

PROFESSIONAL TESTER

SUBSCRIBE
IT'S FREE

Essential for software testers

| December 2016 | £ 4 / € 5 | v2.0 | number 39 |



Tools of Tomorrow

Including articles by:

David Gelperin
ClearSpecs Enterprises






Peter Zimmerer
Siemens

Christian Brandes
imbus AG

Paul Gerrard
Gerrard Consulting

Andreas Golze
Cognizant

All-in-One Test Automation

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

1
License
All Technologies.
All Updates.



www.ranorex.com/try-now

Cross-Technology | Cross-Device | Cross-Platform | Cross-Browser

Tools of Tomorrow

Tools of tomorrow

As 2016 draws to a close, we asked our contributors to look ahead and give their views on what could - or should - shape testing in the years ahead.

David Gelperin tackles the failure to address 'technical debt' and urges testers to focus on the definition and verification of quality goals as well as functional requirements. Peter Zimmerer reveals that Siemens has created 'Test Architect' as a new role which it now views as critical. He outlines the 'why' and 'how' of the role and points out that this is far more than a 'face lift' for testing, as it will play a central role in the ongoing success of the company.

As well as fresh thinking, Paul Gerrard also looks at how bots could play a part in

supporting (not replacing) testers. And in the first of a two-part feature, he invites readers to join the conversation on how a tool-supported hybrid exploratory/structured approach could work.

There is plenty more to read and we are looking forward to hearing your views. In the meantime, the team at Professional Tester would like to wish you all a very happy Christmas and here's to a prosperous 2017.

Vanessa Howard

Editor



Contact

Editor

Vanessa Howard
editor@professionaltester.com

Managing Director

Niels Valkering
ops@professionaltester.com

Art Director

Christiaan van Heest
art@professionaltester.com

Sales

advertise@professionaltester.com

Publisher

Jerome H. Mol
publisher@professionaltester.com

Subscriptions

subscribe@professionaltester.com



Contributors to this issue

David Gelperin
Peter Zimmerer
Christian Brandes
Paul Gerrard
Andreas Golze

Editorial Board

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

Professional Tester is published
by Professional Tester Inc

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

© Professional Tester Inc 2016

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

IN THIS ISSUE

4 Fixing agile

David Gelperin sets out how a new approach to agile can avoid the high cost of technical debt.

10 Test Architect a key role defined

Siemens are aiming to employ 50 test architects by the end of 2017 and Peter Zimmerer explains why this will prove fundamental if tomorrow's challenges are to be met.

14 Designing near-production test environments

Christian Brandes outlines a new approach to designing test environments that is feasible, describable and comparable.

18 New Model Testing: A new process and tool – part 1

Paul Gerrard argues that old test methods won't work in the future and in this opening feature, examines a new approach that incorporates tools to support testers.

22 Testing Talk

In this issue's Testing Talk Cognizant's Andreas Golze discusses how companies with siloed practices can move to a well-implemented DevOps strategy.

Fixing agile

by David Gelperin

The way to manage quality debt is early identification of quality goals



Projects with tight deadlines are often under pressure to take shortcuts that may sacrifice quality.

Big Requirements Up Front (BRUF) is an agile anti-pattern. Agile discourages early identification of comprehensive requirements for fear of waste and gold-plating. Building something incrementally and showing it to customers for timely feedback often makes perfect sense **for functionality**. But, functionality is not the whole story.

Consider quality attributes, both external and internal, and their goals. **External qualities can be seen by customers.** Behaviours such as safety, security and reliability need time periods of years to suggest their satisfactory achievement. Other external qualities, such as ease of use, can show satisfactory achievement in

a shorter time. The situation is much clearer when achievement is unsatisfactory. This is when systems kill, crash or are breached.

Internal qualities such as portability, testability and code readability, **can't be seen by customers.**

Achievement of all internal and many external quality goals crosscut the functional code. For example, exception handlers should be everywhere. Cross-cutting qualities can't be implemented in a single module or group of modules. They must be implemented everywhere.

This fact is a problem for any development process that doesn't identify its quality goals, achievement strategies, and verification strategies before incremental development. Without early identification, each early increment will have high-cost technical debt due to voluntary ignorance.

Technical debt refers to the extra work created when "easy-to-develop" code is used instead of code for the best overall solution. "Easy-to-develop" entails the use of simplistic or unnecessarily complex designs. Technical debt always entails shortcuts, both deliberate and unintentional. Unintentional debt results from ignorance of needs or effects.

Examples of deliberate debt are code that doesn't validate input, handle edge cases, or handle exceptions. Examples of debt that may be deliberate or unintentional are missing or incorrect functionality, code that doesn't comply with development standards, and code that doesn't deal with other quality goals such as safety, security, or privacy. Examples of unintentional debt are code that has unnecessarily complex logic or design and unnecessary functionality.

From a requirements perspective, we distinguish functional debt from **quality**

debt. Functional debt such as incomplete or incorrect functionality normally has a moderate cost of failure and repair. Quality debt usually has a much higher cost because of its criticality and crosscutting nature.

A smart debt management tactic is to **avoid unintentional quality debt** by:

1. deepening quality understanding using a comprehensive quality model
2. collecting quality lessons
3. complying with effective development standards that control complexity
4. using risk-prioritized quality verification

Unless quality debt is paid off before final delivery, the final product will have significant quality defects. Delayed identification is also a problem for customer expectations. Customers have seen the functionality, so they have difficulty understanding the delay in application delivery caused by quality refactoring.

The way to manage quality debt is early identification of quality goals and incremental execution of achievement and verification strategies. Since developers often focus on functionality, testers should focus on the definition and verification of quality goals as well as functional requirements. Verification (tactics are included in the model shown in Figure 1) includes various forms of testing as well as operational monitoring, technical reviews, verification-focused analysis, and verification-focused measurement. Since the achievement of internal qualities can't be demonstrated by testing or operational monitoring, their verification relies on automated static analysis and technical reviews. Even the effective verification of some external

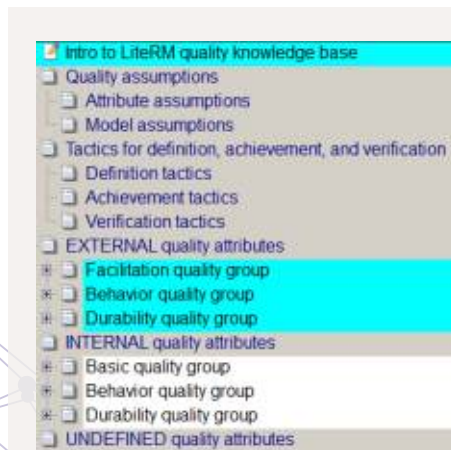


Figure 1. Top-level view of the LiteRM quality model.



Figure 2. External quality groups and subgroups.

qualities such as safety relies on analysis and technical reviews as well as testing.

To verify quality goals, testers must expand their scope of responsibilities to include the team's use of other verification tactics. Testers should promote quality-aware development. Quality-aware development is NOT a development methodology, but a three-part supplement to whatever incremental methodology you are using now (including agile).

Part one of the supplement is an initial quality sprint that includes:

1. Selection of relevant quality attributes from a comprehensive quality model, definition of quality goals and assignment of priorities, during a technical review;

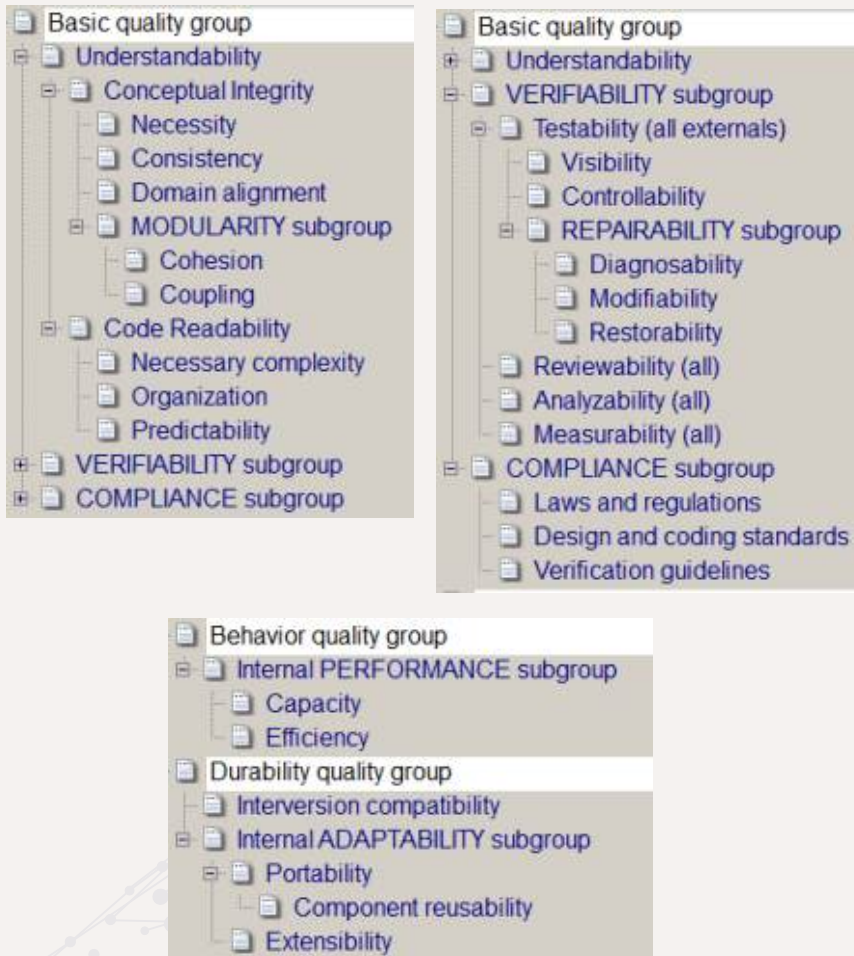


Figure 3. Internal quality groups and subgroups.

2. Analysis of each pair of potentially conflicting quality goals and identification of ways to resolve the conflicts so an adequate architecture can be identified;
3. Analysis of each quality goal to identify its hazards, mitigations, and supports, specification of incremental achievement and verification strategies, assessment of their feasibility, and adjustment as needed.

Part two is a set of tasks to be added to each iteration.

For each quality goal:

1. Reassess its achievement and verification strategies and update as needed;
2. Carry out its achievement strategy, clearly identifying quality support code;

3. Verify its incremental achievement and remove defects as needed.

Part three is a quality retrospective that collects quality learnings and records them in the comprehensive quality model and/or the development standards.

The feasibility of quality-aware development depends on the use of a quality model that describes a comprehensive set of qualities and their characteristics including leading and operational indicators, supporting and conflicting qualities, threats and mitigations, and achievement and verification tactics. Such a model helps to define quality goals, achievement strategies, and verification strategies.

One quality model is organized into external and internal attributes, as shown in Figure 1.

These two collections (as shown in Figures 2 and 3) are organized into groups and subgroups.

Each group displays possible support relationships between its attributes. When one attribute might support another, such as reliability supporting availability, the supporting attribute is indented.

The model uses “subgroups” to indicate **quality topics** that are fully factored into a set of quality attributes. For example, the performance subgroup is a quality topic, decomposed into responsiveness, throughput, capacity and efficiency. Topics have no existence or meaning separate from their component attributes. If “improved performance” were to appear as a requirement, it should be identified as unacceptably vague.

Topics and attributes are **displayed as an outline**. However, this is a simplification of their actual relationships. For example, the basic attributes (the first group of internal attributes) support all other attributes, including themselves. Some attributes are supported by attributes that are not directly below them. The actual support structure is explicitly described in the attribute characteristics. Using safety as an example, we

illustrate the characteristics associated with each quality attribute in the table below. Please note that the characteristics in *italics* are project-specific.

Safety characteristics

Definition

The ability of a system to do little or no harm to valuable assets

Software subfield

Safety engineering

Concerned stakeholders

Customers, lawyers, tasked users, general public, designers

Assumptions/rationale

Safety is a fragile quality because it depends on up to 39 other qualities as well as on an accurate analysis of the hazards that must be managed

Quality attribute scenarios

Described in **Software architecture in practice**

Leading indicators - provide preoperational evidence of attribute goal attainment

- Ratio of hazards added during HA technical review to hazard count after HA technical review
- Ratio of defects found in safeguards during testing to number of safeguards

Operational measures

- Time since last "dangerous" failure or defect
- Number of "dangerous" failures or defects detected per time interval
- Greatest harm from a harmful event
- Shortest harmful event free duration
- Longest harmful event free duration
- Expected length of harmful event free duration
- Expected rate of harmful events
- Ratio of actual loss to acceptable loss in a duration
- Estimated residual risk

Supporting qualities – always or some-times - dependability, **ease of learning**

Note: While safety-critical functionality may be supported by these qualities, at the same time they may conflict with non-safety-critical functionality. For example, availability supports dependability, but it may cause non-safety-critical functionality to be sacrificed so the system can continue to operate in a safe, but degraded mode.

Conflicting qualities - adaptability qualities

Threats - identify using hazard analysis

Mitigations - identify after identifying hazards

Other achievement tactics

- identify valuable assets and hazards
- identify safety-critical and safety-related functions and constraints needed for safety e.g. "The fire detection system shall detect smoke above X ppm within 5 seconds"
- isolate and protect safety-critical functions
- guard safety-critical functions with explicit conditions i.e. never with defaults such as "otherwise"
- identify safety-critical users
- eliminate or mitigate hazards i.e. identify appropriate control actions
- effectively execute control actions and receive accurate and sufficient feedback
- monitor system state to make sure safety-critical and safety-related functions are permitted
- alert users to dangerous actions with rotating warning messages
- precede each dangerous action with a delay so user can change their mind and cancel
- limit complexity
- design interfaces that prevent and detect user errors
- use warning labels and messages when appropriate

Verification tactics - review hazards and mitigations for completeness and effectiveness during a safety audit, thoroughly test each safeguard, measure and track time since last "dangerous" failure or defect and number of "dangerous" failures or defects, verify all supporting qualities

Elicitation Questions

- What valuable assets are at risk
- Which functions are safety-critical or safety-related
- Who/what can perform these safety-critical or safety-related functions and under what conditions
- What harm can the system or its actors possibly do
- What can mitigate these hazards

Associated Tools

- Measurement
- Achievement
- Verification

Resources

"Quality Attributes" Technical Report CMU/SEI-95-TR-021 Chapter 6
Engineering a Safer World
Software Safety Primer

Risk Factors

- Developer understanding = [superficial, limited, deep]
- Cost (implementation, verification, maintenance) = [high, medium, low]
- Feasibility (technical, cost, understanding) = [low, medium, high]

Other Characteristics

- Sources/Enterprise goals
- Type = behaviour quality
- Associated scope = [system, <specific partitions>]
- Design scope = crosscutting [local, crosscutting]

- e. *Consensus priority* = [critical, important, desirable]
- f. *Architecture-relevant* = maybe [yes, maybe, no]
- g. *Visibility group* = externals

States

Goal states are < @unverified, verified, implemented, inactive>

Notes

Past goal specs

Past achievement and verification strategies

Current goal spec

Current achievement and verification strategy

This quality model is freely available at
www.quality-aware.com/q-a-daves-stuff.php

The 10th Annual State of Agile Report (2016) by Version One suggests that agile needs to become quality-aware.

Among survey respondents, 56% don't comply with coding standards and 63% don't refactor. Unless most respondents not complying with coding standards only work in small teams, it appears that some agile projects sacrifice code readability from the start.

The meaning of the second result is less clear. Do the non-refactoring respondents: (1) never refactor, (2) refactor early, but not late or (3) have their code refactored by others? If their code is never refactored and they don't identify quality goals early,

the code will have many quality defects.

In conclusion, we know that waterfall development promotes BRUF including Quality Goals Up Front (QGUF). While BRUF may be a poor strategy for functionality, QGUF is the best strategy for quality goals. Projects with tight deadlines are often under pressure to take shortcuts that may sacrifice quality. Projects that develop capabilities in priority order using QGUF and incremental quality development are in a better position to stop.

If you like the challenge of effective testing, **you will love the challenge of effective quality verification ■**

Since 1965, David has been involved with software development, with a focus on testing, quality support, and requirements. He enjoys learning and helping others improve their software practices.

Test Architect a key role defined

by Peter Zimmerer

Siemens is committed to creating a new role within testing so what has driven the decision and what responsibilities will come with the position?



A close collaboration with other roles and stakeholders is crucial for test architects.

What is a “Test Architect”? At Siemens we have invented this new key role by defining the responsibilities and tasks of a test architect and this article is about our journey to establish this new key role within the company.

What?

Currently there is no universal definition of a “test architect” in the community and only a few companies are implementing this role differently, for some examples see References 1-5. To me a test architect is neither just “a fancy title for an experienced tester” nor only a “senior test role with wide strategic scope” (see Reference 2). And defining a test architect as “a person who provides guidance and strategic direction

for a test organization ...” (see Reference 5) is ok, but there is more ... and there is also a real business need for more ...

Why?

Imagine that you build a system with 10 million lines of code. To be successful this system should have a good architecture, and for that you need good software architects (see Reference 1). Next, to perform good testing on 10 million lines of code, you also need to build an appropriate test system - not by spaghetti coding but with a well-designed, sustainable test architecture and by applying innovative software and test technologies.

In the real world, our test systems increase in size, volume, complexity and unpredictability (think about testing of autonomous systems for example). Additionally, digitalization (virtualization, cloud, mobile, big data, data analytics, internet of things, etc.) requires more than just a face lift in testing.

But, who is responsible for making this happen? Typically, neither the test manager (focus on budget, people, plans and logistics for the test organization) nor the quality manager (focus on process quality, certification and audits, regulations and norms across lifecycles and versions) will do this; therefore we need to create a new role on eye level with the software architects: the *Test Architect* is born.

Overall there are two major goals we want to achieve by test architects to meet the diverse challenges of shorter time-to-market, increasing complexity and more agility while keeping quality and other key system properties high:

- Better *test strategies* – raise testing to the strategic level closely aligned with the business goals and business drivers

- Better *test systems, test architectures, test technologies*

How?

To specify, implement, introduce, and establish this new key role we have developed the following four pillars:

- A role profile specifying the responsibilities and tasks of a test architect in different areas: business understanding, requirements engineering, architecture and development, testing and quality, social skills and leadership
- A corresponding target profile with the needed skills and competencies in the different areas (see Figure 1)
- A unique expert training for test architects (see Figure 2) to set standards for testing within Siemens and to foster practice sharing and experience exchange within the test architects' network
- A certification to assure competencies and practical experience for critical projects

A test architect has two main responsibilities that are directly linked to the two major goals mentioned above:

- For one the test architect is the *test expert* to define and apply innovative test strategies, test technologies, and novel test automation approaches and thus drive the overall quality of the system by effective and efficient testing over the whole lifecycle.
- On the other hand the test architect is the *software architect for the test system* (including test environment, test infrastructure, test automation frameworks) that is needed to test the system under test (SUT). Because the SUT is getting more and more complex, heterogeneous, and large in size with millions of LOC, the corresponding test system (that won't be a small system in itself!) must be based on an adequate, sustainable, well-designed "test architecture". The term "test architecture" is not widely used in the industry yet, but it is increasingly addressed in different research communities and standards, e.g. see References 6-10. An example for a high-level test architecture as specified

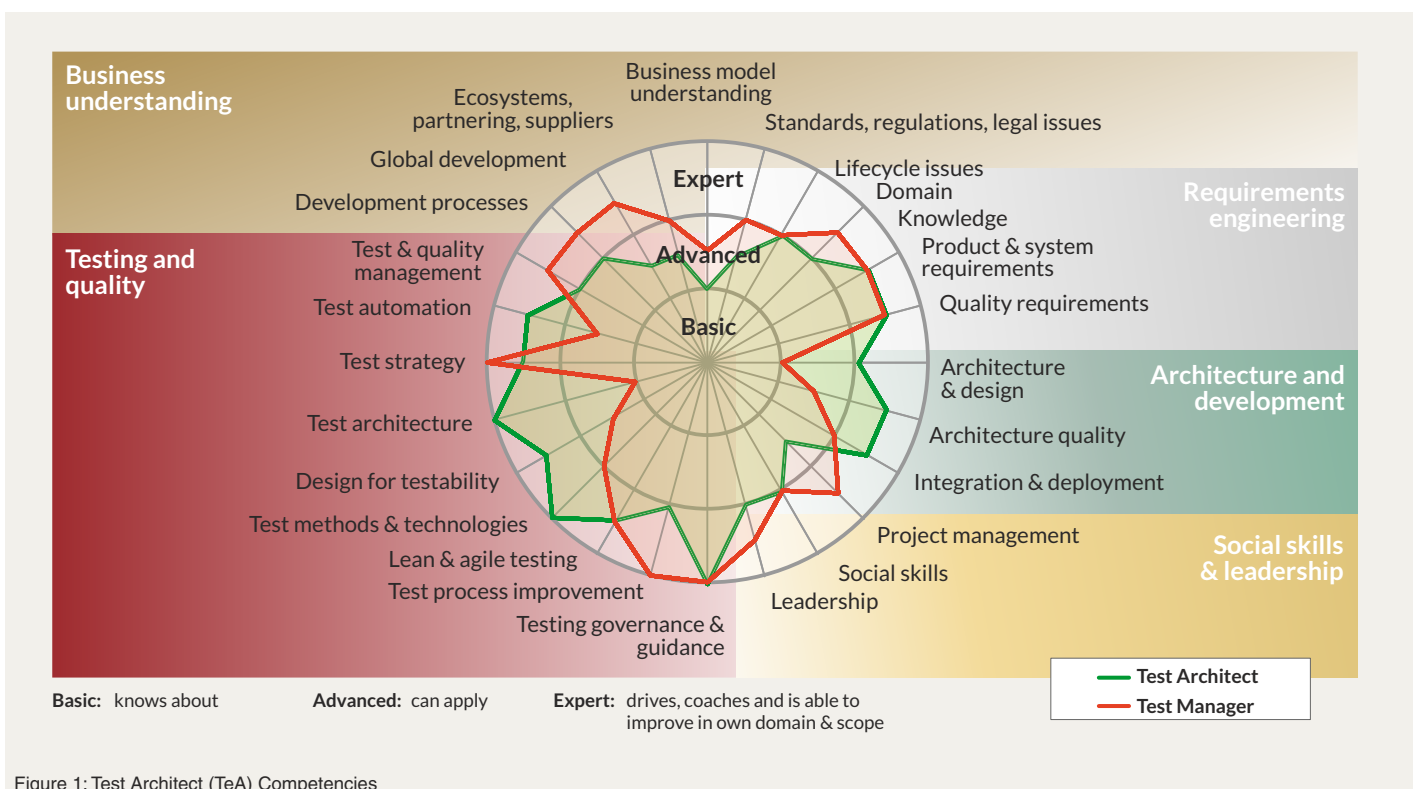


Figure 1: Test Architect (TeA) Competencies

in Reference 7 is shown in Figure 3: that can be the starting point to derive and detail a “test architecture” for a specific test system.

To achieve this, test architects also have to apply state-of-the-art strategies, methods, practices, and tactics from software architecture and design: e.g. identify architectural forces (i.e. requirements and constraints), derive architecture significant requirements (ASR) based on business and technical risks, create a domain model for the problem space, model dynamics of scenarios, determine scope and boundaries of the system, do functional partitioning (logical view), structure the baseline architecture, add different architectural views, use design patterns, do strategic and tactical design, etc. That’s a lot of (interesting!) stuff for test architects to apply – an explicit description and explanation of these topics goes beyond the scope of this article, for more details see for example Reference 11.

A close collaboration with other roles and stakeholders (e.g. product manager, software architect, system integrator, operation engineer, etc.) is crucial for test architects, so collaboration and interaction is explicitly included in the test architect’s role profile. It specifies precisely what topics the test architect is “responsible” for and for which they are “involved”.

For example, a test architect is responsible for fostering the use of test-driven approaches in requirements engineering; to identify and manage technical risks for the testing environment and to drive integration of the testing infrastructure with the system under test (SUT). A test architect is involved in reviewing the architecture of the SUT and performing impact analysis for changes in the software system.

Especially important is the collaboration between the software architect and the test architect in a kind of “joint venture” to ensure that the system under test (SUT) and the test system (that is used to test the

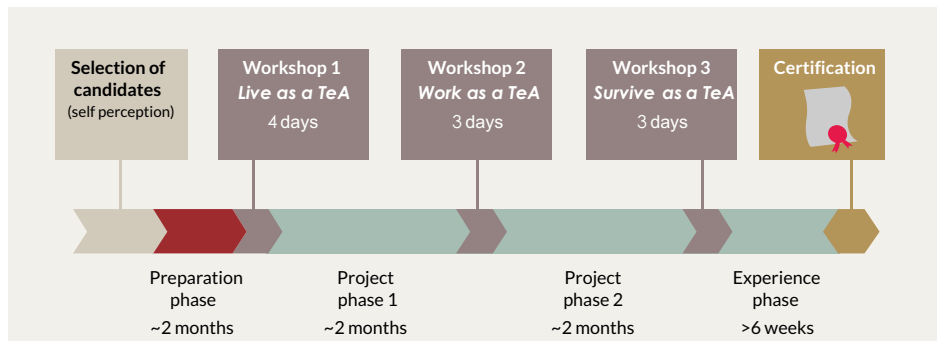


Figure 2: Test Architect (TeA) expert training – schedule

Test environment

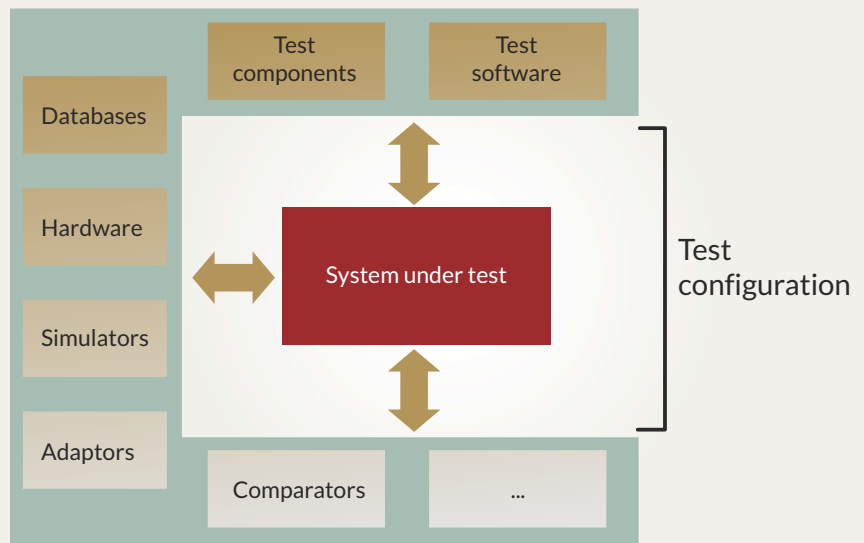


Figure 3: UML testing profile – Concepts of test environment and test configuration (see Reference 7)

SUT) fit together well by good design for testability (this will influence sustainability, maintainability, serviceability as well) and that the architecture of the overall test automation approach (test architecture, testware artefacts, test automation frameworks, testing tool landscape) is effective and efficient (see Reference 12).

For the participants of our expert training we recommend an ISTQB advanced level certification as a prerequisite, so we do not compete with ISTQB; in fact, our expert training is different, complementary and more advanced:

- To match the role profile and to achieve the target profile (see Figure 1) our test architect expert training explicitly focuses strongly (ca. 60% of the content) on topics beyond the “testing and quality” topics: business understanding (e.g.

product lifecycle models), requirements engineering (e.g. non-functional requirements), architecture (e.g. systematic architecting, architectural views and documentation, architecture reviews, refactoring) and social skills, as well as cross-cutting topics (e.g. lean and agile development).

- The expert training is a series of highly interactive workshops with project phases in between (see Figure 2): here the participants work on concrete tasks in their real-world projects to enforce learning, practicing and continuous improvement. At the end of the day it's all about using and applying the right set of strategies, tactics, methods and techniques as a major lever to design, implement, execute and sustain your specific approach for better testing.

Outlook

A test architect (on eye level with the software architect!) can also be a new career path for testers but only if you as a test architect provide value and create impact on the business. Let's continue our journey in this direction – at Siemens our target is to have nearly 50 test architects by the end of 2017 ■

If more hardware is involved than we also need system architects, but for simplicity, this article focuses solely on software architects.

References

- [1] Page, A., Johnston, K. and Rollison, B. 2008. *How We Test at Microsoft*, Chapter 2. Microsoft Press, 2008.
- [2] Page, A. 2008. *What is a Test Architect?*
<http://blogs.msdn.com/b/alanpa/archive/2008/05/01/what-is-a-test-architect.aspx>
- [3] Morrison, J. 2007. *Test Architect*.
https://blogs.oracle.com/johnmorrison/entry/test_architect
- [4] Brewer, J., Guise, L. and Emmert, J. *The Value of the Test Architect*. Raytheon. March 2012. www.dtic.mil/ndia/2012TEST/13942_Brewer.pdf
- [5] ISTQB Glossary, *Test Architect*, <http://www.istqb.org/downloads/glossary.html>,
<https://www.astqb.org/glossary/search/test%20architect>
- [6] Zimmerer, P. 2011. *Architecture Testing @ Siemens. Workshop on Architecture-Based Testing at the Software Engineering Institute (SEI)*, Pittsburgh, PA, US, 2011. <https://saturnnetwork.wordpress.com/2010/11/23> and <https://saturnnetwork.wordpress.com/2011/02/22>
- [7] UML Testing Profile (UTP) Version 1.2. 2013. Chapter 8 "Test Architecture", pp. 19-24. <http://utp.omg.org/>, <http://www.omg.org/spec/UTP/>
- [8] Nishi, Y. *Viewpoint-based Test Architecture Design. International Workshop on Metrics and Standards for Software Testing (MaSST)*, 2012.
- [9] Eldh, S. 2015. *Software Test Architecture. 2nd International Workshop on Software Test Architecture (InSTA)*, 2015.
<http://www.aster.or.jp/workshops/insta2015/>
- [10] Nishi, Y. *Design principles in Test Suite Architecture. 2nd International Workshop on Software test Architecture (InSTA)*, 2015.
<http://www.aster.or.jp/workshops/insta2015/>
- [11] Bass L., Kazman, R., Clements, P. *Software Architecture in Practice*. 2012
- [12] Paulisch, F., Zimmerer, P. 2016. *Collaboration of Software Architect and Test Architect Helps to Systematically Bridge Product Lifecycle Gap. 1st International Workshop on Bringing Architecture Design Thinking into Developers' Daily Activities (Bridge'16), co-located workshop at ICSE 2016, Austin, TX, US, May 14-22, 2016*

Peter Zimmerer is a principal key expert engineer at Siemens AG, Corporate Technology.

Designing near-production test environments

by Dr. Christian Brandes

Cut through the complexities of constructing test environments to show a necessary near-production level.



Using this approach, designing test environments becomes feasible, describable and comparable.

Designing test environments can be a complex task, especially when considering how frequently their contribution towards a project's success can be underestimated. Testing environments lack an enshrined set of standards and the definition of something as fundamental as a near-production environment can be changeable or vague.

This article volunteers a vanguard definition of “near-production” and suggests a systematic approach in describing and managing test environments, as well as unifying various test environment concepts in order to facilitate their fair comparison.

Status quo

Organizations with legacy systems can sometimes suffer from missing standards to regulate their test environments, which can often cause them to descend into confusion and obfuscation over time. This is understandable in a subject as complex as test environments where such aspects as test data, data organisation, data simulation, user IDs, operational components, interfaces and deployment must all be considered.

Towering above all these considerations is the goal of having test environments as “near-production” – i.e. as similar as possible to the actual production environment – as possible to actually generate meaningful test results. The **exact** definition of such an objective – to what extent test environment A may be more similar to production than test environment B – often remains nebulous, at best. In response to that, an unfortunate trend developed that has encouraged testing in the actual production environment. Adherents of the saying, “The only real thing is the real thing,” then find themselves following inefficient test strategies that, crucially, lose sight of the central project goal, which should be the earliest possible identification of errors.

Environments in IT - standards

Complexity may also be to blame for the relative neglect with which the term “environment” has been treated thus far by standard software literature.

Reference 3 shows the definition of as taken from the ISTQB glossary (c.f. Reference 1) that cites the IEEE610 standard, stating, “Test environment: an environment containing hardware, instrumentation, simulators, software tools and other support elements needed to conduct a test.”

Reference 2 explains that there are essentially three types of test environments: laboratories, system test environments and “near-production” environments. While all of this may be valid, it fails to be precise enough for a solid definition of “near production”, inasmuch as it is insufficient for the governance or management of test environments. Operational standards such as ITIL do not provide feasible guidelines for test environments, either. Even the oft-quoted TPI, when looking at test processes, does not disclose **how** well-fit test environments are to meet the necessary test requirements.

“Near-production” as a central term

The concept of “near-production” can be refined by establishing its definition.

Take, for instance, the environment characteristic “network bandwidth”. This characteristic is completely irrelevant for usability testing, or to execute a functional component test. If it is performance that one hopes to test, however, this environment characteristic suddenly becomes crucial. Accordingly, certain aspects of test environments can, at times, be parts of a “near-production” environment, while in other times they become irrelevant. At the root of this, lies the **objective** of the test activity in question, hence, the following definition can be derived:

Near-production refers to those test environment characteristics that must imitate the actual production environment in order to reach a defined test objective.

“Test objective”, in this case, is understood as set forth according to ISTQB – see Reference 1.

Using test objectives as a starting point leads to the following procedure: for every intended test activity, as associated test objective is set. The necessary “near-production” environment can then be established. On that basis, the required test environments can be identified and, if necessary, linked up with each other, whereby a purpose-orientated mapping

between tests and environments can be obtained.

This practical approach will prove more productive than the inverse scenario, in which constructing test environments precedes the planning of activities to pursue within them. Broadly put, it could be argued that one ends up having produced a solution to a previously non-existent or improperly identified problem, resulting in a test environment without a specific purpose, which is then used arbitrarily.

The journey from test objective via required near-production to the test environment

How does one go about translating the above proposition into practise? As a first step, all **test activities** must be listed and matched with their respective **test objectives**. The following table generally illustrates this point by enumerating some test activities and test environments. It has been purposefully complemented by items such as “production” and “training” in order to form a more rounded picture.

The second step requires summing up all **environment characteristics** that have to be taken into consideration. A list of such characteristics can be produced pragma-

tically and iteratively, perhaps while consulting existing Configuration Management Databases (CMDB), if accessible. Apart from completeness, the granularity of these characteristics may raise their own issues, requiring separate considerations. Crucially, however, every relevant requirement of environment characteristics can be captured by the evolving scheme.

The following lists instantiates a possible enumeration of these characteristics:

- Hardware
 - Server
 - Client
 - Storage
 - Sizing
 - Peripherals
 - Network
- Software
 - Server OS
 - Client OS
 - Databases
 - Test object configurations
- Middleware
 - Application server
 - Enterprise service bus
 - Batch controller
 - Service balancer

(Test-) activity	(Test-)objective
Evaluation	Free testing.
Development	Coding.
Component tests	Early identification of functional, robustness and performance errors.
Component integration tests	Test of interfaces and cross -component functionality.
Functional system tests	Tests from the user’s perspective (regarding functionality, robustness, system interfaces).
Performance tests	System test looking at performance and resource efficiency.
Usability tests	Considers usability and accessibility.
Functional acceptance tests	Validation of acceptance criteria.
Technical acceptance tests	Test to establish the fulfillment of operational acceptance criteria.
System integration tests	End-to-end test of important business processes.
Production	Productive use of the actual system.
Training	Training of follow-up, as well as current versions of test objects.
Failure analysis and re-test	Reproduction of complex production-flaws. Proof of their redemption.

R = required	HARDWARE					SOFTWARE/MIDDLEWARE					DATA				OTHER (SECURITY, ADMINISTRATION...)						
	Server	Client	Storage	Peripherals	Network	OS Server	OS Client	Load balancer	DB/Cluster	Test object/config.	Base data	pre-cond. data	amount	up-to-date-ness	Deployment	partner systems	accessibility tools	downtime	data simulation	SSO	Auth. Server
Evaluation	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Development	○	○	○	○	○	○	R	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Component tests	○	○	○	○	○	○	R	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Integration tests	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	R	○	○	○	○	○
Functional system tests	○	R	○	R	○	R	R	○	○	○	R	R	○	○	○	R	○	○	○	○	○
Performance tests	R	R	R	○	R	R	R	R	R	○	R	○	R	R	○	○	○	○	○	R	R
Usability tests	○	R	○	○	○	○	○	○	○	○	○	○	○	○	?	○	○	○	○	○	○
Accessibility tests	○	○	○	○	○	○	○	○	○	R	○	○	○	○	○	○	R	○	○	○	○
Functional acceptance tests	○	R	○	R	○	R	R	○	○	R	○	○	○	R	R	R	○	R	R	R	R
Technical acceptance tests	R	R	○	R	○	R	R	○	R	R	○	○	○	○	R	R	○	R	R	R	R
System integration tests	○	○	○	○	○	○	○	○	R	○	○	○	○	○	○	R	R	○	○	○	○
Production	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Training	○	○	○	R	○	○	R	○	○	○	○	○	○	○	○	R	R	○	○	○	?
Failure analysis & re-tests	R	R	○	R	○	○	○	R	R	R	R	R	○	R	R	R	○	R	○	○	○

Figure 1: Example of a completed near-production matrix

- Data
 - Basic data (e.g. postcode catalogues)
 - Historic data (e.g. existing customer data)
 - Amount
 - Accuracy
- Security
 - Single sign-on
 - Authentication server
 - Encryption
 - Certification
- Administration
 - Deployment
 - Maintenance downtimes
 - User IDs
- Other
 - Internal partner systems
 - External partner systems
 - Multi-client capabilities
 - Accessibility and personalization support
 - Language services
 - Run-time libraries
 - Date/time simulations

On collating the information assembled at both stages – test activities/objectives and environment characteristics – a matrix emerges which provides the possibility of describing the degree of “near-production” per test activity in a straightforward manner.

Bearing in mind our definition of “near-production”, a “R” in the matrix signifies that the given environment characteristic **must** be reproduced as it is in the production environment. Figure 1 gives an excellent example of such a matrix. It features a row labelled “production”, which refers to the production environment itself, in which every environment characteristic by definition is marked with a “R”.

Identification of multi-purpose environments and their characteristics

After completing the table with the help of the respective knowledge stakeholders, similar matrix rows can be included under common categories (clustering).

Drawing from this, all necessary environments and their properties (supersets of required environment characteristics) can be deduced. As a result, one may obtain environments such as:

- Development environment
- Integration environment
- Production environment
- Production mirroring

When looking at such results, it should be remembered that the characteristics of **physical** environments are taken into

consideration. A physical environment generally serves as the basis for several virtual environments that can be located in the same physical environments at the same, or at different points in time - for example, an integration environment that temporarily serves as an environment for acceptance tests, and is later used for system integration tests. The topic of virtualization can be equally managed with the above scheme as it is similarly addressed by the question, "Is the respective test objective affected by the use of virtual machines?"

Finally, it is crucial not to forget the most important motivation behind the reproduction of test environments: the need for multiple, different versions (of the same environment) to run simultaneously. Most test activities target the next version of the product as it is to be released. Due to processes like the analysis of production errors, or additional migration activities like database upgrades, however, there is a constant need for a test environment that represents the actual production environment. This demand must be recognized during the overall environment planning and design phase.

Achievements

The table in Figure 1 constitutes the basis for establishing usage guidelines, management, planning and governance of test environments. Additionally, it opens up two lines of interpretation. Firstly, one may use it to determine **what** is tested **where**. Secondly, one may also derive the totality of all requirements to be fulfilled by a test environment. A similar table may be found within Reference 2, which seeks to relate topics such as test type, quality characteristic and environment type to one another, but which does not seek to describe environment characteristics in detail.

Furthermore, Figure 1 allows answers to be sought for more detailed questions. For instance, some would dismiss the use of automation tools within an integration environment due to its "near-production" state, but this argument can

now be proven invalid, especially if the use of automation tools does not jeopardise a given test objective.

Additionally, this allows the focussed establishing of future tests within the most suitable environment. When met with a request like, "I am in need of an environment for ...", the test objectives and necessary near-production requirements can then be assessed. The profile which emerges will clearly determine the most appropriate test environment for any given test.

Conclusion

Using this approach, designing test environments becomes feasible, describable and comparable. Newly emerging requests for additional environments can be catered to in a reliable and systematic manner. Moreover, all stakeholders are given a mutual voice, in that they are equipped with a shared understanding and common vocabulary to communicate. Finally, this approach is widely adoptable because of its simplicity, as it offers a chance to document diverging ideas about test environments in a collaborative and comprehensible fashion ■

References

1. ISTQB – Glossary of Testing Terms (Version 2.1)
www.istqb.org/display/ISTQB/Glossary+of+Terms
2. Pol, M.; Koomen, T.; Spillner, A.: Management und Optimierung des Testprozesses dpunkt-Verlag, 2. Auflage, 2002, 2nd Edition (only available as e-book)
3. Spillner, A; Linz, T: Basic Concepts of Software Testing, (Volume 2) 4th Edition, dpunkt-Verlag 2010

Dr Christian Brandes is a trainer and principal consultant for imbus AG in Germany.

New Model Testing: A new process and tool – Part 1

by Paul Gerrard

In the first of two articles, Paul Gerrard shares his experiments with the next generation of testing tools to encourage conversations about the future of testing.



At least in the medium term, tools will support, rather than replace, testers.

This paper sets out some work I've been doing based on the New Model for Testing (see Reference 1). I have been investigating the use of the New Model approach with a tool to support exploration and testing and want to share some findings and consequences of this work. I've created a partial, prototype tool that aims to support testing in digital, collaborative, shift-left, continuous delivery, DevOps environments. I've said before, "The old test methods won't work in the future," and with new thinking, the tasks of exploration and testing presumes the use of supporting tools.

Some interesting consequences arise: we need tools that differ from those currently available and the test process that emerges is somewhat different to what we are used to. This first article describes the goal and a proposed solution and tool architecture. The second article will be an experience report.

If you are interested in collaborating, sharing these ideas and/or looking at the tool in the future, I'd love to hear from you.

A hybrid test approach

Much has been said and written about the benefits of what is commonly called exploratory testing, but it remains that scripted or pre-planned testing has benefits too – and, of course, both have their drawbacks. Although people get into heated arguments about whether scripts are a help or distraction, whether exploratory testing can be trusted to be thorough or whether test automation is testing at all – these approaches have their place.

In Table 1, I have separated out the good and bad aspects of both approaches and the obvious question arises: is it feasible to combine the best of structured and exploratory testing and create a new test approach?

Table 1 generalises of course and you might challenge some of my suggestions. I have never believed the two approaches are "opposites" in some way. A rather better representation would be that they are styles of testing that differ in emphasis. Like the Agile Manifesto, it's more a set of values/preferences that drive testers' behaviour:

- Differences in scale and duration
- Planned vs. improvisational

- Copious vs. light documentation
- Process and governance vs. agility and freedom
- Etc., etc.

Obviously, there are many hybrid approaches, not just these two.

New model testing

The New Model suggests that testing is comprised of two activities and modes of thinking:

1. Exploring our sources of knowledge. Requirements, specifications, users, developers, interviews, the old system and the new system (when available) are all sources. They are all fallible in different ways. To assimilate the information (to “understand” a requirement or feature) – from diverse, confusing and sometimes conflicting sources – we build mental models.

2. Testing: When we trust that we understand a requirement or how a feature should work – that our mental model is sound – we use our model (our understanding) to inform our test design.

The model (Figure 1) identifies ten thinking activities that mostly correspond with physical tasks. Note that the ten activities – enquiring, modelling, informing and so on – don’t have entry and exit criteria, a procedure or deliverables. They aren’t a process as such – each arrow represents the different modes of thinking that occur as we explore and test.

Our brains are more sophisticated than we know, and it is clear that several of these thinking activities can take place simultaneously. Our subconscious often makes a large contribution to our thinking, of course.

	Structured testing	Exploratory testing
The good	<ul style="list-style-type: none"> • Systematic • Transparent, documented • Reviewable • Auditable • Repeatable, measurable • Automatable 	<ul style="list-style-type: none"> • Agile (in the broadest sense) • Improvised, imaginative • Flexible and responsive to change • Faster, cheaper • More effective • Personally enjoyable
The bad	<ul style="list-style-type: none"> • Inflexible, not responsive • Obsolescent/inaccurate documentation • Prone to biases, inattention • Outdated process • Expensive, inefficient • Unimaginative, boring 	<ul style="list-style-type: none"> • Not repeatable • Not easily automated • Little or no documentation • Hard to manage • Hard to scale • Opaque • Not auditable, measurable.

Table 1. The good and bad of structured testing vs. exploratory testing

Bots support surveying and testing

I was inspired by the success of Google’s AlphaGo programme (see Reference 2) to look further into how artificial intelligence (a.k.a. machine learning) might replace testers. My conclusion was that, at least in the medium term, tools will support, rather than replace, testers (see Reference 3). With this background, I started to think about how a tool-supported hybrid exploratory/structured approach could work.

Right from the start, I thought the best use of a tool would be as a “paired tester”. Pairing in testing can be highly effective, with two people working together, exchanging ideas and taking turns to “drive” the system under test (SUT). Could the bot listen to human speech, understand and speak back to the tester? I found some Python libraries that made incorporating both speech recognition (SR) and text-to-speech (TTS) into programs quite easy. SR and TTS would allow the “bot” to communicate, but would it be possible for the bot to simulate a human tester?

The bot needs to capture the voice of the tester and record narrative text for sure, but I also want to capture “structured” data that locates the testers in the SUT and record detail about system features, questions, ideas for tests and defect or bug reports.

One of the problems I see with a lot of exploratory testing is that testers create

mental models of the SUT, but they don’t usually capture those models in a reusable or shareable format (although mind maps are used by many people). The bot must create a system model which can be navigated and re-used.

Obviously, the bot could not do everything a human pair can do. Certainly, the bot could capture an initial survey of the SUT and to capture the outcomes and interpretation of tests. But a desktop or web application would be more appropriate for doing test design, management and reporting, of course.

What also became clear was that the initial exploration of the SUT is a kind of survey. Surveying, as a metaphor for exploring and modelling (the left-hand side of the New Model) fits. It also avoids the confusion with “traditional” exploratory testing, so I will use the term “surveying” to reflect what testers do when they learn how the SUT should behave.

Location hierarchy

Now, if the bot is to capture a model or – perhaps better – a map of the SUT, what should that map look like? The purpose of the map is to easily locate features of the SUT and to assign our notes, observations, questions and, of course, tests to identifiable places. Many testers find mind maps useful in this respect. A mind map is essentially a hierarchy, so I decided to capture the map as a hierarchy of locations.

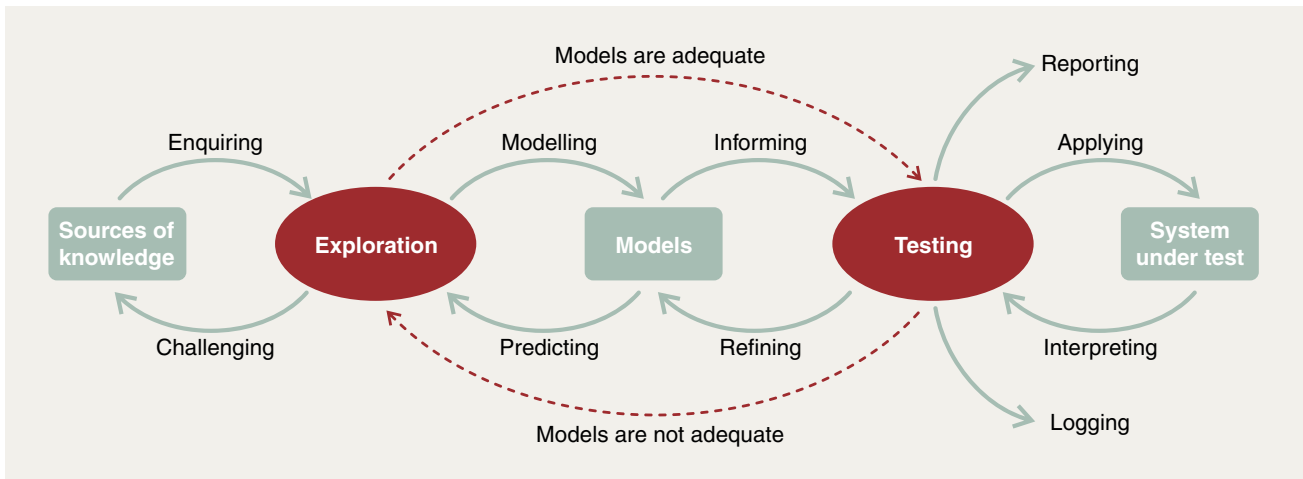


Figure 1. Ten thinking activities that correspond with physical tasks.

Place	The highest level. Represents a sub-system, a module or component within the SUT.
Feature	A feature represents an identifiable and piece of functionality that a user would recognise. It might be accessed through a menu option and could be a web page or mobile app screen.
Form	A form represents an executable transaction or action that can be tested independently. A feature such as a web site home page might have multiple forms on it. Note also that a form maps directly to a web service if you are testing through an API.
Field	A field is an observable place on a screen where a data item is displayed, and/or can be entered. Note the fields can be hidden or implicit (e.g. the content of cookies or database values). For the purpose of testing, form buttons and the outputs of a transaction or call to a service are defined as fields.

Table 2. The four-level location hierarchy within a system under test

The location hierarchy has four levels within a SUT, as you can see in Table 2, below. Places are at the top level and Fields are at the lowest level.

Given this hierarchy, when a tester surveys and creates the map of the territory, they can log ideas, queries, observations or risks to locations at any level. The map and all the notes captured by testers are recorded and can be shared. Tests of forms use the form fields as placeholders for test inputs (and outcomes).

Architecture

There are three main components of the tool, codenamed Cervaya.

Schema Server

This server maintains what is, in effect,

a state model for the bot; there is a web interface to the schema server to maintain the state model. Note that, in principle, schemas for other applications are possible; for example, home automation/IoT.

The bot downloads the schema from the schema's server when it starts up. The schema is the configuration of the bot, instructing it to accept and act on some commands and not others, to call web services on the Cervaya server, and so on. The tester using a bot would not interact with the schema server.

Cervaya bot client

The bot is a Python program that includes the Robot Engine (a Python module). There are two versions of the client program. One has a command line interface and the other manages the SR and TTS interfaces (see References 4 and 5). The two interfaces are identical in their functionality: the command line accepts text commands; the SR interface accepts speech commands.

Cervaya Server/Web Application

The Cervaya server is a conventional website that hosts the web services used by the bot and a SaaS web application used to manage the data in the system. The web services provide simple enquiry, add and update facilities used by the Cervaya Bot Client. The web application manages user and team administration, projects, application models, environments, app versions, chartering and reporting.

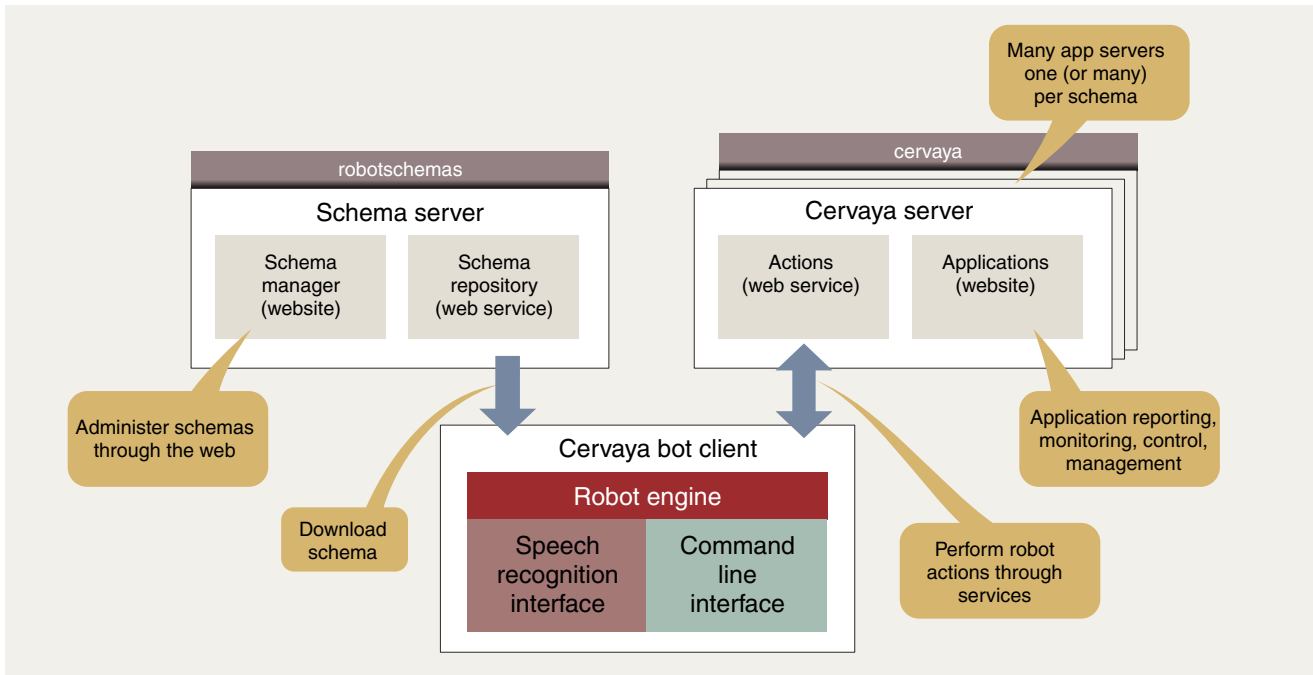


Figure 2. The architecture of the Cervaya Bot

The web app gives the test team shared access to the system model, all notes, queries, tests and results. It also provides some simple graphical navigation facilities, in particular, a navigable hierarchy of locations in the system model.

Summary

Testers can create, manage and share a system model. The testing bot supports system surveys which means that the system model and all notes, queries and defects can be captured using just voice commands. Although tests can be captured as narrative text descriptions using the bot, the web application can be used to capture full test case definitions for documentation purposes (with the potential to export test automation scripts).

Speech recognition is implemented using the Google Speech Recognition API (see Reference 6). Although it is easy to set up, it can be very slow to respond. Also, although it works well in quiet room, in a normal office environment it makes far too many errors. We intend to replace the bot and Google API with an Echo Dot (see Reference 7). The Echo Dot hardware and the Alexa AI services from Amazon are easy to set up and use. It is accurate and fast, even in a noisy environment. The Alexa voice recognition API can be

configured to have pre-defined “skills” that interact with public web services. We will use an Echo Dot as the user interface to the bot.

This article introduces the New Model for Testing and an architecture for a bot-

supported test process. The second article will describe how the prototype bot we’ve created can be used in practice, and how the test process is affected. I will suggest how we need to change our test process to take advantage of a robot test partner ■

References

1. New Model for Testing, Paul Gerrard, <http://dev.sp.qa/download/newModel>
2. Match 5 - Google DeepMind Challenge Match: Lee Sedol vs AlphaGo, <https://www.youtube.com/watch?v=mzpW10DPHeQ> (final game, award ceremony, press conference)
3. The Future of Tools in Testing, <https://tkbase.com/resources/viewResource/14>
4. <https://pypi.python.org/pypi/SpeechRecognition/>
5. <https://pypi.python.org/pypi/pyttsx>
6. <https://cloud.google.com/speech/>
7. Amazon Echo Dot, <https://www.amazon.co.uk/Amazon-Echo-Dot-Generation-Black/dp/B01DFKBL68>

Paul Gerrard is principal consultant at Gerrard Consulting.

Testing Talk

Anthea Whelan asks Andreas Golze, passionate fan of DevOps, about the enhancements that accompany embracing this approach.



Integrating the complete non-functional testing phases into the overall process is seen as a core best-practice.

Who should be interested in embracing DevOps?

Large enterprises, in particular, but any business that requires faster releases, with good feedback – those that are struggling to implement effective communication and collaborations between their IT and business teams.

By emphasising “people and culture”, DevOps instigates a cultural shift that requires a balance between the current market perception of an organization and how the business is envisioned in the future.

That sounds like a challenging leap for some companies to make. What's stopping them?

As more and more companies are seeking to provide digital offerings as part of their product suites, speed of deployment becomes a correspondingly important factor. Businesses born in the cloud era are automatic adopters of DevOps, a culture of practices that matches their DNA. Older businesses, with their more siloed practices, have more legacy systems they depend upon, some of them extremely complex, isolated and bound to their own protocols. Coupled with long-defined development, operations and testing processes and practices, it can be difficult to meet the varied needs of modern enterprises.

What would you advise as a good first step, for those wishing to make the leap?

Continuous Testing (CT) should be adopted without delay. It provides continuous feedback that drives software delivery through the entire development cycle. Automated feedback at each checkpoint works like a feeding mechanism for the

next process in the delivery chain, if the feedback says to move forward.

If the feedback does not say to move forward, then the entire process can be immediately halted so corrective measures can be taken.

Also, the automation code base should be treated just like an application code base – it should reside in its own version control repository. The automation suite must be integrated with the build deployment tool. This enables centralised execution and reporting.

If the automation suite is then categorised into layers of automated tests – build test runs; health checks; smoke tests; full-scale regression tests – then regression execution can be carried out overnight, depending on how frequently you want to put out a build, or during the weekend if you find that the CT setup is becoming less effective due to longer feedback cycles.

That all sounds just a little too easy! Automate everything and it will all work out?

No, of course not! There are always challenges. Some tests built on commercial tools can often slow down over time due to tool architecture. Also, automation is often built using different tools – one for user interfaces; one for APIs and mobile coverage, for example.

There are workarounds, however. Existing automation and migration tests should be executed using more open source tools, such as Selenium, which automates web browsers across many platforms. This helps to further enhance the effectiveness of DevOps and ensure it is more integrated with development tools.

What about non-functional testing?

There are known challenges for integrating non-functional tests into the CT process. Load testing, for instance, raises the issues of non-availability of dedicated servers to generate the desired user load, or capacity constraints impacting the ability to scale the CT environment and sustain the size of load tests, or a lack of on-demand tools to identify the bottlenecks.

Again, cloud-based infrastructures allow organizations to make effective use of available resources and establish the capability to run these load tests on-demand, plus tools such as Apache JMeter are quickly emerging as invaluable for performance testing by organizations using DevOps practices.

It is the core philosophy of CT processes to test every single change made to the application, as early as possible. If non-functional tests are not included in the overall CT process, the organization is only solving part of the puzzle. If these issues are not tested for and identified as early as possible, it may affect improvements in functionality, and those may not be reversible. You could put your entire release schedule in jeopardy. Integrating the complete non-functional testing

phases into the overall process is seen as a core best-practice.

What are the DevOps habits of the most effective businesses?

Agility in the ability to execute tests is the key to successful CT. New techniques in test data management and service virtualisation are emerging all the time and can only enhance the effectiveness of DevOps. Automated processes which are also providing meaningful metrics.

Enhancing an existing DevOps set-up by driving automation through open source tools, or by using a flexible commercial tool that integrates well with both upstream and downstream activities. Some have found the “Monday to Friday – Weekend” model quite effective, where teams focus on continuous quality via greater automation during the weekday build and test phases, while running automated regression tests for the final QA tests over the weekend.

A well-implemented DevOps strategy complemented by an intelligent test automation framework puts businesses in a strong position to benefit from the effective collaboration between all of their information technology professionals ■

Andreas Golze is the VP of Quality Engineering and Assurance for Europe at Cognizant.

A close-up, slightly blurred photograph of a Christmas tree. The tree is covered in green needles and is decorated with numerous warm white lights that are glowing. Various ornaments are visible, including a red knitted hat ornament in the center, a white birdhouse-like ornament on the left, and several white beaded garlands hanging vertically. Other smaller ornaments like a red angel, a white snowflake, and a small white figure are also visible.

Happy New Year

With thousands of subscribers, make sure
that your solution is showcased in 2017.