests possible.

We know that our tests are going to be run, in most cases these days, across a large matrix of configurations, browsers, devices, and appliances. In addition, now the tests will more than likely be run on a variety of build environments. It is becoming increasingly common to run the same suite of tests on a dev environment, testing environment, user acceptance or staging environment, and sometimes live/production environments. For some tools and suites, the performance demands are too great: the tool itself becomes an issue, not just the suites it runs. I myself have used some tools that develop huge problems running tests as the number of virtual machines increases. And that is only the start.

Our automation has to get better. But more automation is not always the answer. Today, the answer must be: better and faster automation. I hope this issue of our magazine gives you valuable guidance to achieve this.

We've also just published our 2016 editorial calendar, to give you an idea of what's ahead for next year. As always, if you'd like to submit an article, just let us know.

All of us at LogiGear wish you a joyful and healthy holiday season and a happy new year. We look forward to continuing to provide you with great software test information in 2016! ■

ARTICLES

BOOK REVIEW

BRIEFS

GOOGLE WORKING TO GET ANDROID UNDER CONTROL

In an effort to take on the issue of fragmentation – one of the most vexing problems with Android – Google is negotiating deals with chipmakers, The Information reports. By working directly with chip partners who implement its designs, Google plans to create hardware that works seamlessly with Android, just as Apple products do with iOS.

According to one estimate, there are over 18,700 distinct versions of Android, compared to Apple's five or six. This fragmentation of the Android ecosystem has become a major headache for Google, not to mention phone makers and testers. By exerting more control over the Android world, Google would not only make life easier for everyone, but also be in a better position to offer new features more rapidly.

- ***Source:*** **eWeek** *Why Google Might Want to Design Chips for Android Phones*

WHY PERFORMANCE TESTING MATTERS

Discount retailer Target Corp's website was down due to heavy traffic on Cyber Monday.

Shoppers looking for bargains on the target.com website were greeted with the message: "So sorry, but high traffic is causing delays. If you wouldn't mind holding, we'll refresh automatically & get things going ASAP."

The retailer wasn't alone in glitches during the high profile shopping holiday. Other websites with outages included Victoria's Secret and Foot Locker.

- ***Source:*** **Reuters** *Target website down on Cyber Monday traffic*

LOGIGEAR MAGAZINE 2016 EDITORIAL CALENDAR

LogiGear Magazine has just released its editorial calendar for 2016. The magazine, published on a quarterly basis, dedicates each edition to a particular theme, one of relevance to the dynamic field of software QA. Our plan for 2016:

March	**Test strategy & methods; Test design**
June	**Testing in the new world of DevOps**
September	**Testing SMAC down (social, mobile, analytics & cloud)**
December	**Riding the new development paradigm wave; Trends in test**

We welcome content from seasoned as well as new authors, QA experts, test engineers and anyone who would like to share their knowledge and insights with the wider software test community. Submitted articles may be original works or ones previously posted on blogs, websites or newsletters, as long as you, the author, hold the rights to have such content published by us.

- ***Please see our Detailed editorial calendar and submission guidelines at***
  http://www.logigear.com/magazine/issue/news/2016-editorial-calendar-and-submission-guidelines/

Cruise Control: Automation in Performance Testing

When it comes to performance testing, be smart about what and how you automate

By Tim Hinds

Listen closely to the background hum of any agile shop, and you'll likely hear this ongoing chant: Automate! Automate! Automate! While automation can be incredibly valuable to the agile process, there are some key things to keep in mind when it comes to automated performance testing.

Automated performance testing is important for many different reasons. It allows you to refactor or introduce change and test for acceptance with virtually no manual effort. You can also stay on the lookout for regression defects and test for things that just wouldn't come up manually. Ultimately, automated testing should save time and resources, so you can release code that is bug-free and ready for real-world use.

Recently, I spoke with performance specialist Brad Stoner about how to fit performance testing into agile development cycles. This week, we'll use this blog post to follow up with greater detail around performance testing automation and recap which performance tests are good candidates for automation. After all, automation is an important technique for any modern performance engineer to master.

Automation Without Direction

Most of the time, automation gets set up without performance testing in mind. Performance testing is, at best, an afterthought to the automation process. That leaves you, as a performance engineer, stuck with some pretty tricky scenarios. Maybe every test case is a functional use case, and if you want to adapt them for performance, you have to go back and modify them for scale or high concurrency. Or perhaps the data required for a large performance test is never put together leaving you with a whole new pile of work to do.

Use cases are strung together in an uncoordinated way, so you have to create another document that describes how to use existing functional tests to conduct a load test. And of course, those test cases are stuck on "the happy path" making sure functionality works properly, so they don't test edge cases or stress cases, and therefore, don't identify performance defects.

None of these scenarios is desirable, but they can be easily rectified by incorporating performance objectives into your automation strategy from the start. You want to plan your approach to automation intelligently.

What Automation Is – And Isn't – Good For

You can't automate everything all the time. If you run daily builds, you can't do a massive load test every night. That idea would be even worse if you build several times a day. Instead,

you'll have to pick and choose your test cases, mapping out what you do over periods of time in coordination with the release cycle for the app.

Too many use cases to cover at a time will kill your environment. Constantly high traffic patterns are next to impossible to maintain. Highly specific test scenarios can also cause difficulty because you may need to adjust performance tests every time something changes. That's why it pays to be smart about what you automate.

Look for a manageable number of tests that can be run generically and regularly. Then, benchmark those tests. After that, you can focus your manual time on ad hoc testing, bottlenecks, or areas under active development. This isolation will catch a ton of defects before production.

Get Automation Working for You

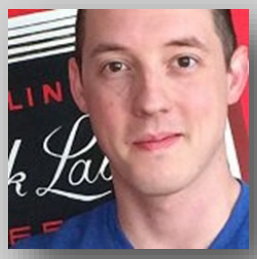
Automation can be great, but it has to identify performance defects and alert you. Just like functional tests validate a defined plan of how an application should behave, performance tests should validate your application's service level agreement. Define the tests in which you want to leverage automation. Is it for workload capacity? Or are you looking for stress, duration, and soak tests? Will you automate to find defects on the front end?

It's easy to automate these problems, and you can do it at a low cost. You'll want to establish benchmarks and baselines often to see if performance degrades as applications are further developed. Testing with direction means that you don't test just for the sake of testing. You always test with a purpose and motive: to find and isolate performance defects. This is a critical thing to do as a performance engineer because you're always dealing with pushing the envelope of the application. You need to know where that boundary lies.

Get Ready for Smooth Sailing

Automated performance testing can be a huge time saver. To make the most of that time-saving potential, you want to do it right. Work smart by always testing with purpose. Ready to dive even deeper into these topics? Jump right in and check out the full webcast here where we go into greater detail about automation strategies. You can also learn how Neotys can help you with the overall agile performance testing cycle.■

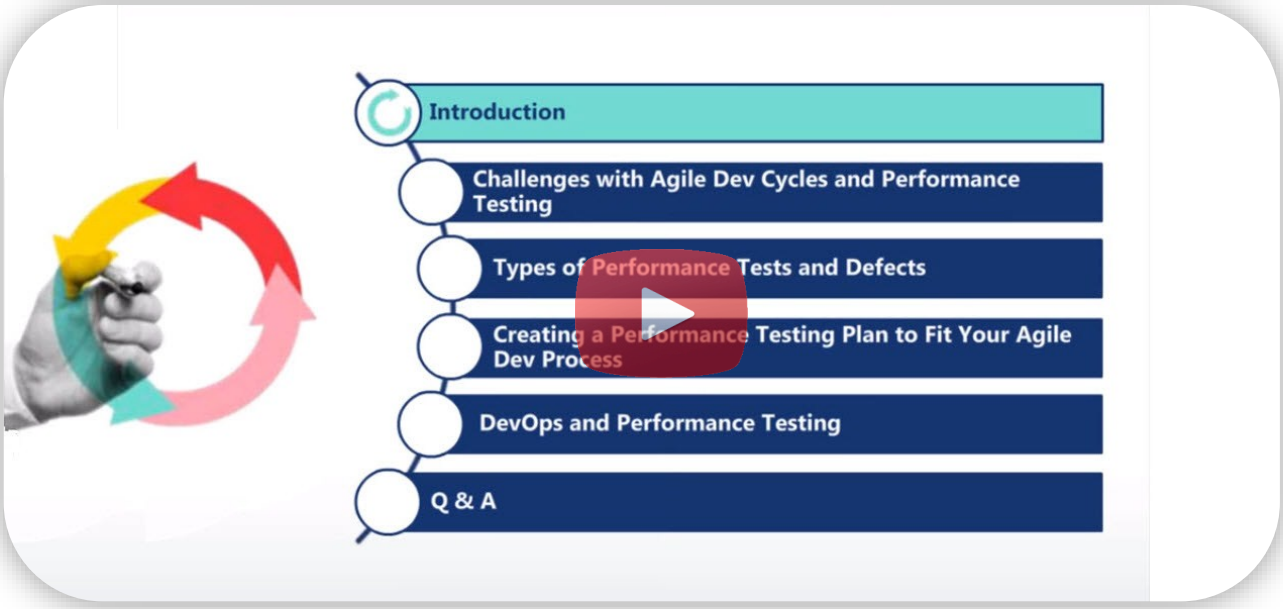
This article originally appeared in NEOTYS BLOG .



Tim Hinds is the Product Marketing Manager for NeoLoad at Neotys. He has a background in Agile software development, Scrum, Kanban, Continuous Integration, Continuous Delivery, and Continuous Testing practices.

For more on this, check out Tim Hinds' and Brad Stoner's webinar,

How to Fit Performance Testing into Agile Dev Cycles:



Pitfalls of Testing Automation Efforts

Top 3 Reasons Why Firms Fail

Lack of Planning

Fragile Tests

Lack of Training

According to Forrester Research's Biannual Global Survey on Agile Software Development...

...Half of Agile teams that automate do not succeed in automating more than 29% of their tests

Risks associated with Automation Failure

Important releases delayed due to testing not keeping up with development

Risking a poor user experience from lack of testing

Spending too much on Manual Testing

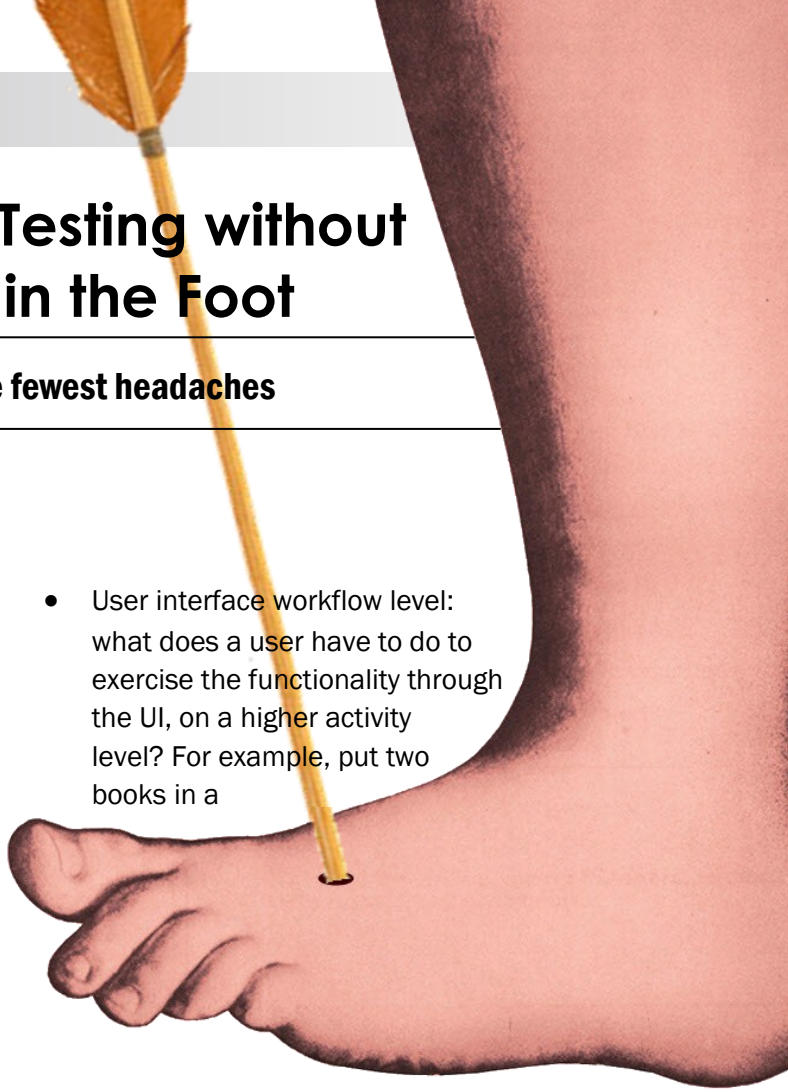
LogiGear

www.logigear.com

How to Implement UI Testing without Shooting Yourself in the Foot

How to do UI test automation with the fewest headaches

By Gojko Adzic



I'm currently interviewing lots of teams that have implemented acceptance testing for my new book. A majority of those interviewed so far have at some point shot themselves in the foot with UI test automation. After speaking to several people who are about to do exactly that at the Agile Acceptance Testing Days in Belgium a few weeks ago, I'd like to present what I consider a very good practice for how to do UI test automation efficiently.

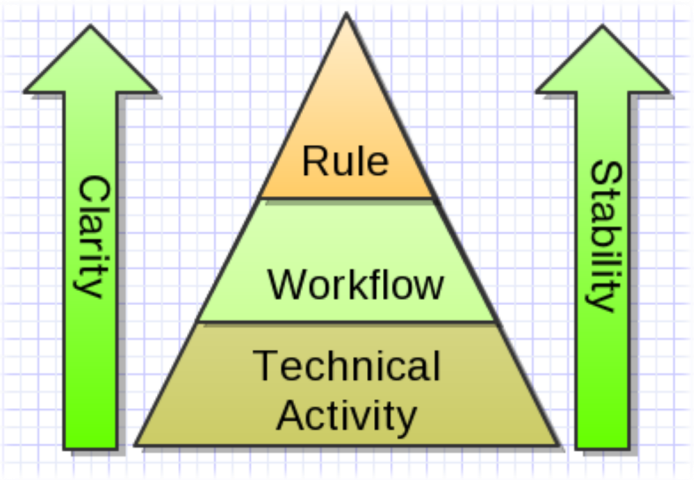
I've written against UI test automation several times so far, so I won't repeat myself. However, many teams I interviewed seem to prefer UI level automation, or think that such level of testing is necessary to prove the required business functionality. Almost all of them have realized six to nine months after starting this effort that the cost of maintaining UI level tests is higher than the benefit they bring. Many have thrown away the tests at that point and effectively lost all the effort they put into them. If you have to do UI test automation (which I'd challenge in the first place), here is how do go about doing it so that the cost of maintenance doesn't kill you later.

Three levels of UI test automation

A very good idea when designing UI level functional tests is to think about describing the test and the automation at these three levels:

- Business rule/functionality level: what is this test demonstrating or exercising? For example: Free delivery is offered to customers who order two or more books.
- User interface workflow level: what does a user have to do to exercise the functionality through the UI, on a higher activity level? For example, put two books in a shopping cart, enter address details, verify that delivery options include free delivery.
- Technical activity level: what are the technical steps required to exercise the functionality? For example, open the shop homepage, log in with "testuser" and "testpassword", go to the "/book" page, click on the first image with the "book" CSS class, wait for page to load, click on the "Buy now" link... and so on.

At the point where they figured out that UI testing is not paying off, most teams I interviewed were describing tests at the technical level only (an extreme case of this are recorded test scripts, where even the third level isn't human readable). Such tests are very brittle, and many of them tend to break with even the smallest change in the UI. The third level is quite verbose as well, so it is often hard to understand what is broken when a test



fails. Some teams were describing tests at the workflow level, which was a bit more stable. These tests weren't bound to a particular layout, but they were bound to user interface implementation. When the page workflow changes, or when the underlying technology changes, such tests break.

Before anyone starts writing an angry comment about the technical level being the only thing that works, I want to say: *Yes, we do need the third level.* It is where the automation really happens and where the test exercises our web site. But there are serious benefits to not having *only the third level*.

The stability in acceptance tests comes from the fact that business rules don't change as much as technical implementations. Technology moves much faster than business. The closer your acceptance tests are to the business rules, the more stable they are. Note that this doesn't necessarily mean that these tests won't be executed through the user interface – just that they are defined in a way that is not bound to a particular user interface.

The idea of thinking about these different levels is good because it allows us to write UI-level tests that are easy to understand, efficient to write and relatively inexpensive to maintain. This is because there is a natural hierarchy of concepts on these three levels. Checking that delivery is available for two books involves putting a book in a shopping cart. Putting a book in a shopping cart involves a sequence of technical steps. Entering address details does as well. Breaking things down like that and combining lower level concepts into higher level concepts has the benefit of reducing the cognitive load and promoting reuse.

Easy to understand

From the bottom up, the clarity of the test increases. At the technical activity level, tests are very technical and full of clutter – it's hard to see the forest for the trees. At the user interface workflow level, tests describe how something is done, which is easier to understand but still has too much detail to efficiently describe several possibilities. At the business rule level, the intention of the test is described in a relatively terse form. We can use that level to effectively communicate all different possibilities in important example cases. It is much more efficient to give another example as "Free delivery is not offered to customers who have one book" than to talk about logging in, putting only a single book in a cart, checking out, etc. I'm not even going to mention how much cognitive overload a description of that same thing would require if we were to talk about clicking check boxes and links.

Efficient to write

From the bottom up, the technical level of tests decreases. At the technical activity level, you need people who understand the design of a system, HTTP calls, DOM and such to write the test. To write tests at the user interface workflow level, you only need to understand the web site workflow. At the business rule level, you need to understand what the business rule is. Given a set of third-level components (e.g., login, adding a book), testers who are not automation specialists and business users can happily write the definition of second level steps. This allows them to engage more efficiently during development and

reduce the automation load on developers.

More importantly, the business rule and the workflow level can be written before the UI is actually there. Tests at these levels can be written before the development starts, and be used as guidelines for development and as acceptance criteria to verify the output.

Relatively inexpensive to maintain

The business rule level isn't tied to any particular web site design or activity flow, so it remains stable and unchanged during most web user interface changes, be it layout or workflow improvements. The user interface workflow level is tied to the activity workflow, so when the flow for a particular action changes we need to rewrite only that action. The technical level is tied to the layout of the pages, so when the layout changes we need to rewrite or re-record only the implementation of particular second-level steps affected by that (without changing the description of the test at the business or the workflow level).

To continue with the free delivery example from above, if the login form was suddenly changed not to have a button but an image, we only need to re-write the "login" action at the technical level. From my experience, it is the technical level where changes happen most frequently – layout, not the activity workflow. So by breaking up the implementation into this hierarchy, we're creating several layers of insulation and are limiting the propagation of changes. This reduces the cost of maintenance significantly.

Implementing this in practice

There are many good ways to implement this idea in practice. Most test automation tools provide one or two levels of indirection that can be used for this. In fact, this is why I think Cucumber found such a sweet spot for browser-based user interface testing. With Cucumber, step definitions implemented in a programming language naturally sit with developers and this is where the technical activity level UI can be described. These step definitions can then be reused to create scenarios (user interface workflow level), and scenario outlines can be used to efficiently describe tests at the business rule level.

The SLIM test runner for FitNesse provides similar levels of isolation. The bottom fixture layer sits naturally with the technical activity level. Scenario definitions can be used to describe workflows at the activity level. Scenario tables then present a nice, concise view at the business rule level.

Robot Framework uses "keywords" to describe tests, and allows us to define keywords either directly in code (which becomes the technical level) or by combining existing keywords (which becomes the workflow and business rule level).

The Page Object idea from Selenium and WebDriver is a good start, but stops short of finishing the job. It requires us to encapsulate the technical activity level into higher level "page" functionality. These can then be used to describe business workflows. It lacks the consolidation of workflows into the top business rule level — so make sure to create this level yourself in

the code. (Antony Marcano also raised a valid point that users think about business activities, not page functionality during CITCON Europe 09, so page objects might not be the best way to go anyway).

TextTest works with xUseCase recorders, an interesting twist on this concept that allows you to record the technical level of step definitions without having to program it manually. This might be interesting for thick-client UIs where automation scripts are not as developed as in the web browser space.

With Twist, you can record the technical level and it will create fixture definitions for you. Instead of using that directly in the test, you can use "abstract concepts" to combine steps into workflow activities and then use that for business level testing. Or you can add fixture methods to produce workflow activities in code.

Beware of programming in text

Looking at UI tests at these three levels is, I think, generally a good practice. Responsibility for automation at the user interface level is something that each team needs to decide depending on their circumstances.

Implementing the workflow level in plain text test scripts (Robot Framework higher level keywords, Twist abstract concepts, SLIM scenario tables) allows business people and testers who aren't automation specialists to write and maintain them. For some teams, this is a nice benefit because developers can then focus on other things and testers can engage earlier. That does mean, however, that there is no automated refactoring, syntax checking or anything like that at the user interface automation level.

Implementing the workflow level in code enables better integration and reuse, also giving you the possibility of implementing things below the UI when that is easier, without disrupting the higher level descriptions. It does, however, require people with programming knowledge to automate that level.

An interesting approach that one team I interviewed had is to train testers to write code enough to be able to implement the user activity level in code as well. This doesn't require advanced programming knowledge, and developers are there anyway to help if someone gets stuck.

Things to remember

To avoid shooting yourself in the foot with UI tests, remember these things:

- Think about UI test automation at three levels: business rules, user interface workflow and technical activity
- Even if the user interface workflow automation gets implemented in plain text, make sure to put one level of abstraction above it and describe business rules directly. Don't describe rules as workflows (unless they genuinely deal with workflow decisions – and even then it's often good to describe individual decisions as state machines).
- Even if the user interface workflow automation gets implemented in code, make sure to sequester technical activities required to fulfil a step into a separate layer. Reuse these step definitions to get stability and easy maintenance later.
- Beware of programming in plain text. ■

This article originally appeared in the author's blog, gojko.net.

Gojko Adzic's latest book is *Fifty Quick Ideas to Improve your Tests*.

Please note that readers of LogiGear Magazine are entitled to a 50% discount on the Ebook version of this book when they use the LogiGear discount code through March 31, 2016.

[Editor's note: A review of *Fifty Quick Ideas to Improve your Tests* may be found elsewhere in this edition of LogiGear Magazine.]

Is your automation more "virtual" than reality?

There's a better way to automate

TestArchitect™

works with Visual Studio too!

Speed test creation

Keyword-driven test authoring and pre-programmed keyword actions eliminate coding and increase productivity.

Increase testing

Multi-application architecture lets you test multiple versions of browsers and applications, so you can scale testing easily.

Reduce maintenance

Keyword actions and test modules minimize the effort needed to update tests when the application changes.

Visual Studio Microsoft Partner

Get a 30-day trial at **testarchitect.com**
Or call 800.322.0333

LogiGear

Get Automated Testing "Done"

How to fit automated testing into scrum, and keep testers in sync with other teams

By Hans Buwalda and Subu Baskaran

One of the benefits of the approaches of agile projects is their friendliness towards testing. The testing activities, and the testers with it, are integrated into the teams, and testing and quality are redefined as team responsibilities. Automation nowadays is a must-have that needs to be addressed. Automation happens on multiple levels in a system, starting with unit tests. Here, we'll focus on functional tests at the UI level.

For functional tests timely automation can be difficult, due to UI dependency. A team might be "done" with work items in a sprint, but the development and automation of functional tests might not have been finished yet. Having to do such automation later is unattractive; it places the testing and automation out of sync with the other activities in the team, making it harder to cooperate.

A first step to make testing manageable in short sprints is to use a domain language approach. This can help the whole team express and communicate tests quickly. We use keyword based "actions" that are easy to manage, use and implement, and is supported well with our product TestArchitect™. We also have a tool to translate actions to and from Behavior Driven Development (BDD) scenarios, another domain language approach.

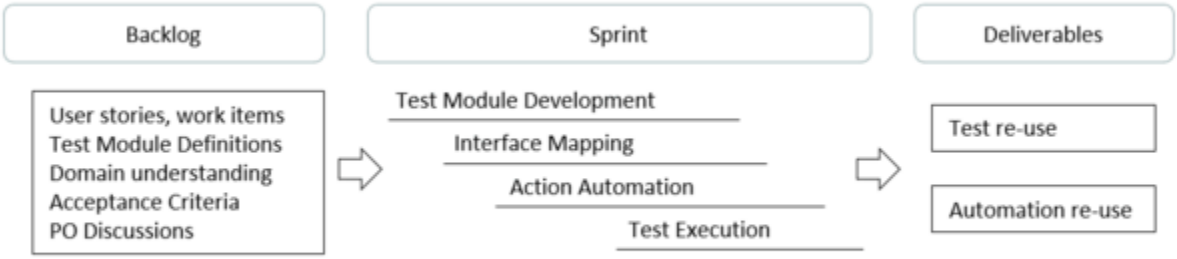
Actions become the basis of the modular test design method called Action Based Testing (ABT), which organizes the tests into "test modules". A main distinction is made in ABT between modules for "business tests" and for "interaction tests". In a business test one would use business level actions like "rent car" or "check balance", while in an interaction test the actions would be at a lower level, like "select menu item" or "check window exists".

To get to automated testing "done" in agile sprints we ended up with a process that is shown in the picture. When the sprint starts the testers create the higher business level tests. These tests stay at the same level as the user stories and acceptance criteria. Further into the sprint, interaction tests are developed, when the user interfaces have become stable enough to make it worthwhile.

Also very early in the sprint interface mappings are made. We recommend that those are created without the use of an interface viewer or other spy tool. This encourages the team to define easy to maintain identifying properties for upcoming UI elements, like the "name" property in Java. This also encourages better collaboration between QA and Dev teams.

When the UI becomes available, the team is ready with:

- Test Modules
  - ⇒ Business Level Tests
  - ⇒ Interaction Tests
- Interface mappings
- Actions – At this point actions are well thought out and finalized but not yet automated



The automation focuses on the actions and only happens when the UIs become fairly final. (If you follow the approach, last-minute changes will be accommodated easily).

An additional way to relieve teams and keep automated testing in sync with development is something we like to call "Outsourcing 2.0", which is part of our services offering as a company. Under this model, excess workload for test development and automation is handed over to a service group that can grow and shrink over time, and can service multiple agile teams. This way the attention of all team members can be focused on new functionalities.■

Hans Buwalda, CTO of LogiGear, is a pioneer of the Action Based and Soap Opera methodologies of testing and automation, and lead developer of TestArchitect, LogiGear's keyword-based toolset for software test design, automation and management. He is coauthor of *Integrated Test Design and Automation*, and a frequent speaker at test conferences.

Subu Baskaran has over 11 years of experience in leading large, complex outsourcing projects both from onsite and offshore. He has a Bachelor's degree in electronics and instrumentation engineering from Sastra University, India, and an MBA from Cass Business School, London.

Further Reading

A selection of articles from past issues of LogiGear Magazine, dealing with issues of test automation and Agile

Principles for Agile Test Automation
Emily Bache — *July 2013*

Quantify the Impact of Agile App Development
Larry Maccherone — *July 2013*

Testing Tools: It ain't Only About Automation!
Michael Hackett — *April 2014*

Is Your Cloud Project Ready to be Agile?
David Taber — *July 2013*

Avoid Epic Fail: Get Professional Help
Michael Hackett & Joe Luthy — *April 2014*

Automation Frameworks & How to Build a Simple One
Karthik KK — *April 2013*

Implementing Automated Software Testing
(Book review) *Jim Holmes* — *April 2014*

Technical Debt: A Nightmare for Testers
Michael Hackett — *July 2013*

A Practical Guide for Testers and Agile Teams
(Book review) *John Turner* — *July 2013*

Misconceptions About Test Automation
Hans Buwalda — *April 2013*

Automation Selection Criteria – Picking the "Right" Candidates
Robert Galen — *April 2014*

Testing Netflix on Android
Amol Kher — *April 2013*

MIND MAP

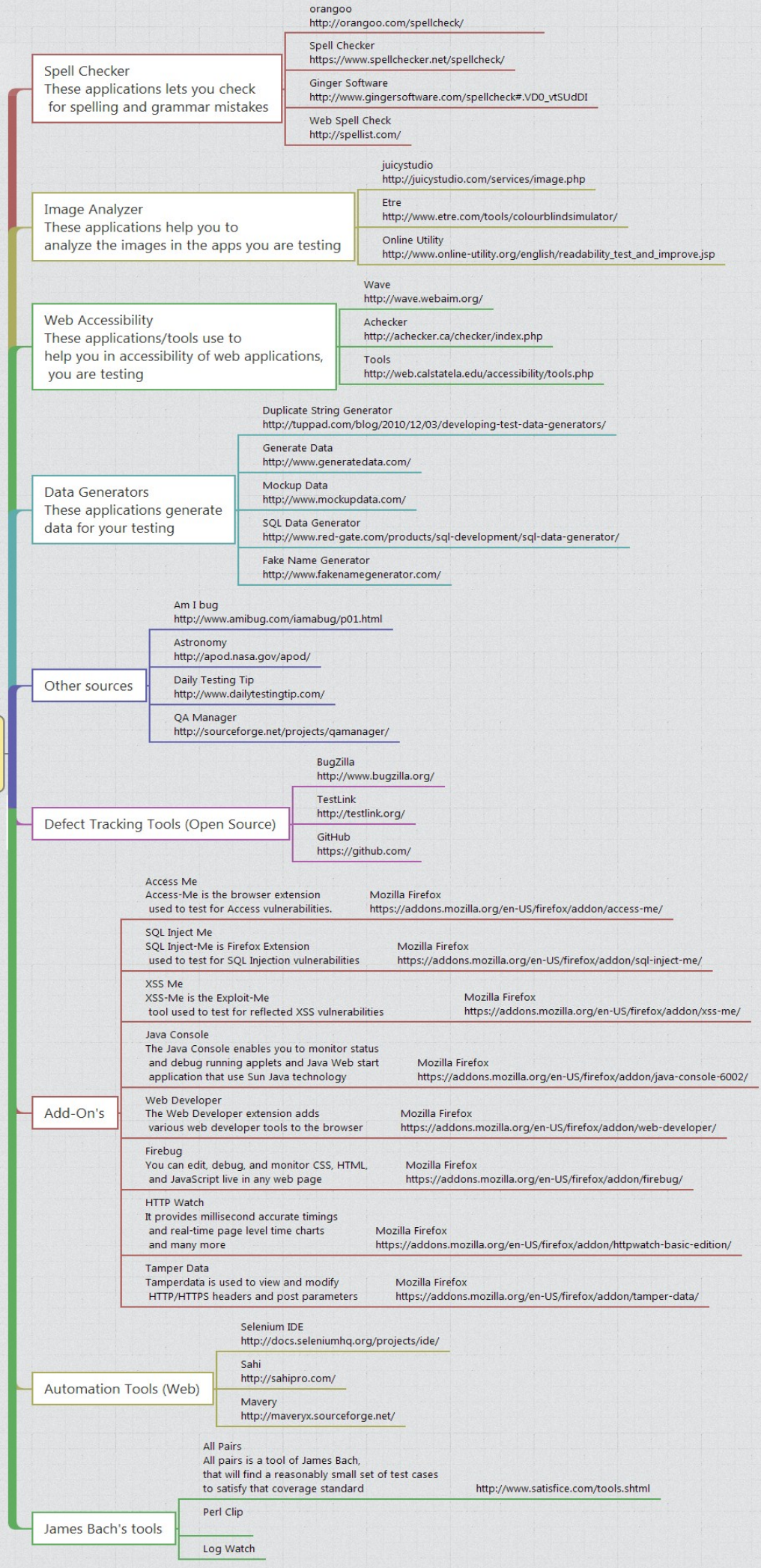
With this edition of LogiGear Magazine, we introduce a new feature, Mind Map.

A mind map is a diagram, usually devoted to a single concept, used to visually organize related information, often in a hierarchical or interconnected, web-like fashion.

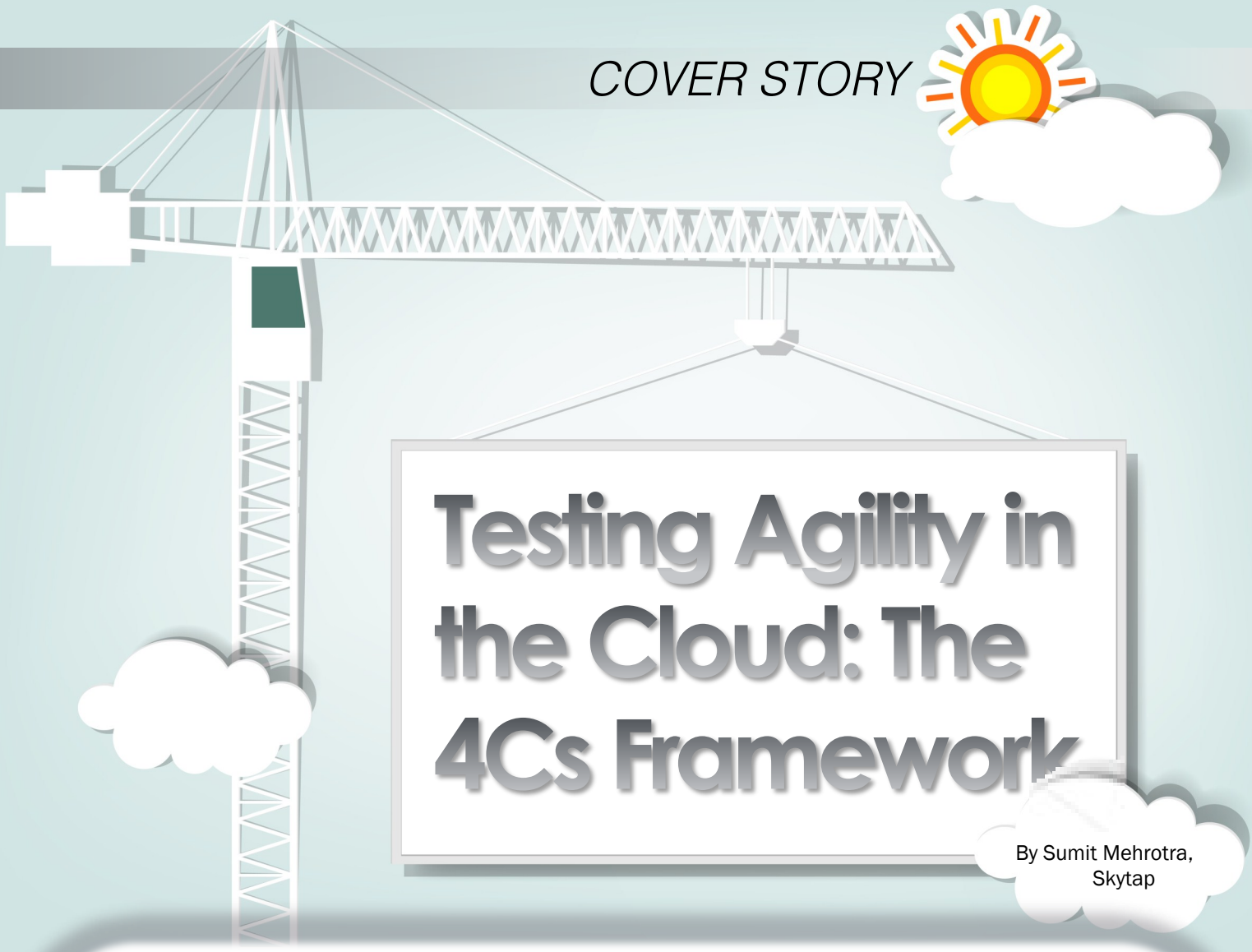
This edition's mind map, created by Sudhamshu Rao, focuses on tools that are available to help you in your testing.

The original map, as well as its downloadable source code, is available from TestInsane.com.

Tools which can help you test better



Sudhamshu Rao is a top notch tester and an active participant of Weekend Testing. He has won testing competitions several times at 99Tests.



Testing Agility in the Cloud: The 4Cs Framework

By Sumit Mehrotra, Skytap

Application development and delivery teams are under constant pressure to release quality features as quickly as possible. CIOs rate delivering applications faster, with higher quality and with strong control on application development as their key priorities. What's more, supporting this type of agile environment is particularly complex to IT teams that are also tasked with supporting multiple, older versions of applications.

Moving faster, with higher quality and stronger control on costs is a common mantra in enterprise application development and delivery (AD&D) teams today. However, these requirements often pull teams in different directions. To release things faster, teams often skip various pieces of testing to compress the timelines that result in costly customer issues later. And conversely, to achieve required quality, teams often have to sacrifice features, thereby impacting business deliverables. Lastly, in order to achieve business deliverables *with* the desired quality, teams tend to be forced to spend a lot, both in resources and people.

To avoid being forced to sacrifice quality for speed, and vice versa, I recommend a 4Cs framework. This framework eliminates common constraints faced by AD&D teams looking to adopt DevOps practices like continuous integration and continuous delivery, and helps deliver agility in the cloud. Many enterprises today are adopting this framework to help them evaluate the variety of tools and resources in the ecosystem to help them deliver business value faster, with higher quality and lower costs.

In this article, we introduce the 4Cs framework, and use it in the context of four transformations — each addressing a given set of **problems**, with an appropriate array of **tools** for a **desired end result** — that teams are trying to achieve in their software delivery pipelines.

The 4Cs Framework for the Software Delivery Lifecycle

The 4Cs framework is a set of simple questions that teams should ask when evaluating which tools to implement in an agile software development lifecycle (SDLC) in order to achieve faster releases, with higher quality and optimal costs. The 4Cs are:

- **Configurability**
- **Consistency**
- **Collaboration**
- **Control**

Configurability

The key question to ask here is:

*Can I have test **environments** that capture the **complexity** of my **application** at each **stage** of testing?*

environments: We are talking about multiple development and test environments that are needed across the SDLC by various teams.

complexity: We need to capture the complexity of the entire application. This includes:

- Topology of the application, i.e. multiple networks, VPN connections, open ports, etc.
- The scale of the application, i.e. size of VMs being used in RAM, CPU, Storage, the number of VMs, etc.
- The platforms and components being used, i.e. OSs, middleware, databases, appliances, etc.

application: An enterprise application consists of multiple components, even products, delivered by different product teams.

stage: Each stage of testing entails the testing of different application components by different teams at different levels (functional, systems, integration, performance, etc.).

Consistency

The key question to ask here is:

*Can I depend on my test environments to be in the **exact state** that I need them to be, and **whenever** I want?*

exact state: Consistent test results require testing an application in a known state. This includes not just the infrastructure topology to be in the desired state but also the application (OS and up) to be configured correctly.

whenever: Being able to test continuously and also as needed — based on priorities — is key to achieving continuous integration/continuous delivery and DevOps workflows.

Collaboration

The key question here is:

*Can I make it easier for my team (devs + qa + ops) to **work together** more productively?*

work together: Feedback loops are important for DevOps. Finding bugs, reproducing them quickly, fixing them and verifying them happens continuously in the SDLC. The shorter and faster these loops are, the more agility there is in the SDLC.

Control

The key question here is:

*Can I **ensure** the **right people** have the **appropriate resources** to do their jobs?*

ensure: Being in control of providing resources for AD&D throughout the SDLC while still servicing the needs of various development and test teams in a self-service and agile fashion.

right people: Being able to secure access to the resources, so that that only teams/users that need access to a set of resources have access to them. This means being able to secure your enterprise resources from the outside world.

appropriate resources: Being able use the resources in the most optimal way— keeping in mind the needs of the AD&D teams and the budgetary constraints in the organization. Being able to proactively monitor the usage of resources, and reactively being able to report on the efficacy and ROI of resources used.

Transformations to the SDLC

The picture below depicts a typical SDLC pipeline.

There are 4 stages in this pipeline (NOTE: this is just an example, there may be more stages in your own pipeline).

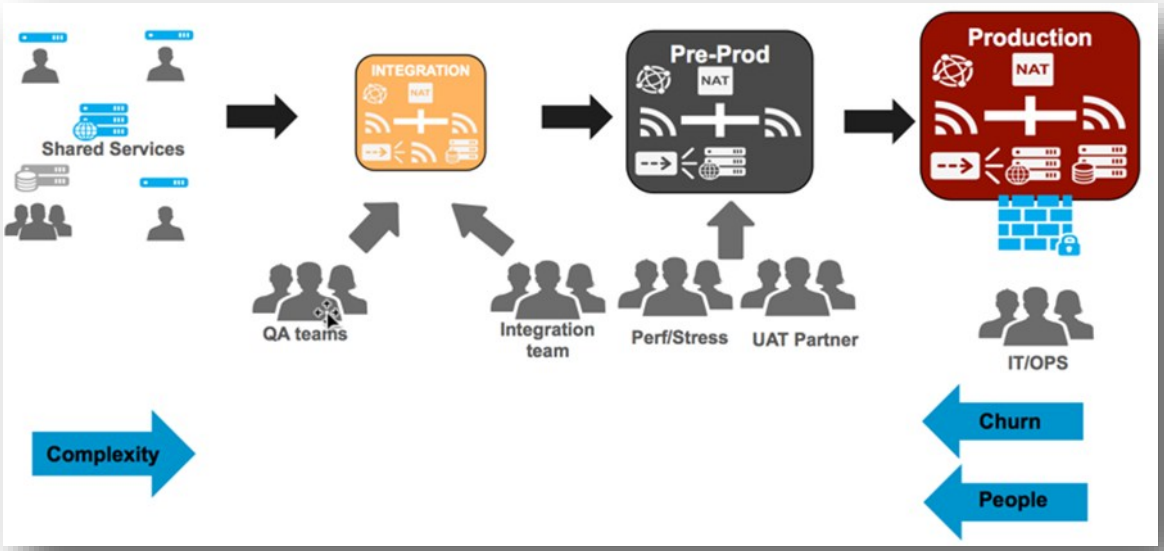
- Development stage: Developers and testers are working on their individual features and automation tools. Developers from various product teams are checking code into their feature branches. There is some unit testing happening. QA teams for a feature are performing functional testing at a feature level. Multiple teams may be using a set of shared services as well, comprising centralized services like source control systems, build services and centralized databases.
- Integration stage: Code from multiple features gets integrated and larger scale integration testing is conducted to ensure the quality of the entire product. There may be multiple QA teams working on various aspects of quality of the product in this stage.
- Pre-Production stage: This is the last stage before releasing the product to customers. Typically, the most complex testing is done in production-like environments, hoping to weed out complex bugs that can only be found when testing at production scale.
- Production: The final stage, when the product is deployed for customers. Services are managed by an IT/Ops team, and release teams manage a boxed product.

As we move from left to right in this pipeline, the following changes are observed:

- Complexity: Increases from left to right. Complexity of the application applies to both the topology and the application configuration. As more and more features come together and as more intensive tests are performed, complexity increases.
- Churn: Decreases from left to right. More code is added more frequently towards the left. More bugs are found and fixed towards the left. This results in lots of churn in the application.
- People: Decrease from left to right. There are more developers and testers touching the application code towards the left. By contrast, on the other side in production, the goal is to have as few people touching the application code and configuration as possible.

With this context, let's take a look at a set of transformations targeted for areas of this SDLC pipeline that are ripe for change. In each transformation we will discuss the problem, introduce the class of tools at our disposal to address the issues, and use the 4Cs framework for evaluation.

It is worth mentioning here that we will go through these transformations in an order, left to right. You may choose to adopt some or all of these transformations, and in a different order than discussed here. We indeed have seen customers take this journey of transformations starting at different points.



Transformation 1: On-demand Test Environments

In this transformation, we focus on the individual dev and test teams on the left.

The **problems** in this stage are:

- Developers are checking in code and doing unit tests on their own machines.
- There is no consistency in test environments used by developers and testers.
- There may be one lab shared between application teams for running functional tests.
- There are lots of delays between test passes, either due to lack of environments or due to environments not being properly configured.
- There is contention between users and teams trying to use test environments that result in lower quality and delays in testing.

The **tools** available for this transformation fall into the following categories:

- Infrastructure Platforms: In addition to common on-premise infrastructure management tools, leading cloud platforms — especially the IaaS portions of those platforms — are well-suited to provision lab infrastructure on demand.
- Configuration Management and Deployment tools: These tools are used to configure software components on top of raw infrastructure and also to deploy application components. Examples of such tools are: Chef, Puppet, Ansible, Salt, UrbanCode, etc.
- Task Management Tools: These tools are used for tracking work items, bugs, etc. An efficient task management system is needed to ensure that quality issues don't fall through the cracks and that the right people are working on the right set of work items at any given time.
- Unit Testing: A robust set of unit tests for each piece of code being checked in is a basic requirement for continuous delivery and DevOps models. There are various unit testing platforms available today, like JUnit, NUnit, Cucumber, etc., that make the task of writing unit tests easy for development teams.

With the application of these tools, the **desired end result** is:

- More testing being done by individual developers and testers in consistent environments
- Lower wait times between test passes
- Bugs found, fixed, and validated faster
- Features getting into the integration stage faster and with a higher level of quality

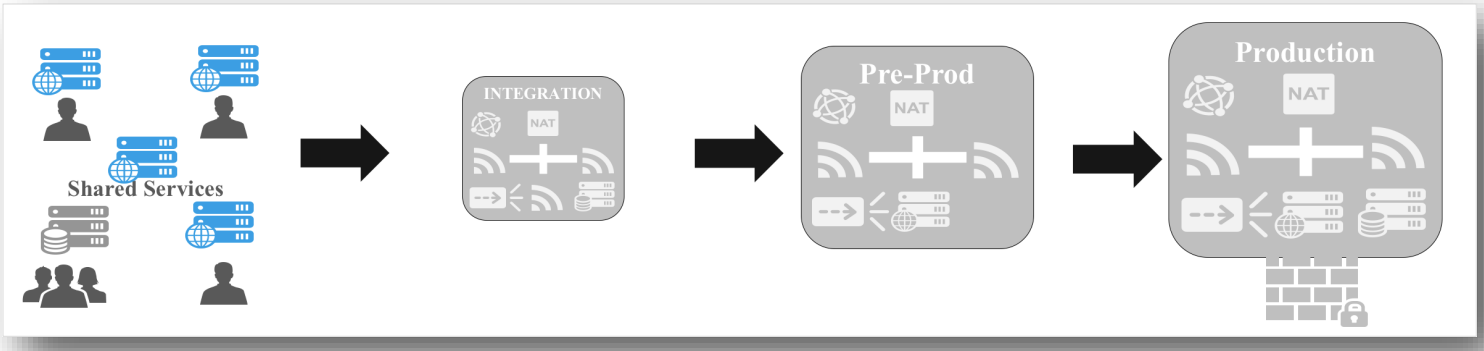
The 4Cs criteria to evaluate the tools to achieve the end result are as follows:

Configurability: Give each team or even an individual developer or tester a complete test environment for their component.

Consistency: Create base environments in the desired state quickly within seconds or minutes. Incremental changes can be applied on top of this known state and testing can be efficiently and consistently conducted.

Collaboration: Ability to easily share one's test environment with other team members to collaborate on testing and bug fixing. Ability to share a set of common services like databases, source control systems, and build servers with other teams and users.

Control: Ability to provide such test environments whenever needed, stow away when not in use and rehydrate quickly in a consistent state, and optimize spending.



Transformation 2: Continuous Integration

For this transformation, we will place our focus on the integration stage.

The **problems** in this stage are:

- Integration environments are typically complex to set up, so there is only one. Teams forgo doing integration tests at the feature level and wait to integrate at a much later stage.
- Integration environment may not run reliably.
- Components integrate infrequently, causing breaks and integration being blocked for several components.
- Bringing the integration environment to a known good state takes a long time, thus increasing the time for feature integration.
- After transformation #1, features are getting into integration faster but hitting a choke point here.

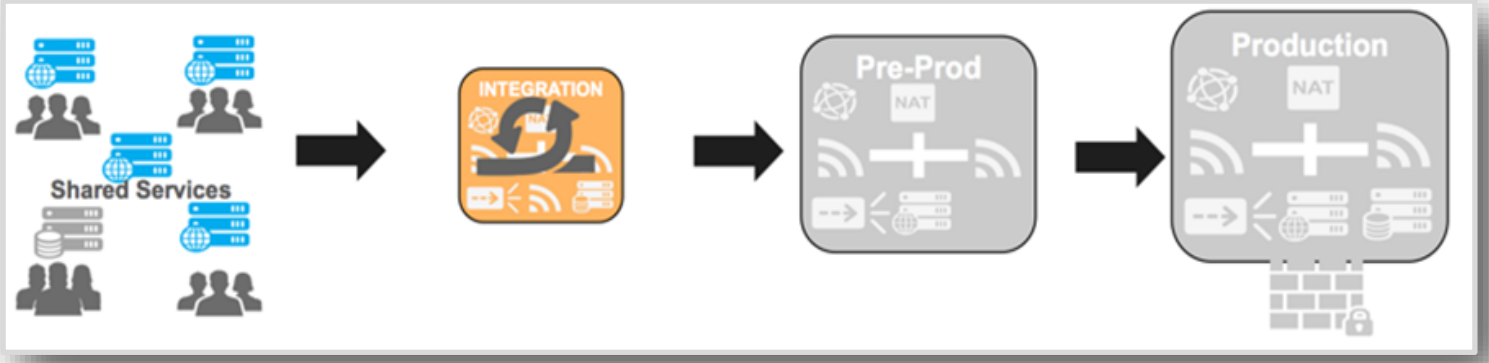
Along with the **tools** discussed in transformation #1, there is an additional class of tools available to set up this transformation:

Continuous Integration tools: These tools are designed as workflow engines to make the task of automated integration testing easier. They can help you set up workflows to track changes in any feature in the product and run a battery of tests, from the simplest to the most complex, as needed, to validate the quality of the change. Tools in this category include Jenkins, Visual Studio, Teamcity, and Bamboo. Recently, there have been a host of new service-based tools in this category — TravisCI, CodeShip, etc. — catering to applications built on the PaaS model.

Static Analysis tools: These are tools that give you 'quality for free'. These tools are designed to weed out bugs just by analyzing the source code without you having to write any test cases. These tools can be easily integrated into continuous integration cycles to improve the quality of the code. Examples of tools in this category are SonarQube, FxCop, Fortify, and Parasoft Static Analysis tools.

The **desired end result** of this transformation is:

- Run automated tests for every check-in, for every feature in a representative test environment. The test passes are usually very fast — generally under a couple of minutes — so the results of each check-in can be communicated quickly.
- Run a more intensive set of integration tests at a periodic interval — e.g., daily — on all changes made to the product since the last run, in a full scale application deployment. These tests usually take hours, if not days.
- Ability to point out the cause of failure, immediately attribute the failure to the right person and communicate enough information (logs, repro, data, etc.) to resolve the issue quickly.
- This results in early and intensive testing of a large portion of the application, mostly in an automated fashion.
- The saving of valuable QA time that can be spent on creative testing areas like exploratory testing.



The 4Cs criteria to evaluate the tools to achieve the end result are as follows:

- **Configurability** – Ability to support a large variety of continuous integration testing workflows. On one end of the spectrum are simple one box CI environments where all application components, at a smaller scale, are deployed on one machine and the integration test suite is deployed on it. On the other end of the spectrum are more complex environments that represent the production-like deployment of the application, including multiple VM, networks, external interfaces, VPN connections, appliances, etc.
- **Consistency** – The need is to complete the integration runs as fast as possible, and ensure that they're of reliable quality. Each integration run should start off as a brand new environment that is configured in the base state required by the test. The latest build is deployed and configured on top of this base state and tests are run.
- **Collaboration** – Results of CI runs should be disseminated to teams quickly, with pointers to builds, results, and test environments, especially in case of failures.
- **Control** – Should be able to consume use-and-throw CI environments and save them off when needed (e.g. failures). Should be able to scale up the resources needed for CI based on business needs and scale them down when required.

Transformation 3: Testing at Production Scale

For this transformation we will focus on the pre-production stage. However, this transformation can be applied anywhere in the pipeline. The earlier this type of testing is done, the better.

The **problems** in this stage are:

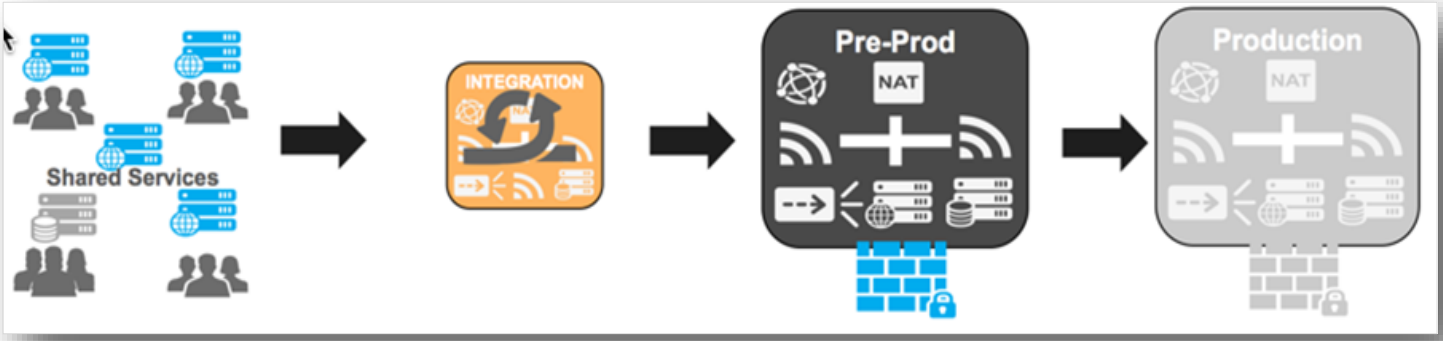
- Teams often have a pre-production environment that is not up to production standards.
- It is hard to build such an environment, given the complexity of the application, the scale and the configuration of the application and data.
- Maintaining such an environment in a consistent state is hard, given that there are intensive tests being run at this stage and there are a number of teams that work together in these environments.
- Product upgrades, even seemingly simple ones like OS upgrades, can sometimes leave the environments in a broken state for a long time, delaying product releases.

Along with the **tools** discussed in Transformations 1 and 2, we should also add the following class of tools to our arsenal:

- **Test Data Management tools**: Should be able to create production-like environments and populate them with data that is at the scale and state of production. This should be an easily repeatable process as well.

The **desired end result** of this transformation is:

- Ability to create production-level environments at any point in the pipeline
- Ability to create production-like environments in a consistent state (application and data) and apply product changes to them
- Run intensive, and even destructive tests in this environment
- Discover hard-to-find bugs before reaching production



The **4Cs criteria** to evaluate the tools to achieve the end result are as follows:

- **Configurability**: Should be able to handle complex networking topologies, access policies, data management, scale of resource requirements, and monitoring.
- **Consistency**: Should be able to ensure that pre-production truly reflects the state of production before each release.
- **Collaboration**: Should be able to share early access to pre-production with internal stakeholders, like feature teams, QA teams, and Ops, as well as external stakeholders like customers, contractors and partners.
- **Control**: Should be able to provide access to pre-production environments when needed and stow away when not needed. Should be able to limit access to certain components to specific users, teams or departments.

Transformation 4: Parallel Testing

This transformation applies across the SDLC pipeline. Parallelism can be introduced at any stage in the pipeline to accelerate the process without compromising quality.

The **problem** we are trying to address is:

- Needs to allow the business to grow over time. This results in an increase in products/features being developed and in the number of people working on them. More components, more people = more features + more check-ins.
- Single-threaded pipeline (single CI environment, single staging/pre-prod environment) does not suffice if the team has to deliver requirements with the same quality in the same time.
- Teams that are geographically spread end up accessing single testing environments that may be remote to them. This creates more inefficiencies in the testing process.

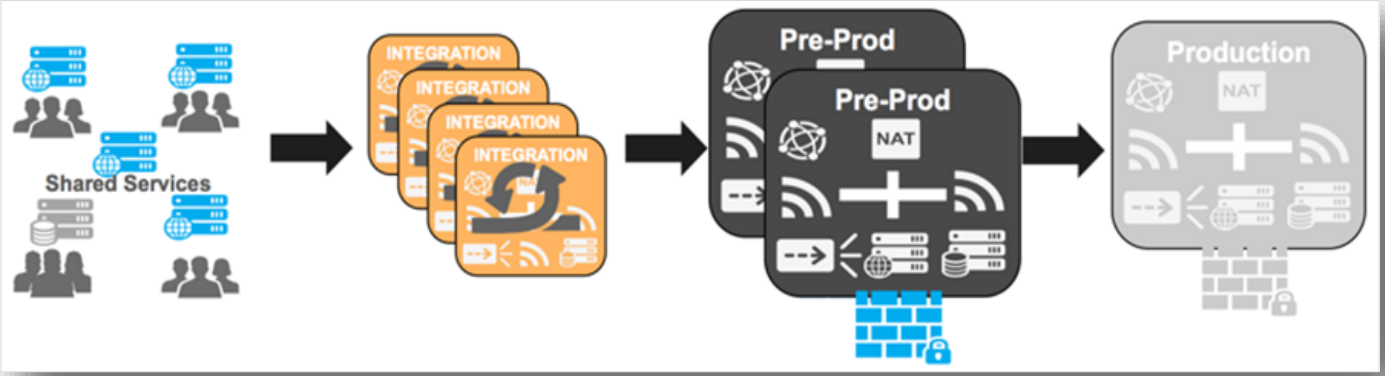
In terms of **tools** for this transformation, we will discuss a set of practices that can be implemented with the class of tools discussed previously:

- Patterns: This deals with using code to create multiple copies of the same application environment. This includes infrastructure-as-code and application configuration-as-code. Combining these produces the complete application stack needed for testing. This code can be run over and over again to create environments.
- Clones: This implies cloning an existing application stack that has been built either manually or with a pattern. The tools being used for cloning usually take care of the cloning process without any special knowledge needed by the user performing the clone.

Both of these practices can be used exclusively or, more effectively, together in different parts of the SDLC, based on teams' needs. Patterns create and validate code that can be propagated throughout the SDLC and can even be used for production (continuous delivery). However, each run can take a long time. Cloning makes the process much faster and easier for end users (devs and testers).

The **desired end result** of this transformation is:

- More feature teams (existing and future) are onboarded quickly and are productive sooner.
- More features make it through the pipeline and/or features take less time in the pipeline.



The 4Cs criteria for evaluation are:

Configurability: Should be able to handle parallel, possibly identical, environments. E.g., complexity in managing network address spaces, application components

Consistency: Should be able to ensure that testing is in environments with a consistent state, especially if the teams are in different geographic locations throughout the world

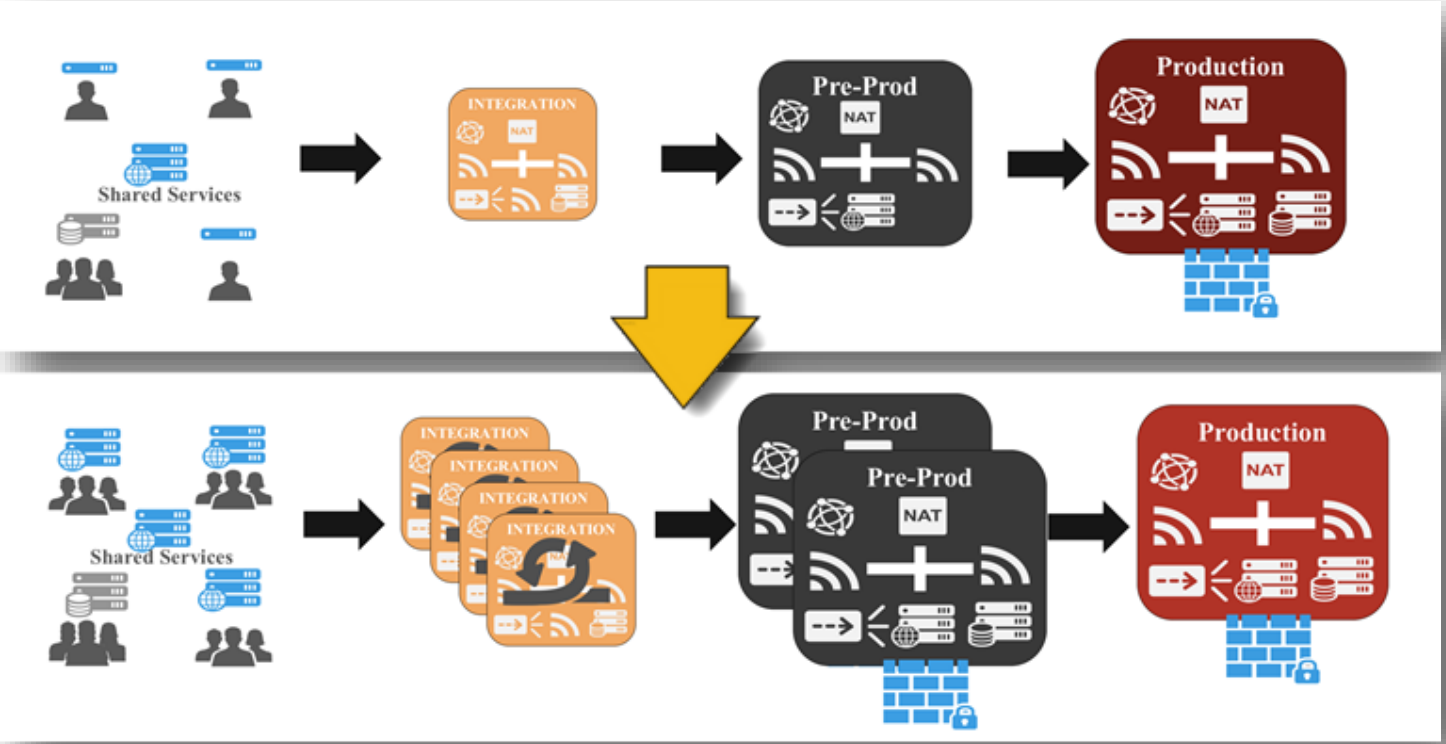
Collaboration: Should be able to handle dependency management of components going through parallel testing environments

Control: Should provide oversight on resource utilization. Increased parallelism increases the expenditure on resources that must be managed judiciously

Transformations at a glance

In summary, we have gone through FOUR transformations to introduce the following changes into a typical SDLC pipeline:

- On-demand self-service test environments
- Continuous integration and continuous quality
- Earlier testing in production-like environments
- Parallel testing



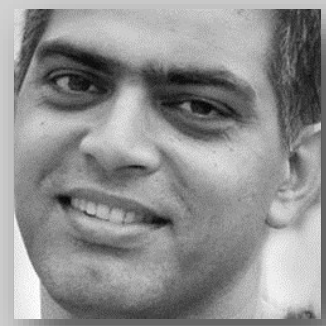
We have typically seen teams start this journey from two points:

- Testing in production-like environments: Enterprise teams typically face much difficulty when testing in production-like circumstances, and they have taken on this problem as the first step toward transforming their SDLC. Once successful with the right set of tools, they quickly graduate to parallelism in these production-like environments. Subsequently, as they become more efficient, they start thinking about breaking up the monolithic application architecture into more modular blocks. With these modular components and modular teams, it becomes easier to equip those teams with on-demand self-service environments, in order to implement continuous integration/delivery practices.

- On-demand, self-service test environments: In this path, teams are usually on a journey to modularize their application more from a dev/test perspective than from an IT/Ops perspective. They are implementing continuous integration/delivery practices earlier in the cycle. Once these practices are honed, they are promoted to stages and teams further on the right of the SDLC. Production-like environments are also included in the mix as the complexity of testing increases.

Each team may take the journey through a different path and with different tools, but the end goal is always faster, higher, stronger.

Enterprises want to produce business results faster with good ROI. A key enabler of that is the speed at which software is delivered, the quality at which it is produced — and the cost incurred. It is important for development and test teams to think about the ways they can transform their software delivery lifecycles to achieve those objectives. There is a large ecosystem of patterns, tools and processes that are available to accomplish that goal. In this paper we talked about four such transformations, and the 4Cs framework for evaluating the tools that can help you achieve those transformations. ■



Sumit Mehrotra is Sr. Director of Product Management at Skytap, a role in which he is responsible for product strategy and roadmaps.

Prior to Skytap, Sumit worked at Microsoft in different roles and has shipped a number of products, including Windows Azure and Windows operating system.

Sumit holds an MBA from University of Chicago Booth School of Business and a Masters in Computer Science from Boston University.

The following terms are either found in the accompanying articles, or are related to concepts relevant to those articles.

Action Based Testing (ABT)

A refinement of the keyword-driven test approach that provides a powerful framework for organizing test design, automation and execution around keywords. In ABT keywords are called "actions". Actions are the tasks that are executed during a test. Rather than automating an entire test as one long script, tests are assembled using individual actions.

Unlike traditional test design, which begins with a written narrative that must be interpreted by each tester or automation engineer, ABT test design takes place in a spreadsheet format called a test module. Actions, test data and any necessary GUI interface information are stored in separate files and referenced by the main test module.

Behavior Driven Development (BDD)

A software development methodology in which an application is specified and designed by describing how its behavior should appear to an outside observer. BDD combines the general techniques and principles of test-driven development (TDD) with ideas from domain-driven design and object-oriented analysis and design to provide software development and management teams with shared tools and a shared process to collaborate on software development.

BDD offers the ability to enlarge the pool of input and feedback to include business stakeholders and end users who may have little software development knowledge. Because of this expanded feedback loop, BDD works well in continuous integration and continuous delivery environments.

Source: SearchSoftwareQuality

Concurrency

Concurrency refers to multiple things happening at the same time. In testing, it's all about the fact that your web application, mobile application, etc., may be required, in a real world setting, to respond to multiple demands occurring in parallel. Load testing is the method by which we test to ensure that an application, and the resources it has to work with, is equipped to handle the level of concurrency that it can expect to find in the field.

Continuous Integration (CI)

A software engineering practice in which the changes made by developers to working copies of code are added to the mainline code base on a frequent basis, and immediately tested. The goal is to provide rapid feedback so that, if a defect is introduced into the mainline, it can be identified quickly and corrected as soon as possible. Continuous integration software tools are often used to automate the testing and build a document trail. Because CI detects deficiencies early on in development, defects are typically smaller, less complex, and easier to resolve. In the end, well-implemented CI reduces the cost of software development and helps speed time to market.

Source: SearchSoftwareQuality

DevOps

(Term derived from the words "Development" and "Operations") A software development practice, grounded in agile philosophy, that emphasizes close collaboration between an organization's software developers and other IT professionals, while automating the process of software delivery and infrastructure changes. It aims at establishing a culture and organizational workflow in which building, testing, and releasing software happens rapidly, frequently, and more reliably.

Source: Wikipedia

Domain language

(Also referred to as domain-specific language) A computer language tailored for a specific application or discipline (domain). In automated testing, for example, the keyword-driven approach, such as that which is implemented by LogiGear's TestArchitect automation tool, allows teams to develop their own customized domain languages. Such languages allow for easier implementation of testing scenarios, and aid in communication between organizational teams.

Refactoring

The process of restructuring existing computer code without changing its external behavior. From a functional standpoint (or at least from the standpoint of satisfying existing specifications) code refactoring should be transparent. Instead, it is the nonfunctional attributes of the software that are improved.

Code refactoring is considered a form of "hygiene", the advantages of which include improved readability and reduced complexity. These in turn can improve source code maintainability and create a more expressive internal architecture or object model to improve extensibility.

If done well, code refactoring may also resolve hidden, dormant, or undiscovered computer bugs or vulnerabilities in the system by simplifying the underlying logic and eliminating unnecessary levels of complexity.

Source: Wikipedia ■

Why is test automation the backbone of Continuous Delivery?

The path to continuous delivery leads through automation

By Ranjan Sakalley

Software testing and verification needs a careful and diligent process of impersonating an end user, trying various usages and input scenarios, comparing and asserting expected behaviours. Directly, the words "careful and diligent" invoke the idea of letting a computer program do the job. Automating certain programmable aspects of your test suite thus can help software delivery massively. In most of the projects that I have worked on, there were aspects of testing which could be automated, and then there were some that couldn't. Nonetheless, my teams could rely heavily on our automation suite when we had one, and expend our energies manually testing aspects of the application we could not cover with automated functional tests. Also, automating tests helped us immensely to meet customer demands for quick changes, and subsequently reaching a stage where every build, even ones with very small changes, went out tested and verified from our stable. As Jez Humble rightly says in his excellent blog about continuous delivery, automated tests "take delivery teams beyond basic continuous integration" and on to the path of continuous delivery. In fact, I believe they are of such paramount importance, that to prepare yourself for continuous delivery, you *must* invest in automation. In this text, I explain why I believe so.

How much does it cost to make one small change to production?

As the complexity of software grows, the amount of effort verifying changes, as well as features already built, grows at least linearly. This means that testing time is directly proportional to the number of test cases needed to verify correctness. Thus, adding new features means that testing either increases the time it takes a team to deliver software from the time development is complete, or it adds cost of delivery if the team adds more testers to cover the increased work (assuming all testing tasks are independent of each other). A lot of teams — and I have worked with some — tackle this by keeping a pool of testers working on "regression" suites throughout the length of a release, determining whether new changes break already built functionality. This is not only costly, its ineffective, slow and error prone.

Automating test scenarios where you can lets you cut the time/money it takes to verify if a user's interaction with the application works as designed. At this point, let us assume that a reasonable number of your test scenarios can be automated — say 50% — as this is often the lowest bound in software projects. If your team can and does automate this set to a certain number of repeatable tests, it frees up people to concentrate more on immediate changes. Also, let's suppose that it takes as much as three hours to run your tests (it should take as little as possible — less than 20 minutes even). This directly impacts the amount of time it takes to push a build out to customers. By increasing the number of automated tests, and also investing in getting the test-run time down, your agility and ability to respond increases massively, while

also reducing the cost. I explain this with some very simple numbers (taking an average case) below:

Team A

1. Number of scenarios to test: 500 and growing.
2. Time to setup environment for a build: 10 minutes.
3. Time to test one scenario: 10 minutes.
4. Number of testers on your team: 5.
5. Assume that there are no blockers.

If you were to have no automated tests, the amount of time it would take to test one single check-in (in minutes), is:

10 + (500*10)/5 = 1010 minutes.

This is close to two working days (standard eight hours each). Not only is this costly, it means that developers get feedback two days later. This kind of a setup further encourages mini-waterfalls in your iteration.

Team B

Same as Team A, but we've automated 50% (250 test cases) of our suite. Also, assume that running these 250 test cases takes a whopping three hours to complete.

Now, the amount of time it would take to test one single check-in (in minutes), is:

task 1 (manual): 10 + (250*10)/5 = 510 minutes.
task 2 (automated): 10 + 180 minutes.

This is close to one working day. This is not ideal, but just to prove the fact about reduced cost, we turned around the build one day earlier. We halved the cost of testing. We also covered 50% of our cases in three hours.

Now to a more ideal and (yet) achievable case:

Team C

Same as Team B, but we threw in some good hardware to run the tests faster (say 20 minutes), and automated a good 80% of our tests (10% cannot be automated and 10% is new functionality).

Now, the amount of time it would take to test one single check-in (in minutes), is:

task 1 (manual): 10 + (100*10)/5 = 210 minutes.
task 2 (automated): 10 + 20 minutes = 30 minutes.

So in effect, we cover 80% of our tests in 30 minutes, and overall take 3.5 hours to turn around a build. Moreover, we've

increased the probability of finding a blocker earlier (by covering the vast bulk of our cases in 30 minutes), meaning that we can suspend further manual testing if we need to. Our costs are lower, we get feedback faster. This changes the game quite a bit, doesn't it?

Impossibility of verification on time

Team A that I mentioned above would need 50 testers to certify a build in under two hours. That cost is, not surprisingly, unattractive to customers. In most cases, without automation, it is almost impossible to turn around a build from development to delivery within a day. I say *almost* impossible, as this would prove to be extremely costly in cases where it is. So, assuming that my team doesn't automate and hasn't got an infinite amount of money, every time a developer on the team checks in one line of code, our time to verify a build completely increases by hours and days. This discourages a manager from scheduling running these tests every time on every build, which consequently decreases the quality of coverage for builds, and ups the amount of time bugs stay in the system. It also, in some cases I have experienced, dis-incentivizes frequent checking in of code, which is not healthy.

Early and often feedback

One of the most important aspects of automation is the quick feedback that a team gets from a build process. Every check-in is tested without prejudice, and the team gets a report card as soon as it can. Getting quicker feedback means that less code gets built on top of buggy code, which in turn increases the credibility of the software. To extend the example of teams A, B

and C above:

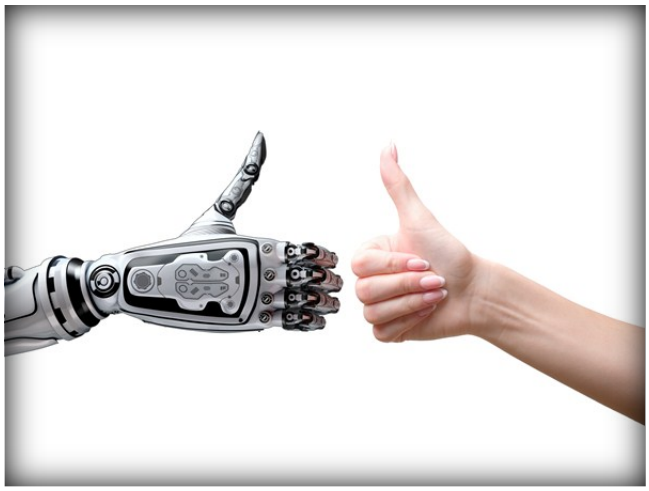
For Team A: The probability of finding a blocker on day one is 1/2. Which basically means that there is a good risk of finding a bug on the second day of testing, which completely lays the first days of work to waste. That blocker would need to be fixed, and all the tests re-verified. In the worst case, a bug is found after two days of an inclement line of code getting checked-in.

For Team B: The worst case is that you find a blocker in the last few hours of the day. This is still much better than for Team A. Better still, as 50% of test cases are automated, the chance of finding a blocker within three hours is very high (50%). This quick feedback lets you find and fix issues faster, and therefore respond to customer requests very quickly.

For Team C: The best case of all three. The worst-case scenario is that Team C will know after three hours if they checked-in a blocker. As 80% of test cases are automated, by 20 minutes, they would know that they made a mistake. They have come a long way from where Team A is — 20 minutes is way better than two days!

Opportunity cost

Economists use an apt term – opportunity cost – to define what is lost if one choice amongst many is taken. The opportunity cost of re-verifying tedious test cases build after build is the loss of time spent on exploratory testing. More often than not, a bug leads to many, but by concentrating on manual scenarios, and while catching up to do so, testers hardly find any time to create new scenarios and follow up on issues. Not only this, it is a given that by concentrating on regression tests all the time, testers spend proportionately less time on newer features, where there is a higher probability of bugs to be found. By automating as much as possible, a team can free up testers to be more creative and explore an application from the "human angle" and thus increase the depth of coverage and quality. On projects I have worked on, whenever we have had automated tests aiding manual testing, I have noticed better and more in-depth testing which, has resulted in better quality.



Another disadvantage to manual testing is that it involves tedious re-verification of the same cases day after day. Even if managers are creative and distribute tests to different people every day, the cycle inadvertently repeats after a short period of time. Testers have less time to be creative, and therefore their jobs less gratifying. Testers are creative beings and their forte is to act as end-users and find new ways to test and break an application, not in repeating a set process time after time. Without automation, the opportunity cost in terms of keeping and satisfying the best testers around is enormous.

Error prone human behavior

Believe it or not, even the best of us are prone to making mistakes doing our day to day jobs. Given how good or bad we are it, the probability of making a mistake while working is higher or lower, but mostly a number greater than zero. It is important to keep this risk in mind while ascertaining the quality of a build. Indeed, human errors lead to a majority of bugs in software applications — errors that may occur during development and/or testing.

Computers are extremely efficient at doing repetitive tasks. They are diligent and careful, which makes automation a risk mitigation strategy.

Tests as executable documentation

Test scenarios provide an excellent source of knowledge about the state of an application. Manual test results provide a good view of what an application can do for an end user, and also tell the development team about quirky components in their code. There are two components to documenting test results – showing what an application can do and, upon failures, documenting what fails and how, so it's easy to manage application abnormalities. If testers are diligent and make sure they keep their documentation up to date (another overhead for them), it is possible to know the state of play through a glance at test results. The amount of work increases drastically with failures, as testers then need to document each step, take screenshots, maybe even videos of crash situations. Adding the time spent on these increases the cost of making changes; in fact, in a way, the added cost disincentivizes documenting the state with every release.

With automated tests, and by choosing the right tools, the process of documenting the state of an application becomes a very low-cost affair. Automated testing tools provide a very good way of executing tests, collating results in categories, and publishing results to a web page, and also let you visualize test result data to monitor progress and get relevant feedback from the tests. With tools like Twist, Concordian, Cucumber and the lot, it becomes really easy to show your test results, even authoring, to your customers, and this reduces the losses in translation, with the added benefit of the customer getting more involved in the application's development. For failures, a multitude of testing tools automate the process of taking screenshots, even videos, to document failures and errors in a more meaningful way. Results could be mailed to people, much better served as RSS feeds per build to interested parties.

Technology facing tests

Testing non-functional aspects of an application – like testing application performance upon a user action, testing latency over a network and its effect on an end-user's interaction with the application, etc. — have traditionally been partially automated (although, very early during my work life, I have sat with a stop watch in hand to test performance — low-fi but effective!) It is easy to take advantage of automated tests and reuse them to test such non-functional aspects. For example, running an automated functional test over a number of times can tell you the average performance of an action on your web-page. The model is easy to set up: put a number of your automated functional tests inside a chosen framework that lets you set up and probe non-functional properties while the tests are run. Testing and monitoring aspects like role-based security, effects of latency, query performance, etc., can all be automated by reusing an existing set of automated tests — an added benefit.

Conclusion

On your journey to Continuous Delivery, you have to take many steps, both small and large. My understanding and suggestion would be to start small, with a good investment in a robust automation suite, give it your best people, cultivate habits in your team that respect tests and results, build this backbone first, and then off you go. Have a smooth ride! ■

This article originally appeared in blog.ranjansakalley.com.

Ranjan Sakalley is a lead developer & software architect with ThoughtWorks who "likes writing code and working with great people". In his career he has worn varied hats, and in particular enjoys being an agile coach and project manager.

His interests include software architecture, leading teams to deliver better, being a hands-on lead, C#, Java, Ruby, javascript, Agile, XP, TDD, Story analysis, and Continuous Delivery, among others.

50 Quick Ideas to Improve Your Tests - a review

By Marcus Hammarberg

They've done it again. Gojko Adzic, David Evans and, in this book, Tom Roden, have written another '50 Quick Ideas' book. And this one is equally as good as the previous book on user stories. If not even better.

I got so much out of this book, and my tool belt expanded significantly. I really like the approach of these short, focused, one-topic books, starting with Gojko's book on impact mapping. They don't promise to be deep dives and total coverage, but rather to give you ideas (well… that's in the title even), be challenged, and investigate further.

In this book on testing, they have divided the ideas into four groups, brushing on different aspects of testing:

- Generating test ideas
- Designing good checks
- Improving testability
- Managing large test suites

One of the things that struck me is how far (agile) testing has progressed during my relatively short period of interest in the field. This is a very sober and concrete look at the new breed of testers that want to be part in design, that takes failed tests as an opportunity to learn. We have sections on measuring test half times (how often do test change) in order to focus our testing efforts, and suggestions for how to involve and inform business users directly in the creation of key examples, etc. This is not your father's testing… and I like it!

The early parts of this book touch more on organization of test efforts and exploratory testing, etc. There's a lot of good things in there, but it's not my area of expertise and interest.

The two last parts I found extremely interesting and packed with battled-hardened experiences that I sometimes found myself nodding in agreement with. And sometimes I had to reread paragraphs a few times because it was really a new take on a situation I've been in.

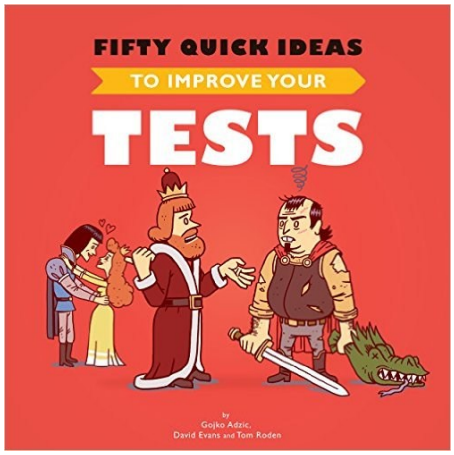
And that's typically how you get the experiences from experts served. Some things you have experienced yourself and others are things that help your knowledge to take a jump ahead.

Everything that I didn't know about before left me feeling that I wanted more pages on the topic. Or examples on how to implement this or that, although every Idea has a "How to make it work" section that gives you a starter.

This is by design.

The book is not meant to be a complete overview. You should, as they point out in the intro, not read this as your first book.

And, I might add: you should not read the entire book in one



go. I would suggest that you browse this book for an overview and general knowledge and then use it as a tool, hands-on, in your team. Keep it next to your team, and as you run into problems, look them up in the book. There are a lot of pointers and ideas that can help you get under control many, if not all, of the testing problems I've seen teams run into.

I could not recommend the book more. Any serious agile tester should have this handy to be inspired to move even further.

Thank you, Neuri "Publishing" — looking forward to the next book. On retrospectives.

P.S. Were you the guys behind '50 Shades of Grey' too? ■

This review originally appeared in the author's blog, marcusoft.net.

Marcus Hammarberg, the author of Kanban in Action, is a programmer, consultant and agile coach who has worked for a number of banks and insurance companies, as well as Ebay and Spotify. At present, he is especially jazzed about Node and Koa Js.

Currently, Marcus lives with his family in Indonesia, where he works for the Salvation Army.

Marcus can be reached on Twitter at @marcusoftnet.

Please Note: Readers of LogiGear Magazine are entitled to a 50% discount on the Ebook version of this book when they use the LogiGear discount code through March 31, 2016.

Upcoming software test-related conferences planned through March 2016

Software Quality Days — **January 18-21** — **Vienna, Austria**

Europe's leading conference on Software Quality

European Testing Conference 2016 — **February 11-12** — **Bucharest, Romania**

A conference about getting experts and practitioners together to talk, learn and practice the art of testing. Looks into advanced new methods of making testing more effective, and enriching understanding of fundamental methods to grow a stronger community.

DeveloperWeek — **February 12-18** — **San Francisco, USA**

San Francisco's largest tech event series with over 60 week-long events including the DeveloperWeek 2016 Conference & Expo, DevOps Summit, WebRTC Summit, LearnToCode Camp, 1,000+ attendee hackathon, 1,000+ attendee tech hiring mixer, and a series of workshops, open houses, drink-ups, and city-wide events across San Francisco.

SOFTENG 2016 - Intl. Conf. on Advances and Trends in Software Engineering — **February 21-25** — **Lisbon, Portugal**

Part of NexComm 2016 - A gathering of multiple co-located conferences in Lisbon

EMBEDDED 2016 - The International Symposium on Advances in Embedded Systems and Applications — **February 21-25** — **Lisbon, Portugal**

Part of NexComm 2016 - A gathering of multiple co-located conferences in Lisbon

CTRQ 2016 - The Ninth International Conference on Communication Theory, Reliability, and Quality of Service — **February 21-25** — **Lisbon, Portugal**

Part of NexComm 2016 - A gathering of multiple co-located conferences in Lisbon

ICONS 2016 - The Eleventh International Conference on Systems — **February 21-25** — **Lisbon, Portugal**

Part of NexComm 2016 - A gathering of multiple co-located conferences in Lisbon

Lean and Six Sigma Conference — **February 29 – March 1** — **Phoenix, AZ, USA**

A conference for those with technical proficiencies and leadership responsibilities who are actively involved in process improvement, organizational change, and development dynamics related to a successful lean and Six Sigma culture.

CQSDI — Collaboration on Quality in the Space and Defense Industry Forum — **March 7-8** — **Cape Canaveral, USA**

If you work with an organization that is involved in the space and defense industry, this event will be your most important and rewarding professional experience for 2016. It includes government and indus-

North Jersey ASQ Spring Quality Conference 2016 — **March 24** — **Whippany, NJ, USA**

"The Global Quality Marches On"

LogiGear®

LOGIGEAR MAGAZINE

DECEMBER 2015 ◆ VOL IX ◆ ISSUE 4

United States
4100 E 3rd Ave., Suite 150
Foster City, CA 94403
Tel +1 650 572 1400
Fax +1 650 572 2822

Viet Nam, Da Nang
VNPT Tower, F/I 7 & 8
346 Street 2/9
Hai Chau District
Tel: +84 511 3655 333

Viet Nam, Ho Chi Minh City
1A Phan Xich Long, Ward 2
Phu Nhuan District
Tel +84 8 3995 4072
Fax +84 8 3995 4076