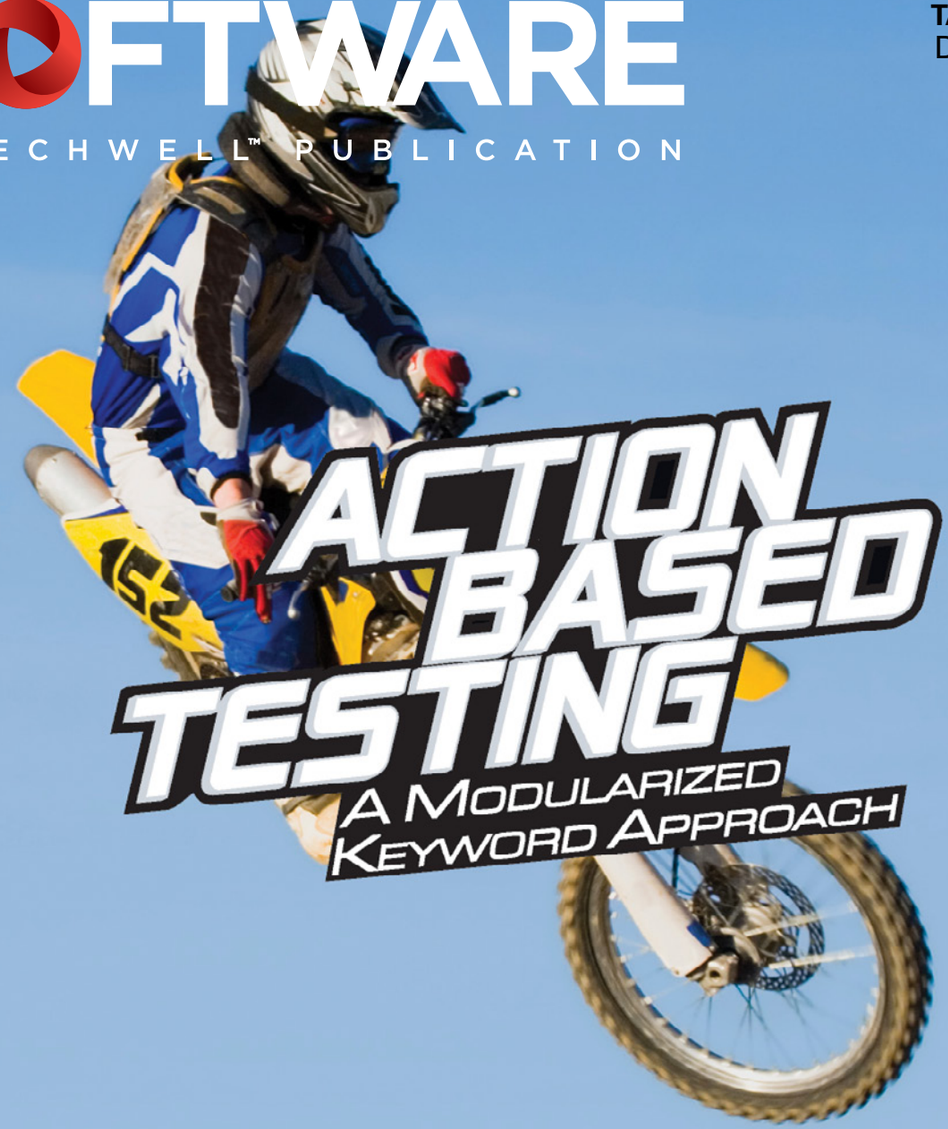
March/April 2011 $9.95 www.StickyMinds.com

BETTER SOFTWARE™

A TECHWELL™ PUBLICATION

DRAWING A BLANK?
3 points to help you strategize

TABLE AMBIGUITY
Decrease defects, increase flow



ACTION BASED TESTING

A MODULARIZED KEYWORD APPROACH

BETTER SOFTWARE™
A TECHWELL PUBLICATION
Volume 13, Issue 2 • March/April 2011

CONTENTS

features

BETTER™ SOFTWARE
A TECHWELL PUBLICATION

Publisher
Software Quality Engineering, Inc.

President/CEO
Drew Thoeni

Vice President of Publishing
Holly N. Bourquin

Editor in Chief
Heather Shanholtzer

Editorial

Managing Technical Editor
Lee Copeland

Online Editor
Joseph McAllister

Production Coordinator
Cheryl M. Burke

Design

Creative Director
Catherine J. Clinger

Advertising

Director of Sales
Sonia Lavin

Customer Success Manager
April Evans

Circulation and Marketing

Circulation Coordinator
Jamie Green-Gago

Product Marketing Manager
Diara A. Sullivan

CONTENTS

columns

in every issue

MARK YOUR CALENDAR

SQE TRAINING
www.sqetraining.com

software tester certification
www.sqetraining.com/certification

April 5–7, 2011
San Francisco, CA
Raleigh, NC

April 11–13, 2011
Boston, MA

May 1–3, 2011
Orlando, FL

May 3–5, 2011
Bethesda, MD

May 10–12, 2011
Portland, OR

May 16–18, 2011
Chicago, IL

training weeks
www.sqetraining.com/public

April 11–15, 2011
Boston, MA

May 16–20, 2011
Chicago, IL

conferences

STAR*EAST* 2011
Software Testing Analysis & Review
www.sqe.com/stareast
May 1–6, 2011
Rosen Shingle Creek
Orlando, FL

Better Software Conference
www.sqe.com/bsc
June 5–10, 2011
Caesars Palace
Las Vegas, NV

Agile Development Practices West
www.sqe.com/adpwest
June 5–10, 2011
Caesars Palace
Las Vegas, NV

STAR*WEST* 2011
Software Testing Analysis & Review
www.sqe.com/starwest
October 2–7, 2011
Disneyland Hotel
Anaheim, CA

Agile Development Practices East
www.sqe.com/adpeast
November 6–11, 2011
The Rosen Centre
Orlando, FL

CONTACT US
Editors: editors@bettersoftware.com

Subscriber Services:
info@bettersoftware.com

Phone: 904.278.0524, 888.268.8770

Fax: 904.278.4380

Address:
Better Software magazine
Software Quality Engineering, Inc.
340 Corporate Way, Suite 300
Orange Park, FL 32073

PRINTED WITH SOY INK

Microsoft

Unreal Speed Made Real

Deliver solutions with incredible velocity.

Time-to-market is critical to driving business forward. Put your solutions in customers' hands faster using the total team toolset of Microsoft® Visual Studio® 2010.

VISUAL STUDIO 2010 SPEEDS DEVELOPMENT WITH TOOLS FOR:

- **Real-time visibility** into project quality and status
- **Increased collaboration** through a common interface
- **Concurrent coding and debugging** by incorporating test early
- **Eradicating "no repro" issues** with rich, actionable bugs
- **Empowering manual testing** and automating rote tasks
- **Provisioning virtual labs** for efficient test and build
- **Eliminating waste** with agile project management

Help eliminate waste and accelerate collaboration in your development and test processes. **Visual Studio 2010** integrates test tools into the development environment, unites team workflow, and can help save time and money.

EXPLORE AND EXPERIENCE THE POWER.

Microsoft® Visual Studio® 2010

Get proof at **www.almcatalyst.com/test**.
Buy at **www.microsoft.com/visualstudio**.

Contributors

International expert and author Hans Buwalda was the first to present keyword-driven testing, now widely used throughout the industry. Currently, Hans is CTO of LogiGear, where he oversees his action-based testing method and its supporting tool suite TestArchitect. Other concepts he is known for are soap opera testing and agile test development.

Lee Copeland has more than thirty years of experience in the field of software development and testing. He has worked as a programmer, development director, process improvement leader, and consultant. Based on his experience, Lee has developed and taught a number of training courses focusing on software testing and development issues. He is the managing technical editor for *Better Software* magazine, a regular columnist for StickyMinds.com, and the author of *A Practitioner's Guide to Software Test Design*. Contact Lee at lcopeland@sqe.com.

Lisa Crispin is the co-author (with Janet Gregory) of *Agile Testing: A Practical Guide for Testers and Agile Teams* and a contributor to *Beautiful Testing*. A tester on agile teams for the past ten years, Lisa enjoys sharing her experiences at conferences and user group meetings around the world. For more about Lisa's work, visit lisacrispin.com.

Seth Eliot is senior test manager at Microsoft, where his team solves Exabyte storage and data-processing challenges for Bing. Prior to Microsoft, Seth led the digital QA team at Amazon.com to release MP3 download, video on demand, and Kindle support systems. Seth ruminates on testing in production, cloud computing, and other topics at his blog at bit.ly/seth_qa.

The co-author (with Lisa Crispin) of *Agile Testing: A Practical Guide for Agile Testers and Teams*, Janet Gregory specializes in helping teams build quality systems. As tester or coach, she has helped introduce agile development practices into companies and has successfully transitioned several traditional test teams into the agile world. Janet is a frequent speaker at agile and testing software conferences in North America, including the STAR conferences.

Contributors

Mark Jenkins is an emerging leader in the business analysis community and frequently shares the experience gained during the last decade as an active business analyst and BA manager. Mark is also the regional director for the Americas Eastern Region at the IIBA. You can follow Mark on Twitter (@JenkoUK) or contact him via LinkedIn (www.linkedin.com/in/markajenkins).

Dave Liebreich has been a tester for a while, and he loves to ask and answer questions about testing. He is currently working for a company that makes software for the insurance industry. You can reach Dave at dave@davelieb-reich.com and follow him on twitter as @ATestGuy.

Founder and CEO of Net Objectives, Alan Shalloway (alshall@netobjectives.com) has developed training and coaching methods for lean-agile that have helped his clients achieve long-term, sustainable productivity gains. He is a popular speaker as well as a trainer and coach. Alan is the primary author of *Design Patterns Explained: A New Perspective on Object-Oriented Design*, *Lean-Agile Pocket Guide for Scrum Teams*, *Lean-Agile Software Development: Achieving Enterprise Agility*, and *Essential Skills for the Agile Developer*.

Ken Pugh (ken.pugh@netobjectives.com) is a fellow consultant with Net Objectives. Since the start of the millennium, he has worked with teams to create software more effectively with lean and agile processes. Ken has written seven books, the latest being *Lean-Agile Acceptance Test-Driven Development*. In 2006, his book *Prefactoring* won the Jolt Award. In his spare time, Ken snowboards, windsurfs, and backpacks.

With more than twenty-five years of management experience, Robert Sabourin, P. Eng, has managed, trained, mentored, and coached hundreds of top professionals in the field. He frequently speaks at conferences and writes on software engineering, SQA, testing, management, and internationalization. Robert is the author of *I am a Bug,* the popular software testing children's book; an adjunct professor of software engineering at McGill University; and the principle consultant (and president/janitor) of AmiBug.Com, Inc. Contact Robert at rsabourin@amibug.com.

Skills Are in the Eye of the Job Holder

I consider myself a pretty good friend to the Earth, but, in an earlier professional life, I protected the environment in an official capacity. My first undergraduate degree is in biology. After graduation, I went to work for the state environmental protection agency in the air quality division. I was responsible for collecting data from and maintaining and calibrating the air monitoring machines in our district.

Eventually, I went back to school and got a degree in communications. When I applied for a job with Software Quality Engineering (publisher of *Better Software* magazine), part of what made me a competitive candidate for the position was my experience working as an environmental specialist/field technician.

You may wonder how the skills I learned working in environmental regulation were transferrable to a job in publishing, but the two are not as different as you might think.

Because our air quality data was audited by the national Environmental Protection Agency, there were strict rules to follow and minute attention to detail was mandatory. I had to follow strict protocol and standard operating procedures—not unlike the way, as an editor, I must adhere to grammar rules and follow the guidelines laid out in the *Chicago Manual of Style*. And, an editor who doesn't have an eye for detail would be pretty ineffectual.

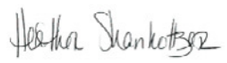
In this issue's Career Development article, "Skills for Agile Testers," Lisa Crispin and Janet Gregory discuss skills testers need to develop to be of the most value to their teams. And—as with my bringing my biology background to play in my editing career—some of those skills aren't as tester-y as you might think.

Also in this issue, Hans Buwalda introduces his newest innovation: action based testing (ABT). ABT emphasizes modularized test design, making tests lean and allowing for stable and maintainable automation.

Ken Pugh and Alan Shalloway help us cut down on looping and increase the flow of working software to the customer with table-driven acceptance tests. And, Seth Eliot explains how to overcome the challenges cloud computing introduces to development, testing, and your team.

As always, I hope you enjoy this issue of *Better Software* magazine. Email me to let me know how you've put the content to work for you.

Happy reading,



Heather Shanholtzer
hshanholtzer@sqe.com

The Third Age of Requirements

As in the early days of software development, programmers lead the way. This is the name of the game.

by Lee Copeland | lcopeland@sqe.com

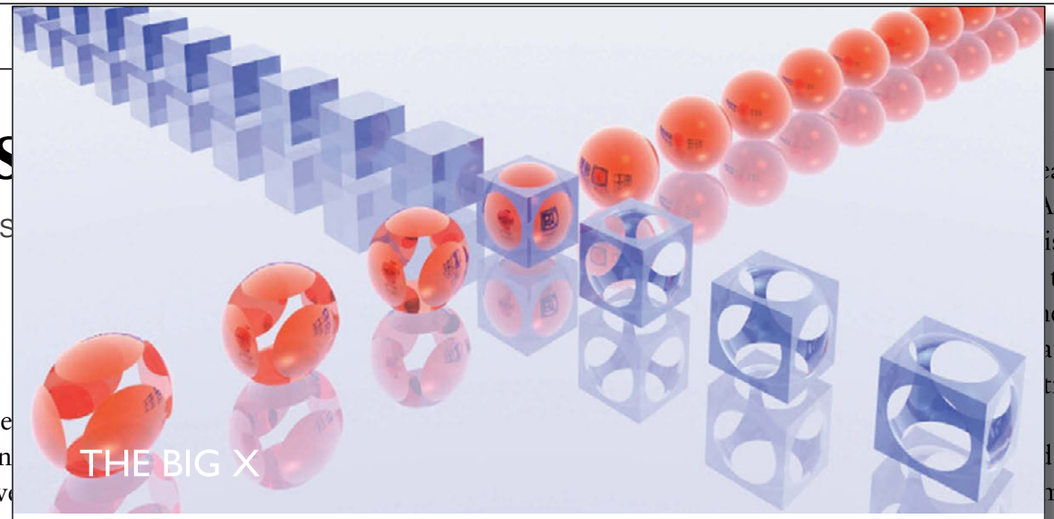
I began my programming career during the First Age of Requirements. In those days, programmers ruled the Earth. We decided what the systems would do and wrote the code to make them do it. And, we were beloved for it. On those few occasions when we asked customers what they wanted the system to do, they would typically reply, "Oh, you computer people are so smart. You tell me what I need." And we—inflated egos and all—were happy to comply.

However, as time went on, more and more of what we built was not what the customer wanted or needed. As systems became more sophisticated, the programmers' understanding of the complexities of the business was not sufficient to guide them in their work. So emerged the Second Age of Requirements—the age of requirements elicitation. This age was based on the belief that customers knew what they needed and could and would tell us, if only we asked properly. Some elicitation methods used individual stakeholder interviews (typically followed by a synthesis process in which the requirements analyst would abandon some stakeholders' requirements in favor of others to meet various constraints). Another method used a joint requirements planning meeting during which stakeholders and their requirements battled for supremacy and a place in the requirements specification. Many experts believed these "out" requirements [1] like beaters on of a more gentle nature believed w requirements [2]. Requirements sp forms: the classic 500-page documen the IEEE 830-1998 standard, prototy cases, and now use

> **"…as time went on, more and more of what we built was not what the customer wanted or needed."**

Now, as the per ourselves in the Th Calling for an end Gilb wrote, "The imagine is when w users, and our ow 'functions and fea carefully disguised ments.' If you go s of these 'false req mediately find that they are not reall **really bad amateur design** for the 'rea Haug Kogstad, executive vice preside echoes the same sentiments: "The wo do what the customers ask."

The Third Age is much like the charge again. However, now we're processes. Rather, we're creating tota



X as a service (XaaS) has now entered the official business dictionary. You can now replace X with any product or service as long as it can be priced, delivered or consumed over the net. The most popular ones are Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). Of course there are other 'business process' based inclusions like Collaboration as a Service (CaaS), Analytics as a Service etc. The category of Business Process as a Service (BPaaS) creates huge opportunities for service providers in multiple ways.

One may ask what it takes for something to become an X! To answer this question we need to go back to basics as to what benefits the XaaS model provides the customer …

- A Rental model - pay per duration or usage
- Automatic 'self-provisioning' of services as per customer requirements
- Scalable and elastic

So, it may appear that any product, service or productized service that can be delivered over the net is probably a good candidate for 'X'. Far from it! The following criteria are critical:

- X must be clearly defined in customer 'deliverable' terms.
- Service parameters must be well defined in order to scope and size the service unambiguously
- Resources required to deliver X must be shareable across multiple customers
- Cost and cycle time of delivery must be kept low through high levels of automation

So let us evolve the concept and practice of 'Testing as a Service' from these first principles

A Business Function, Service or activity?

In the past decade testing has evolved. First, testing is no longer considered to be 'yet another phase' in the development cycle to be planned and executed by the development team itself. It has an independent identity. An independent testing team is considered to be an asset and critical to ensuring quality. Second, testing has emerged with specialized sub-streams like performance testing, package application testing, test automation etc.

Is testing an activity or a business process? Or is it a service? All are valid based on the mandate given to the testing team. To be considered a business process, testing has to redefine itself in business terms. There are so many definitions of testing like 'increase quality', 'reduce defects etc. and they may all be valid. However it is important to re-define testing in higher terms that has business relevance.

For example, defect analysis has to evolve to risk analysis. Test completion criteria should be defined in terms of purpose readiness criteria. Rather than dictating how much time and cost we need for testing, we need to be prepared to exercise tradeoffs based on business constraints. Rather than mere measurement of test productivity (how fast), we need to view the problem as how to improve 'directivity' (right direction) towards deployment milestone.

The

As in the e

is the nam

by Lee Cope

I began my p
quirements. I
cided what th
them do it. An
when we aske
they would t
smart. You te
inflated egos a

However,
more of what
tomer wanted
more sophist
derstanding c
ness was not
work. So eme
ments—the a
This age was
tomers knew
if only we ask
vidual stakeh
process in wh
stakeholders'
constraints).
ning meeting
battled for su

In business terms, testing needs to be re-defined as a function that provides assurance of business purpose readiness. In case of IT Systems this translates to assurance of production readiness and in case of a product, it is market launch readiness. Not only that, the testing function must play a continual advisory role in navigating towards the deployment milestone throughout the program duration. With such a definition, mandate, process, culture and organization direction, it becomes relatively easier for testing to be positioned as a business function.

In summary this redefinition exercise implies:

- Assurance of business purpose readiness (rather than just test execution)
- Risk and vulnerability analysis (rather than just defect analysis)
- Focus on purpose readiness criteria (rather than just test completion)
- An advisory role (rather than just defect reporting role)
- Trade off decisions (rather than order taking approach)
- Risk driven effort (rather than just estimated size driven effort)
- Direction driven (rather than just productivity driven)

It may be right to say that testing can indeed be positioned as a business function that offers a set of services.

Now for services to be 'browsed, bought and consumed' online, they need to be tangible and predictable so that the customer can make the right buying decisions.

Let us now examine the concepts and practice of service productization that are closely linked.

Service Productization

Every service industry is 'productizing' itself. In the early stage of evolution of any service industry, the customer is expected to clearly articulate the service tasks or activities to be performed and the service provider provides it for a fee. Examples are ticket booking, opening a bank account etc. As the industry matures, the customer seeks solutions to problems and is not content with mere task fulfillment. So the ticketing tasks of the travel industry have evolved to travel packages. Banks are offering 'products' that bundle several transactions and services. And it is seen that most problems that customers in any service industry face usually follow a set pattern with minor variations from customer to customer. This implies that the solutions are also more or less common and minor customizations can satisfy a customer's demand.

So, productization means the following:

- Pre-defined services - the customer now clearly knows the activities performed and the solution benefits expected
- Pre-developed tools and methodologies - pre-defined services will be delivered by using pre-developed automation tools to reduce cycle time and cost of delivery
- Pre-trained delivery team - a team is pre-trained on the tools and methodologies mentioned above to deliver the pre-defined services

Benefits of productization

- Clear definition of outcomes ensure the services are more 'tangible'
- Predictability
- Clear connection with the customer's problem
- Provision to aggregate or bundle multiple service products to provide value for money
- Shorter sales cycle

Service Productization in Testing - Testing-as-a-Service

How do the above apply to Testing Services? If we agree that the end outcome of testing is assurance of business purpose readiness, then the testing service portfolio should reflect such an outcome.

Testing services offer immense opportunities for productization provided the customer's problem definition and the solution from a testing standpoint are clearly defined. Here are some examples:

- Certification Services - assurance that the system or the product conforms to specified international standard(s)
- Qualification Services - assurance that the system or the product fulfills certain specified criteria that have significant financial, regulatory, compliance or market implications
- Test Environment and Lab Services - providing a test environment, labs and operational services
- Readiness Assurance Services - Assurance that the system or the product meets specific business readiness criteria like production readiness, market readiness, network readiness etc.

The key success factors in testing services productization are as follows:

- Clear Service definition - often service definition involves abstracting multiple testing tasks into a service statement. This is a critical factor
- Identification of testing parameters that will help sizing and scoping to enable service linked pricing of the services
- Facility to publish a catalogue of testing service products so that customers can browse these services, choose test parameters and request for a proposal, pay and consume online
- High degree of test automation - a big technical challenge since current automation paradigms do not work without the system implementation being in place.

Proven eCommerce and eBusiness principles can also be applied in building the Testing-as-a-Service (TaaS) portal. The portal implementation can be done in the following phases:

Phase I: Host the testing service catalogue with provision to select service parameters and request for a proposal - the proposal and the service delivery will be through an offline channel.

Phase II: Add a provision to get an online commercial quote on services - service delivery through offline channels.

Phase III: Add provision to make online payment after due authentication - service delivery through an offline channel.

Phase IV: Add provision to 'consume' select services online.

Integration with a company's CRM systems can also be planned at some point in the evolution of the portal.

Conclusion

Testing services can be positioned as a business function, offering a set of services, provided the organizational mandate, orientation and processes are appropriately repurposed. Testing can then be delivered as a service over the 'cloud' in the TaaS model. Service productization and ecommerce concepts can be suitably applied to improve customer acceptability and business success of the initiative.

Authors: **Sanjay Seth** - GM, Marketing & Alliances Head, Wipro Testing Services
Mahesh Venkataraman - GM, Innovation Head, Wipro Testing Services

CLOSE WHITE PAPER

ts

his time, innovation

ese processes would "drive
on the African veldt; others
we were only "exploring"
specifications took many
nent formatted according to
totypes, UML diagrams, use
user stories.
pendulum reverses, we find
Third Age of Requirements.
nd to the Second Age, Tom
The worst scenario I can
n we allow real customers,
own salespeople to dictate
features' to the developers,
ised as 'customer require-
o slightly below the surface
requirements,' you will im-
eally requirements. They are
'real' requirements" [3]. Per
sident of Tandberg Telecom,
worst thing you can do is to

the first—developers are in
're not automating existing
otally new, innovative prod-

ucts—products the customers couldn't have envisioned, that they didn't know they couldn't live without until we created them, and that emphasize an exciting user experience. Aside from Dick Tracy, the vast majority of customers didn't envision a small, portable telephone they could carry with them; they seemed content to use pay phones when away from home or office. Customers didn't envision small, portable music players that they could load with music they chose; they carried transistor radios (or boom boxes) with them, playing music and commercials a radio station program director preselected for them. At Taste of India, the Indian restaurant near my home, I didn't envision a device that would display the menu, accept an order, relay it wirelessly to the kitchen, and then provide games to keep my grandchildren occupied until their order arrived—making dining out much more enjoyable. (The bunny catching carrots while dodging cans as they fall from the sky is their favorite game to play while waiting for their chicken tikka masala.)

What does the Third Age mean for better software? It unleashes the combined creativity of thousands of developers to create successes such as Linux, Gmail, the iPad, Facebook, Pandora, and Angry Birds. However, to be a contributor to the Third Age, we must learn new programming languages, platforms, processes, and paradigms. Futurist Joel Barker wrote, "When a paradigm shifts, everyone goes back to zero. Your past successes guarantee you nothing in your future" [4]. That's a wake-up call to all trying to create better software. We've got a lot to learn. **{end}**

For more on the following topic go to www.StickyMinds.com/bettersoftware.

- References

The Third Age of Requirements

As in the early days of software development, programmers lead the way. This time, innovation is the name of the game.

by Lee Copeland | lcopeland@sqe.com

I began my programming career during the First Age of Requirements. In those days, programmers ruled the Earth. We decided what the systems would do and wrote the code to make them do it. And, we were beloved for it. On those few occasions when we asked customers what they wanted the system to do, they would typically reply, "Oh, you computer people are so smart. You tell me what I need." And we—inflated egos and all—were happy to comply.

However, as time went on, more and more of what we built was not what the customer wanted or needed. As systems became more sophisticated, the programmers' understanding of the complexities of the business was not sufficient to guide them in their work. So emerged the Second Age of Requirements—the age of requirements elicitation. This age was based on the belief that customers knew what they needed and could and would tell us, if only we asked properly. Some elicitation methods used individual stakeholder interviews (typically followed by a synthesis process in which the requirements analyst would abandon some stakeholders' requirements in favor of others to meet various constraints). Another method used a joint requirements planning meeting during which stakeholders and their requirements battled for supremacy and a place in the requirements specification. Many experts believed these processes would "drive out" requirements [1] like beaters on the African veldt; others of a more gentle nature believed we were only "exploring" requirements [2]. Requirements specifications took many forms: the classic 500-page document formatted according to the IEEE 830-1998 standard, prototypes, UML diagrams, use cases, and now user stories.

> "...as time went on, more and more of what we built was not what the customer wanted or needed."

Now, as the pendulum reverses, we find ourselves in the Third Age of Requirements. Calling for an end to the Second Age, Tom Gilb wrote, "The worst scenario I can imagine is when we allow real customers, users, and our own salespeople to dictate 'functions and features' to the developers, carefully disguised as 'customer requirements.' If you go slightly below the surface of these 'false requirements,' you will immediately find that they are not really requirements. They are **really bad amateur design** for the 'real' requirements" [3]. Per Haug Kogstad, executive vice president of Tandberg Telecom, echoes the same sentiments: "The worst thing you can do is to do what the customers ask."

The Third Age is much like the first—developers are in charge again. However, now we're not automating existing processes. Rather, we're creating totally new, innovative products—products the customers couldn't have envisioned, that they didn't know they couldn't live without until we created them, and that emphasize an exciting user experience. Aside from Dick Tracy, the vast majority of customers didn't envision a small, portable telephone they could carry with them; they seemed content to use pay phones when away from home or office. Customers didn't envision small, portable music players that they could load with music they chose; they carried transistor radios (or boom boxes) with them, playing music and commercials a radio station program director preselected for them. At Taste of India, the Indian restaurant near my home, I didn't envision a device that would display the menu, accept an order, relay it wirelessly to the kitchen, and then provide games to keep my grandchildren occupied until their order arrived—making dining out much more enjoyable. (The bunny catching carrots while dodging cans as they fall from the sky is their favorite game to play while waiting for their chicken tikka masala.)

What does the Third Age mean for better software? It unleashes the combined creativity of thousands of developers to create successes such as Linux, Gmail, the iPad, Facebook, Pandora, and Angry Birds. However, to be a contributor to the Third Age, we must learn new programming languages, platforms, processes, and paradigms. Futurist Joel Barker wrote, "When a paradigm shifts, everyone goes back to zero. Your past successes guarantee you nothing in your future" [4]. That's a wake-up call to all trying to create better software. We've got a lot to learn. **{end}**

For more on the following topic go to www.StickyMinds.com/bettersoftware.

- References

Filling the Blank Page

It's difficult to analyze a situation without sufficient information. These three points will help you understand goals and create a strategy.

by Mark Jenkins | mjenkins@bigbearsystems.com

Picture the following scenarios: Your manager or project manager has assigned you a project with only a brief overview of the situation, immediately followed by a query into the amount of time it will take to gather the requirements. Or, imagine that you arrive at a meeting and, as you are being introduced as the business analyst (BA), the meeting leader relinquishes the lead to you for the purpose of gathering or documenting the requirements. As BAs, many of us will have uncomfortable memories associated with these two scenarios.

I'd like to share some techniques I've developed to help reduce what I call blank page syndrome (BPS). BPS occurs when you are asked to analyze a situation without sufficient information. The result is an inability to visualize the circumstance because the situation is without context. In the past, without a strategy to lead my thinking, I adopted a "scattergun" approach, meaning I covered a wide range of questions in a few minutes. The scattergun questioning confused my audience because, to them, the questions seemed unrelated. Confusion affected their responses, which impacted my ability to define scope. As a result, my time estimates were infrequently accurate, and I wasn't able to extract all the information needed. I also regularly found myself working many early mornings and late nights to compensate.

Now, let us fast forward to the present. I have learned three important things. Please note that the order is important, so I have assigned each a number.

1. If I understand the need or goal of a project, then connections to a solution, conflict guidance, and scope control are improved.
2. If I understand the basic flow of how a process or system works, then I have a context and framework within which to better understand, direct, and document a project.
3. My estimation of time is only accurate if I divide a project into small chunks of effort and then add them together.

By combining these three points into a cohesive strategy and framework, I have seen my career extend into the enterprise arena and management. For the BAs I have trained in this approach, I have seen a similar progression. This approach quickly changes the perception of a BA from documenter to thought leader. So, let's begin to understand these points in detail.

I recently heard a great quote that encapsulates the idea behind point number one: "There is no point doing efficiently that which should not be done at all." As BAs, we often find ourselves wondering why a project is being undertaken, yet we are unsure of the appropriateness of questioning its purpose. It would seem that a logical solution would be for BAs in this circumstance to question a project's purpose; however, hierarchy of roles and preconceived stakeholder analyses of a problem can often quash any BA-querying opportunities.

To understand how the above situation occurs, let's think for a moment about the mindset of a typical business stakeholder. Broadly speaking, a stakeholder has two types of projects: projects that try to overcome a challenge and projects that try to exploit an opportunity. In a challenge situation, the stakeholder will recognize the challenge and attempt to resolve it with increasing levels of irritation. Realizing that he is unable to resolve the problem, the stakeholder will attempt to find a workaround. Having created a viable workaround for the problem, the stakeholder then will seek a permanent solution. In seeking a permanent fix for the problem, the stakeholder will call IT for help, requesting specific "fixes" rather than explaining the problem.

> "In a challenge situation, the stakeholder will recognize the challenge and attempt to resolve it with increasing levels of irritation."

The path of opportunity follows a similar route. The stakeholder recognizes an opportunity and typically devises an immediate solution to assist with the pursuit of that opportunity. If the opportunity is realized, the stakeholder will be overwhelmed with workload. This is a classic symptom that we see in startups; as businesses grow, their processes and systems fail to keep up with demand.

As a BA, it is vital to recognize the stakeholder's position on a path. The farther along he is, the harder it is for the stakeholder to articulate the real issue, business driver, or opportunity. Lack of accurate information impedes the BA's ability to complete a project that provides true business value, and it makes it harder to suggest project direction. Over time, your goal should be to influence

stakeholders to seek BA involvement earlier in the path. The closer to the origin of the path your involvement begins, the greater your opportunity to add value.

How do you influence the stakeholder to seek your involvement from the get go? By asking appropriate, relevant, objective questions free from assumption. For example, during a visit to the doctor's office, the opening conversation begins with questions like "Why are you here today?" or "What seems to be the problem?" As the patient, we know little to nothing about medicine, so the doctor must extract information that will assist with diagnosis. Questions like the two listed above immediately get to the point and expose the mood of the patient and the accuracy of her self-diagnosis (indicating her position on the path). During your first meeting with a stakeholder on a challenge path, consider opening the dialogue by asking, "So, why are we meeting today?" and following up with deeper probing questions like "Why is this a problem for you?" and "How are you impacted by it?" and "If we do nothing, what would happen?" For a stakeholder on a path of opportunity, ask questions like "How do you anticipate impacting revenue (or cost or efficiency)?" and "How will you measure that impact or success?" As you progress through your project, try to tie everything back to the challenge or opportunity. If a requirement can't be mapped to a challenge or opportunity, then you might ask a stakeholder, "Is this requirement still necessary for you, given that it does not directly relate to the challenge or opportunity?" This provides occasion to control scope and project costs.

Let us move on to point number two: understanding the flow. An understanding of the flow of a process provides context at both the specific, detailed level and the broader enterprise level. Developing this understanding also provides substantial potential for the BA to structure the stages of the project from elicitation sessions to detailed documentation. I refer to my strategy for developing understanding of flow as the *power verb* technique. For example, taking out the trash is a common task in which we all engage. Although taking out the trash is a basic task, it has a structured flow: *Review* the number of trash cans, debris sort, and pick-up schedule; *prepare* the cans for removal by collecting the trash throughout the house and tying the garbage bags; *remove* the bags to the outdoor bins; *deliver* the bins to the curb; and *reset* by returning the bins after pick-up.

> "An understanding of the flow of a process provides context at both the specific, detailed level and the broader enterprise level."

Another example is the process of relocation. In this situation, the power verbs for a complete and successful house move might include *investigate*, *decide*, *prepare*, *organize*, *pack*, *move*, *unpack*, and *recover*. Often, we overlook scope because we don't consider the end-to-end process; rather, we see only our imme-

diate needs or position. In this way, we all exhibit the stakeholder behaviors in our personal lives.

Let us review an example from the IT arena: moving a data center. At first glance, moving a data center seems complicated, but utilizing the power verb technique we have *assess*, *design*, *plan*, *pack*, *move*, *setup*, and *activate*. Having determined power verbs for moving a data center, we have created context in which to structure the next steps. Further, if I wanted to investigate a CRM system, the verbs *attract*, *qualify*, *contact*, *sell*, *order*, *book*, and *support* would categorize my customer-relationship cycle.

So, how do you construct the power verbs? Start with your own knowledge. There may be easily deduced verbs within the project title or existing documentation that can give you some clues. Recognize that while a process flow varies in shape (e.g., linear, cyclical, etc.), it is imperative to place the appropriate detail within each verb section.

Ultimately, in order to determine and populate the power verbs, you need input from the stakeholder. An effective way to solicit stakeholder input would be to ask something like "Can you walk me through how you do things today?" and "What other groups do you interact with, have input to, or have output?"

Having organized and populated the power verbs, I brainstorm ways to deconstruct them into deeper flows or logical sections. In the CRM system example, the *attract* power verb could be further organized into a subset of power verbs: *plan*, *budget*, *initiate*, *manage*, *track*, and *evaluate*. This subset of power verbs frames a basic marketing campaign that could be incorporated into the system while adding further detail to the process. The key is to break projects down into manageable parts. For most application projects, I add three default power verbs: *secure* (security), *manage* (management of the system), and *report* (reporting). These three power verbs cover the requirements that are generally missed or realized too late.

We are now prepared for point number three: time estimation. Having broken down a project by flow stages and associated detail, the BA's ability to estimate time is significantly increased. Eliciting, documenting, and reviewing requirements for each section designed from the power verb technique provides the BA with a higher degree of accuracy and comfort, information, and context. Now, determining the time allotment for each power verb subsection is a manageable task. By adding the time allotments of each power verb section together while factoring in existing workload, BAs render an accurate timeline for a requirements activity.

Using these three points as an outline for your next project is in alignment with BA best practices, as the points structure an effective business analysis plan (BAP). The BAP becomes a helpful tool to negotiate a project timeline with a project manager, manager, or business stakeholder. **{end}**

Catherine Powell

Years in Industry: **10**

Company Name: **Abakas**

Email: **catherine@abakas.com**

Interviewed by: **Dave Liebreich**

Email: **david@daveliebreich.com**

What do not translate between projects are tools and the domain knowledge, but the skill to learn quickly and well helps overcome the translation.

The exciting thing about testing is also the frustrating thing about testing: the uncertainty.

Getting my hands dirty gives me a good sense of what the system actually does.

> **My ideal** cotester is someone who is not like me.

A good tester has a lot of great skills, but there's no reason to hoard those skills or to be defensive about it.

You cannot assume you will only test once. You will have to test more than you think, or retest things that change after your first test.

TECHWELL™

For the full interview visit
www.techwell.com

Action based testing (ABT) is a pioneer among keyword-driven test automation approaches. It was created in 1994 and is continuously evolving and improving.

In ABT, tests are organized into test modules that look like spreadsheets. In a test module, the tests are written as a series of actions, each consisting of an action keyword and arguments representing input data or expected results. This way of working improves readability of tests, and it tends to save enormously on maintainability because the automation efforts focus exclusively on the action words.

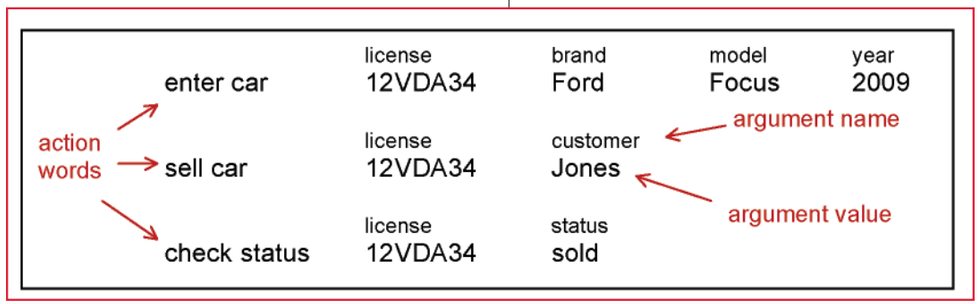


Figure 1

Figure 1 shows the testing of an imaginary car inventory system. Each line is an action and starts with an action word followed by arguments. The first action, entering a car, uses action words "enter car" followed by four arguments. Each argument has a name (and a value): license (12VDA34), brand (Ford), model (Focus), year (2009). The second action describes selling the car to a customer called Jones. The action words are "sell car."

The third action is the check that completes the test. After the car is sold, the status should now be "sold." We call this value of the status argument the "expected value" of the check. You will have to run the test to find out if it matches the actual value.

Note that the three actions of this test specify enough information to understand what is tested from a functional point of view. However, they do not show details on how the test would be executed, like what buttons to click or what fields to fill out in dialogs. The check status action also doesn't specify where to capture the actual value for the status. This value could be captured from a window, but it could also be retrieved from a database with an SQL query. For this test, you don't know, and it doesn't matter.

A test written in this form could easily be executed manually, but normally the execution will be automated. In ABT, the automation effort focuses exclusively on the individual actions, not on the tests. The advantage is that changes in the application under test often can be accommodated by

changing only a limited number of action word implementations.

ABT avoids the verbose descriptions for test cases that are quite common in other testing approaches. The testers work directly in the action format, leaving out all details that do not contribute to the intention of the test. In the previous example, details are left out, such as how a car is to be entered (e.g., which dialogs are needed, what buttons need to be clicked, etc.). The actions solely show the business functionality of entering and selling a car. Yet, the test can be fully automated based on this information alone, assuming that the action implementations take care of details, including supplying default values for fields that are not specified, such as the price of the car.

This does not mean the details of the user interface interaction are always neglected. However, verification of details—like whether the caption on a button is correct—will be done in test modules other than the one testing business functionalities. I consider the concept of test modules to be the core of ABT. The ability to differentiate tests into manageable test modules, each with a clear and unambiguous scope, largely establishes the effectiveness of these test modules as tests and demonstrates the efficiency and stability of their automation.

A test module will typically consist of three parts. In addition to the test cases, the initial and final sections do things like starting up and shutting down the application under test, thus setting up the preconditions for the test cases and cleaning up the test environment after they have finished. The test module will also contain a set of "test objectives," statements that detail the scope of the test module, like "selling a car changes its status to 'sold.'" The test objectives are linked to the test cases in which they are tested. This relation is many-to-many: One test objective can be covered in multiple test cases, and one test case can service multiple test objectives.

The concept of test modules sets ABT apart from various other approaches, even if those approaches also use keywords. Many approaches and test management tools work with a loose collection of isolated test cases. In ABT, the test module, with multiple test cases, is the creative end-product of the test development. The test cases within a test module can be dependent on each other and are expected to be executed in the given order. In turn, the test modules are meant to be independent from each other, so they can be executed in any order. A "global test plan" will define the test modules to be produced. After that, it is the responsibility of the test developer to decide how many test cases to create for each individual test module. This number often varies over time based on growing insights or new ideas related to the scope of the test module and its test objectives.

Action Automation

In ABT, the action words can be automated in three different ways. It is the task of the automation engineer to decide which way works best.

Scripts

The simplest form of automating actions is to interpret them one by one in a playback environment, such as a test tool or a general-purpose programming language like Java or Python. The interpreter essentially is a loop that reads the lines of a test module and, for each line, maps the action to a corresponding script function.

Action definitions

In a good test design, actions come in levels. There are basic actions that do things like click a button or capture a text, and more complex ones like "enter car," which in ABT are called high-level actions. In most ABT projects, we use action definitions to implement many of the high-level actions. Action definitions are sheets similar to test modules, with lower-level actions detailing what needs to happen if the high-level action is encountered in the execution of the test module. A high-level action like "enter car" is written with lower-level actions such as "select a menu item," "enter values in fields," and "click buttons." The actual values, like the name and address of the customer, are passed as arguments to the action definition.

Built-in actions

Commonly used base actions—for example, a UI action like "select menu item"—are usually standard across applications under test. Therefore, it makes good sense to have a library of actions predefined, like we are doing in our own toolset for ABT. Such actions are then called built-in actions. This library of built-in actions can be predefined in a test tool or can be supplied in a corporate environment for use on multiple projects.

The ABT Process

ABT focuses on concrete goals and timelines. The focus is to deliver stable, high-quality, automated tests economically. What follows are the main steps in an ABT project:

Project planning

In this phase, focus on understanding project-level matters (e.g., goals, timelines, budget, organization, environment, risks, assumptions, etc.) Use these to plan the testing project and assemble the team.

After this phase, the activities split into test development and automation. Each has its own timeline, which is mostly independent from the others. While the test developers are defining tests, the automation engineers worry about UIs, APIs, control classes, etc. This work will lead to a set of low-level actions that will be assembled into all the high-level actions the testers need.

Global test design and test development schedule

Directly after or as part of the project-planning step, a global test design should be made. This global test design is mainly a list of test modules to be developed in the remainder of the project. For each test module, establish scope with a clear focus and differentiated from other test modules, stakeholders who are relevant for providing input or assessing the tests and their results, priority, and estimated time.

The test development schedule describes when each test module will be developed and when it will be executed. To define these timelines, a multitude of factors can be used, like external timelines (e.g., overall project planning or sprint schedule), availability of test developers and stakeholders, and complexity and priority.

In my experience, good global test design and derived test development schedules are invaluable tools in controlling the test project. They also allow for flexibility. Changes in the project environment can usually be accommodated by shifting timelines for the test modules.

Test development

The test development in an ABT project is best organized in an agile fashion, even if it is part of a traditional waterfall-type system development project. Regard each test module as a separate mini-project, with two delivery moments: 1) delivery of the test module to a stakeholder or the internal project organization, and 2) first automation or manual execution of the test module. A high level of interaction with the stakeholder is a good way to obtain a strong and efficient test.

Base automation

The automation efforts will typically need to start by addressing the technical aspects of the interfaces in the application under test. For the UIs, that usually means making sure relevant operations can be done on all the control classes appearing in the system under test. The typical areas of concern are third-party controls, grids, trees, context menus, etc. For the non-UI interfaces, attention must be paid to APIs, database access, message protocols, embedded software agents, etc. Handling of graphics and multimedia also needs attention.

How much time these base automation activities will take can depend on circumstances such as the technology used, available tools, expertise, etc. However, the activities usually do not have to wait for any actions to be defined as part of the test development. In our projects, we often do preliminary technical research before or during the project-planning step, so that any issues and bottlenecks are identified in a timely manner.

Action automation

When the base automation is functioning well, the actions can be built as described earlier. The automation engineers have to decide whether to use action definitions or scripts. They design the interface definitions and create and execute tests of their work before making it available for execution of the test modules.

Test execution

The test execution step is the production phase for the test team. Test development and automation engineering come together with the application under test. The test developers, not the automation engineer, should analyze the results of the test execution and follow up on them. A test execution, in essence, compares testers' expectations with actual outcomes in the application under test. This means that "fails" can be misunderstandings in the test or actual faults in the application under test.

Fails should not be due to automation issues. It is the responsibility of the automation engineers to make sure the individual actions work properly, including development of test modules to test each action and the interface definitions involved.

Maintenance

If test modules have been well designed and automation is well structured and tested, maintenance of a test set should not be a big burden. Nevertheless, it is essential to pay attention to organizing the maintenance process. One logical way is to require all tests, including tests of the actions and interface definitions, to pass for each new release of the application under test, making it an integral part of the development and release processes.

I often hear the argument that automated tests become stale and boring after a while, since they are not likely to find new bugs. This tends to be true, but it still makes sense to run the tests routinely. This will keep them up to date and it will help catch "bonus bugs"—unexpected side effects of often-unexpected changes to the application under test. Having a thorough and highly automated test set can be an invaluable contribution to both time to market and quality to market.

Building Success

The components of ABT described so far are only raw building blocks. By themselves they will not lead to success. To make ABT successful, we must consider a number of additional conditions in applying the methods.

Separate test design and automation

If the test project exceeds a certain small size, I recommend against combining the test and automation functions. Creating effective tests and developing automation (actions, interface definitions, scripts) should be separate and well-defined responsibilities. This way, the team members can focus on their tasks and produce good results.

Design good tests

Good test design starts at a high level: identifying the test modules and their scope. One can use numerous criteria to do this, but the key is to pay attention. Even a few days of effort at the beginning of a testing project can make the difference in testing and automation success.

Once the test modules have been defined, each of them becomes a "mini-project." For each module, establish the scope of the test module, the stakeholders, when the test module should be developed, when the test will be executed, who should develop the test module and when, and the best approach to follow to develop the tests.

Planning aspects of the test modules are not set in stone. In many cases, changes in the timelines in the main project, or availability of stakeholders or test developers, cause test modules to be moved around. Having the test modules well defined, with a focused scope, will make it relatively easy to do this. The best way to perform an ABT test project is to follow an agile approach, with flexibility in timelines, relatively small test products, and a high degree of interaction with stakeholders.

Keep your eye on the ball

Once a test module is defined, the scope defines many of its details, including the test objectives that detail out the purpose of the test module, which actions to use, what arguments to specify and for which ones to rely on default values, and what checks to do.

A well-designed test module will typically use some high-level actions to bring the application under test to a certain state and then run a number of test cases, often using more detailed actions. Finally, it uses one or more high-level actions to close down the application under test or bring it to a neutral state like a main window or page.

One of the worst practices in my view is to do checks on the fly. Various testing methods and tools actually encourage this by requiring specification of an expected outcome for each step in a test. So, you might be required to verify that after clicking the button "new car," the new car dialog is visible, even though such verification is not directly relevant for the purpose of the particular test.

Do not "debug tests"

More often than not, I see testing teams struggle to get numerous and large test modules to run successfully, rather than finding bugs in the application under test and reporting them. This leads to requests for more extensive "debug tools" for ABT. However, this is putting the cart before the horse.

Run the tests when appropriate. In particular, do not attempt higher-level test modules until more elementary tests, like smoke tests and UI-oriented tests, have been executed without failures. It does not make sense to enter a number of different transactions if the transaction dialog is not working properly.

Create bulletproof automation

As we see in ABT and other keyword methods, automation concentrates on the actions, not the tests, meaning automation issues—for example, with new versions of the application under test—can usually be resolved quickly by editing only a few actions. However, it still pays to invest efforts in smart automation, since this can make long test runs much smoother.

There are many things that can be said about automation, but there are some main tips to keep in mind. Have "active timing"—if you need to wait for the application under test to be ready for a next operation, look for a criterion that your script can actively watch for, like a "ready" text in a status bar. Too often, I see "sleep" statements with hard-coded values, which I like to call "passive timing." This will slow down the test run if the application under test is faster and, worse, will break the test if the system happens to be a bit slower than anticipated.

In many cases, the developers can help you uniquely identify UI elements in a simple way that is also likely to be stable across future builds of the application under test. Many elements happen to have one or more "hidden" properties that are not visible to a human user but that a test tool can "see." Examples are the "name" property in Java/Swing controls, the "accessibility name" property in .NET controls, and the "id" property in HTML elements. This approach also avoids complications when an application under test has releases in multiple languages: A button can be found whether its caption is "yes," "oui," or "si." It may take some convincing to let developers assign these properties, but it is worth the effort. The automation will be significantly more stable and reliable.

Be sure to test your automation. The automation engineer should take responsibility for the actions and working interface definitions, and he should have his own set of test modules to thoroughly verify this. This way, testers can rely on the actions to work and focus their efforts on the tests and the test results.

Organize it well

Building the right team to do the test task is more than half the work. I wrote a separate article on this in the September 2006 issue of *Better Software* magazine. The main differentiator is between test developers and automation engineers. Test developers focus on analyzing the functionality under test and creating good tests. Automation engineers focus on implementing action, dealing with "how" matters, like access to controls, timing of operations, etc. For both the test developer group and the automation engineers group, it is good to have experienced people who take responsibility for design decisions and give direction to the others.

A particularly good practice is to assign each stakeholder a point of contact with whom the test team has to deal. For the testers, these are typically subject matter experts or business analysts. For the automation engineers, these might be developers or IT managers. Designating points of contact maintains continuity in the exchange of information, assessment of tests and results, test execution environments, etc.—and it relieves the test manager.

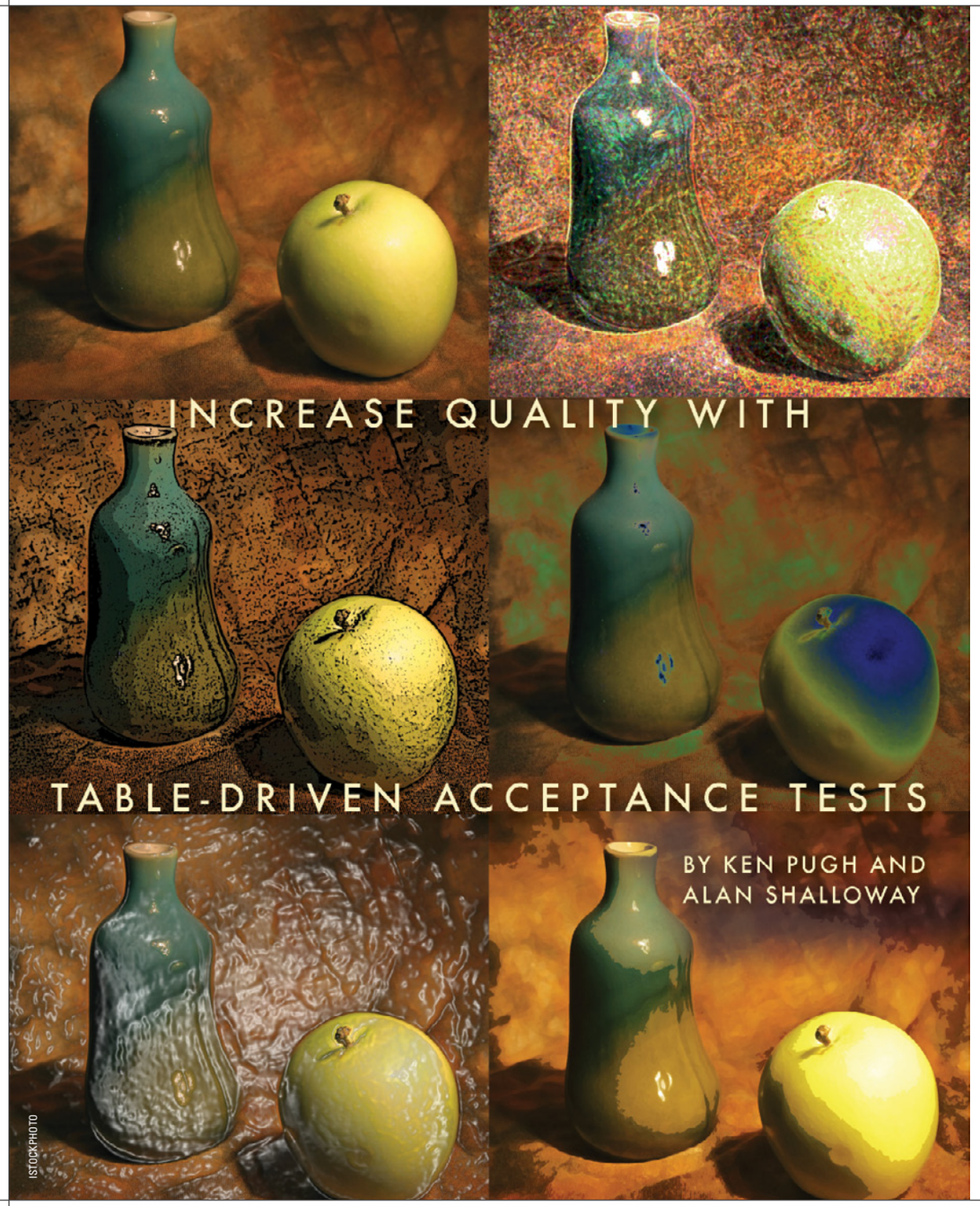
Conclusion

Action based testing is a keyword-based method for test management, test development, and test automation. It describes tests with action words and creates effective and maintainable testing at a large scale. The method can help but, over the years, I have learned that the real success is made by the intelligence and skills of the people applying ABT and the commonsense practices they follow. **{end}**

Sticky Notes

For more on the following topic go to www.StickyMinds.com/bettersoftware.

- References



Many organizations create vague or ambiguous requirements. When this occurs, developers can interpret the requirements one way and testers another way. Sometimes, neither way matches the customer's original intent. This misinterpretation causes loops in the development processes; however, creating requirements that include acceptance tests cuts down on the looping and increases the delivery of working software to the customer.

Typical Development Flow

The flow in a typical waterfall development process is shown in figure 1. Customers only meet with developers at the beginning of the project to go over requirement details. Testers do not receive the system until the end of the development. Each of the dotted lines represents a feedback loop. Defect reports are feedback for the developers. Depending on the type of defect, a change may be required in the code, design, or requirements. These issues cause delays in producing acceptable working software.

Figure 2 shows an alternative development flow. In starting on a requirement, a triad—the customer, developer, and tester—collaborate. During this collaboration, requirements and their corresponding, specific acceptance tests are developed. The tests are recorded in the customer's domain language and are understandable to the customer. The customer (or a subject matter expert) provides some initial information for the tests. To further clarify a requirement, the triad may create additional tests that cover such things as boundary values or behavior in exceptional situations. The developer uses these tests while creating the code. The implementation of the requirement is not complete until the tests have been successfully executed.

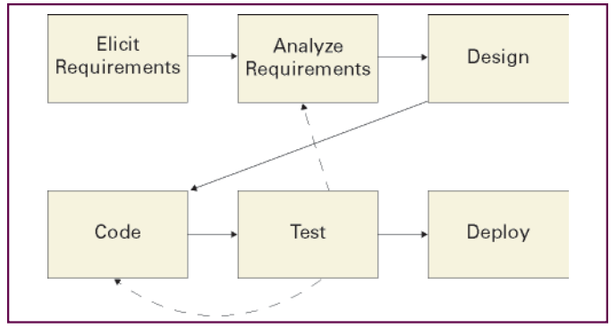


Figure 1

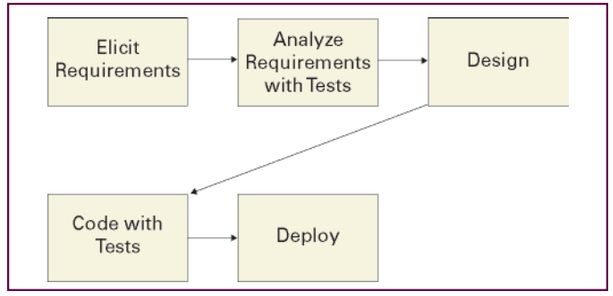


Figure 2

This process is called acceptance test-driven development (ATDD). Acceptance tests are required for all requirements. No coding begins until the tests are defined. The developer is not done with the implementation until it passes all the tests [1].

Requirements and tests are linked together. You can't have one without the other. They are like Bonnie and Clyde, The Captain and Tennille, Bullwinkle and Rocky, Starsky and Hutch,

and SpongeBob and Patrick. The tests clarify and amplify the requirements. A test that fails shows that the system does not properly implement a requirement. A test that passes is a specification of how the system works. Any test created after the code is written is a new requirement.

ATDD does not require that tests be automated. However, it does require that the results of the tests be visible to the customer. A developer entering test values with a debugger usually would not be considered visible to the customer. An xUnit test that is understandable to the customer may serve as a test. A document-based testing framework, such as the Framework for Integrated Tests (Fit), can be much easier for a customer to read.

Test Formation

A requirements document that uses prose or free-form text can be vague or ambiguous. For clarification, developers and testers can ask the customer to give us an example. This simple request is a powerful elicitation technique. Most people can create examples more easily than they can create higher-level abstractions, and these examples can be very specific and remove ambiguity. Then, those examples can be used as the tests. In addition, a tabular structure can record the examples concisely and can even be used in some automated testing frameworks, such as Fit.

Here's a sample of a prose specification:

"ATDD does not require that tests be automated. However, it does require that the results of the tests be visible to the customer."

If the ATDD tests are automated, they can serve as regression tests to ensure that changes have not affected the implementation of previous requirements. The tests can be automated using Fit, Selenium, Cucumber, or other testing frameworks (see the StickyNotes for more tools).

Frequently, there is a question about whether a test is an acceptance test (ATDD) or a unit test (test-driven development [TDD]). The difference is not in the form of the test—Fit or xUnit—but in the intent of the test. If the test is understood by the business, it is an acceptance test. If not, it is a TDD test. Many TDD tests are derived from ATDD tests. They specify the behavior of the modules in the system that support the behavior specified by the ATDD tests.

For each week, hourly employees are paid a standard wage per hour for the first forty hours worked, one and a half times as much for each hour after the first forty hours, and twice as much for each hour worked on Sundays and holidays [2].

The developers and testers are not sure they completely understand, so they ask for examples and receive those shown in table 1. The customer designates the names of the input and output values shown in the column headers. That keeps the examples in business domain terms. The customer should provide the basic cases. For this business rule, the customer may provide the first two lines. The first line describes the standard wage calculation, and the second line shows an overtime calculation.

Regular Hours	Sunday/Holiday Hours	Hourly Wage	Pay
40	0	$20	$800
41	0	$20	$830
40	1	$20	$860

Table 1

The testers and developers often ask for more complex examples. They compute the pay value for the third case ($840) and interpret the requirement to mean that an hour of Sunday work should be paid at twice the regular hourly rate. The customer says that the calculation is incorrect. He intended that the worker be paid at the rate for both overtime (1.5) and Sunday (2). So the multiplier should be 1.5*2=3, and the customer corrects the example. Without this example in advance, the tester or developer might simply use 2 as the multiplier, causing the requirement to be erroneously implemented.

The third case may prompt the question: What if the day were both a Sunday and a holiday? Should it be paid at four times the rate? How would we know? Ask the customer.

Of course, we cannot be certain that we've discovered all of the possible questions and found all the corresponding examples. But, the examples can document which questions have been asked and answered. If a value is not known at the time of asking, you can put "TBD" or "IGBTYOT" ("I'll Get Back To You On That") in that entry to document an "asked, but not yet answered" question.

The examples in table 1 are ones that the customer could readily provide. Often, the tester can create many more, in-

Regular Hours	Sunday/ Holiday Hours	Hourly Wage	Pay	Reason (Requirement)
39	1	$20	$820	39@$20=$780 1 Sun@$20×2=$40 Total=$820
0	1	$20	$40	0@$20=$0 1@$20×2=$40 Total=$40
40	1	$20	$860	40@$20=$800 1@$20×2×1.5=$60 Total=$860
-1	0	$20	Error	Cannot work fewer than zero hours
169	0	$20	Error	Cannot work more than entire week
0	40	$20	$1600	0@$20=$0 40@$20×2=$1600 Total=$1600
0	41	$20	$1600	0@$20=$0 40@$20×2=$1600 1@$20×2×1.5=$60 Total=$1660

Table 2

cluding boundary values. All examples created by developers or testers should be checked and approved by the customer. Often, additional requirements may emerge from those additional examples.

Additional examples created by the tester are shown in table 2. Note that these tests assume that the unit of recording hours is an hour, rather than .1 hour. If the latter were the case, then you might choose 39.9 and .1 as test cases as well. The errors in the fourth and fifth examples are caused by the hours' exceeding the boundaries of reality.

The example may provoke more questions. Is this wage calculation for a single state? Do holidays vary per state? Are there floating holidays? How should working on a floating holiday be handled? These questions, and possibly many more, should be answered before implementation. The answers may suggest that the requirement be broken down into subrequirements that are separately considered for development. Automated acceptance tests can ensure that as you develop these additional subrequirements, you do not break previously developed ones. Perhaps the business value of handling a floating holiday is so low that it should be ignored or handled as a special case (e.g., a worker schedules a floating holiday on the last possible day in a year and then is required to work that day).

Conclusion

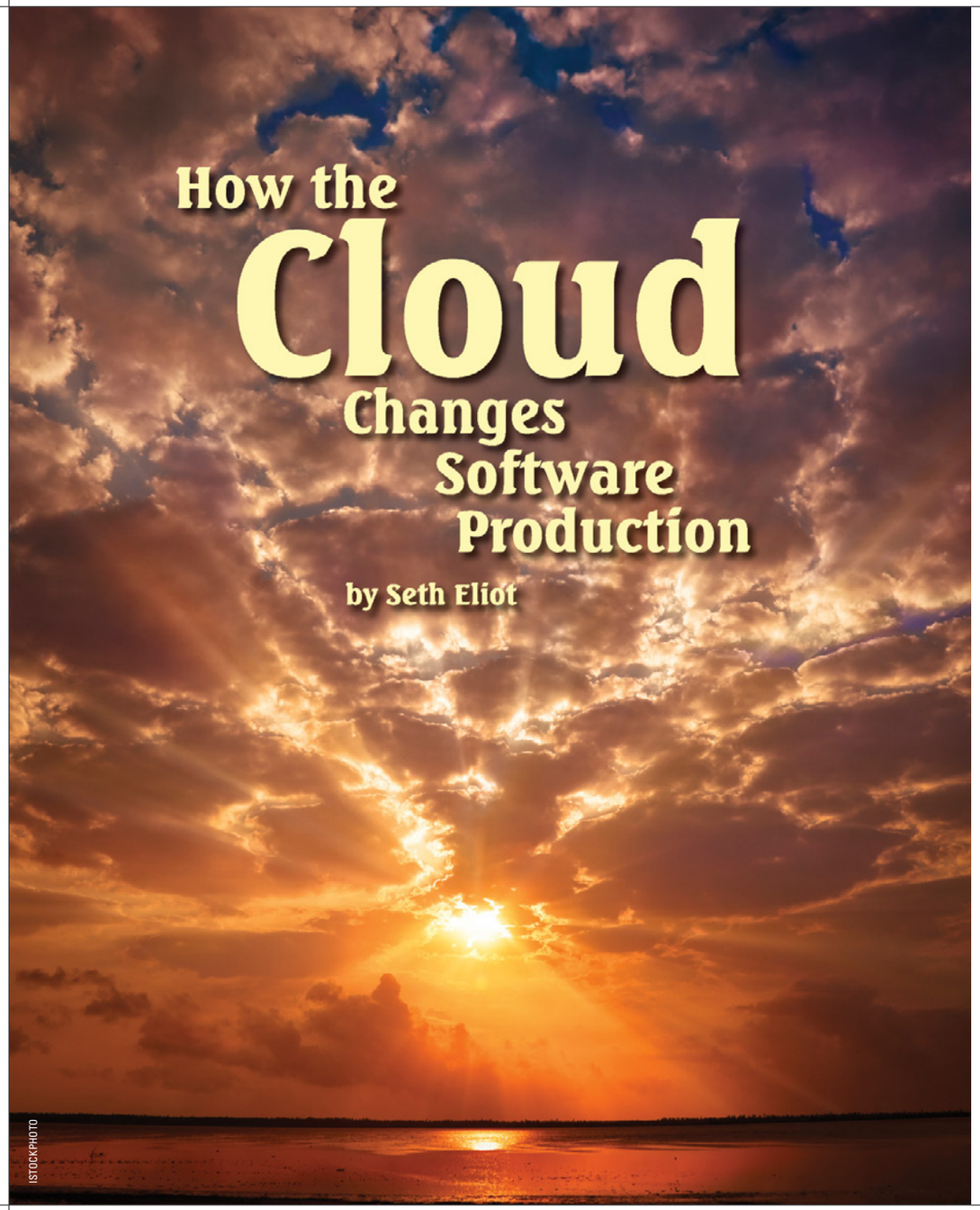
Defining acceptance tests prior to implementation can create a better understanding of the requirements. The specific examples used in the tests can drive out requirements details or even additional requirements. Putting the examples in a tabular structure captures them concisely. The tables can be used for automated testing. By using the tests in conjunction with coding, developers can get immediate feedback as to how their implementation matches the requirements. Early testing decreases defects and helps promote faster flow through the development process. **{end}**

ken.pugh@netobjectives.com
alshall@netobjectives.com

Sticky Notes

For more on the following topic go to www.StickyMinds.com/bettersoftware.

- Automated test tools
- References



For the software engineering and quality professional, cloud computing promises many new opportunities. Engineers no longer need to procure and provision hardware nor manage the data centers or service platforms their software runs on. Scalability is handled for the engineer as the cloud expands to absorb increased service load. And quality practices such as rapid prototyping and ramped service rollout are enabled.

Introducing Cloud Computing

A brief definition of cloud computing provides a foundation from which to explore the challenges and solutions introduced by this technology. Cloud computing is about moving resources, data, and software off the desktop or local server and onto a pool of shared servers. This pool—or the totality of all such pools—is the cloud. We can look at cloud computing as three service-layer categories.

Infrastructure as a Service (IaaS)

In this category, the cloud replaces a developer's need for a physical resource such as storage or entire servers. Instead of maintaining a physical data center building filled with server hardware and the systems to support it, a software developer can instead use the resources of a cloud infrastructure provider. Hard drives or storage area networks are replaced with simple PUT and GET APIs into the cloud storage service. Racks of physical servers are replaced by virtual machines created on demand by the cloud server provider.

Platform as a Service (PaaS)

This category provides developers with resources and functionality used by applications they develop. This may be a relational database (RDBMS) to store terabytes of data or a web server to serve an ecommerce website. PaaS provides functionality that previously required enterprise level software to be purchased and professionally maintained by IT staff, but now developers instead rely on the cloud platform provider.

Software as a Service (SaaS)

These are cloud applications. This category enables consumer-level end-users to use the power of the cloud to run programs traditionally run from the desktop or enterprise workstation. Instead of users editing a document stored on their hard drives using Microsoft Word installed on their PCs, they can store their document in the cloud and edit it using Microsoft Office Web Apps or Google Docs. These products provide word processing capabilities through an Internet browser powered by code running in a Microsoft or Google data center.

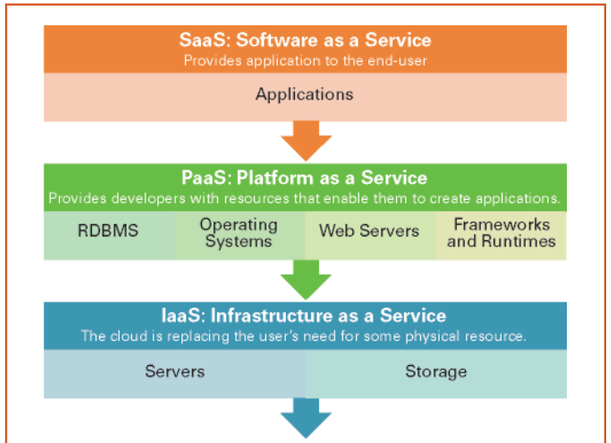


Figure 1: Cloud resources by category form a stack

Real-world examples of cloud services from each of these three categories can be found in the Sticky Notes.

Figure 1 illustrates that the three cloud computing categories form a stack. This means that for each layer, the cloud not only handles its resources but also handles the relevant resources from the layers under it. For example, for PaaS to provide relational database capability in the cloud, the servers and storage (IaaS) used by that database must also be provided in the cloud.

Some gray area exists as to how cloud services are categorized. For example, virtual servers in the cloud are considered infrastructure (IaaS). However, most providers will include an operating system on these servers, which is platform (PaaS). Another example is elastic web hosting like SquareSpace. It is a platform (PaaS) for hosting and serving your website, but it also provides tools (SaaS) for building and maintaining that site. These gray areas challenge the simple three-category model of the cloud and can confuse newcomers to cloud technologies.

Software Production—Leveraging the Cloud

Two elements that contribute to our leveraging the cloud for software production are *virtualization* and *elasticity*.

Virtualization starts at the bottom of the cloud stack in figure 1. The cloud user accesses servers, storage, and other physical resources as logical entities that may have no direct relation to the underlying physical hardware. For example, a user of Amazon EC2, an IaaS implementation, gets access to what appears to be several Linux Servers, each with its own CPUs, disks, and networking. In reality, each virtual server is likely sharing CPU, disk space, and networking on one or more physical servers. The end-user has no visibility into the underlying physical resources; he only cares that he can treat his virtual server like the real thing. The benefit of this is that the end-users get the functionality they care about without needing to worry about how it is implemented or how to maintain it.

Elasticity refers to the ability to increase or decrease virtual resources easily. Adding servers to a physical data center involves purchasing, delivering, installing, and configuring them, but adding virtual servers can happen instantly as the virtualization software reallocates physical resources and spawns new virtual servers.

Development and the Cloud

Using elastic cloud-based infrastructure or platforms frees the services developer from having to worry about scalability and load issues. For a traditional system, the developer needs to worry how increases in user load and other demands on resources will be handled by his system. Freedom comes when such demands are handled elastically by the cloud platform—as long as the developer has chosen the right platform and has designed within the specs for that platform. Do not make the mistake of using this to justify writing sloppy software. Create systems that hog CPU, memory, or disk storage and you will bump up against the configured limits of the logical platforms or infrastructure of your cloud, resulting in service failures and possible user impact. The cloud rewards sound design by freeing developers from the concerns of how their services will handle wild success and the increased usage that success brings, but the cloud does not compensate for poor design.

Selecting the right cloud platform increases software reliability, minimizing developer concerns about service redundancy and data replication, which are handled by the cloud service provider. For example, Microsoft Windows Azure boasts 99.9 percent uptime for services and 99.9 percent availability for storage [1], while Amazon EC2 offers 99.95 percent uptime [2]. Therefore, the right cloud platform is the one that offers the service guarantees that meet your business needs. Good design will further increase reliability. In fact, Microsoft raises its service guarantee to 99.95 percent if developers follow recommended implementation practices [3].

Media delivery is a growing application for cloud services. A recent study found Netflix's movie streaming accounts for 20 percent of all downstream web traffic in the US [4]. Developers of these services want reliable delivery of their media with low latency. Cloud platforms provide the content delivery network (CDN) that provides geographically distributed servers (also known as edge servers) where media is cached and ready for quick delivery to end-users. While CDNs have existed for a while, the cloud-enabled benefit of CDNs like Amazon CloudFront and Windows Azure is that they serve content directly from their respective cloud storage systems on the backend (Amazon S3 and Microsoft Windows Azure Storage respectively). The advantage of this is improved integration and reliability of these cloud services. Storage, databases, web servers, and CDNs are all integrated along with the execution stack as part of the cloud platform. Availability and interoperability of all these systems, especially within a single cloud provider, is much simpler than building a full solution.

However, once again, the cloud is not a panacea. Developers are still responsible for good design and resource utilization. The cloud also will impose new responsibilities on the developer; chief among these is security. When using cloud data storage, for example, the data is out of your hands, and though cloud providers implement user authentication to protect access to your data, your business needs may call for you to encrypt sensitive data before uploading it to the cloud as an extra security measure. In some cases, the cloud may not be appropriate for your data.

Testing and the Cloud

Cloud services open the door to new ways of testing and evaluating. Ease of deployment lends itself well to rapid prototyping, where new software and ideas get tested with real world usage. Ramped deployments—where at first a small but then increasing number of users are exposed to a new version

of your service—are enabled by the cloud. For example, elasticity enables you to spin up new servers to host the new version for users in the ramp up, while other users still see the old version. After all users are moved to the new version and it is vetted, the servers running the old version are released back into the cloud.

Cloud tools can be employed to drive efficiencies in the software test process. In some scenarios, this is as simple as replacing the test lab with cloud-hosted servers and services. The advantages are cost of entry, scalability (via elasticity), and maintenance [5]. Another application of cloud tools is load generation, which can leverage elasticity to ramp simulated user demand to near real-world levels. A test organization could use cloud-based servers (IaaS) and distribute custom load-generation code for its service under test, or it could employ a framework already designed for this, like SOASTA's CloudTest product [6]. SOASTA has published case studies simulating 1 million concurrent users generating 16 GB/second of bandwidth and 77,000 hits/second to test a new MySpace feature [7]. Cloud-based test rigs, similar to CDN functionality discussed earlier, also can enable geographically diverse test load origination.

The Team and the Cloud

The cloud doesn't change the need for quality-minded engineers to create product and drive the process, although the process itself may be affected when creating cloud services. The availability of cloud resources and the techniques they enable lends itself to an iterative agile process. Agile staples, such as continuous integration, are complemented by cloud technologies such as virtual labs, where a shared infrastructure provides on-demand lab resources throughout the organization [8]. Rapid prototyping and flighting enabled by the cloud are also techniques that are most effective in agile, iterative environments. Project managers and software teams will want to consider this natural affinity for agile when deciding on processes to produce their cloud services.

Keep the Change

The technologies and applications described here illustrate the power that cloud-based services and tools can bring to bear on the production of quality software. Developers benefit from ease of deployment and scalability of their services, as well as on-demand access to resources for hosting their software and data. Testers gain new tools for applying large-scale loads and new methods for assessing functionality with real-world usage. And, teams can employ iterative approaches that better align produced software with the users' needs. With such benefits, it appears that the cloud is here to stay, and we in the software engineering field will produce better software because of it. **{end}**

seth.eliot@microsoft.com

Sticky Notes

For more on the following topic go to www.StickyMinds.com/bettersoftware.

- References
- Cloud services examples

Codenomicon Announces SNMP Trap Suite

OULU, FINLAND and CUPERTINO, CA—Codenomicon has announced the launch of its SNMP Trap security test suite. This latest addition to the SNMP test tool suite marks a new threshold for the company, as its model-based testing solutions now cover more than 200 protocol simulations.

Codenomicon has also announced a new Defensics 3.12 release, which includes a number of new features including controls for creating more test scenarios and for adjusting test coverage and speed. Other exciting new features include the reproduction crash-level issues and the improved test performance and portability configurations, which enable Defensics to be used in modern virtualization environments and cluster-based setups with hundreds of parallel test generators and simultaneous test targets.

Visit www.codenomicon.com/defensics for more information.

Microsoft Dynamics AX "6"

REDMOND, WA—Microsoft Corp. has revealed the next version of Microsoft Dynamics AX, code named Microsoft Dynamics AX "6".

New innovations in Microsoft Dynamics AX "6" include:

- A unique, model-driven, layered architecture that accelerates software development, requiring less coding than building from scratch and easing maintenance and upgradability. This allows developers to build high-value functionality quicker and better.
- A unified ERP solution with prebuilt capabilities for five industries and thirty-eight countries, providing a rich foundation that allows ISVs to rapidly expand their solutions to new verticals and geographies.

Visit www.microsoft.com for more information.

FAQ

expert answers to frequently asked questions

What are good sources of less common agile test ideas?

I see many agile planning sessions that generate plenty of interesting programming tasks but only the blandest of testing ideas. I encourage testers to take an active part in agile planning. Be on the lookout for important test opportunities; don't just go with the flow. You can do better than just adding a bunch of tasks called "Test Story X." Try out some new sources of test ideas in your next agile planning meeting.

Normal Flow: Exercise the typical path a user takes to successfully fulfill the story. An example would be buying a book online given a valid and available title. You should consult users about their common usage patterns.

Alternate Flow: Exercise alternates to the typical path a user takes. An example would be buying a book with an ISBN number or author name as opposed to the title.

Error Flow: Exercise paths through the story in which the user initiates an error state or invalid condition. An example would be trying to buy a book that does not exist.

Work Flow: Exercise the story as part of the way a user would get his "normal" job done. Include steps a user would take before and after the story being tested. Try to include manual and automated steps, too.

Data Flow: Exercise the different ways data flows through the application. Look at data created, updated, deleted, or just accessed by the story under test. Consider data with different origins and with a variety of record types.

Control Flow: Exercise different paths through the code. Look at the source code and identify the fundamental pathways through the code. Set up test conditions to navigate selected pathways.

Other types of testing ideas you might consider relate to exploring characteristics of the application that may be impacted by the new story. For example, you may consider performance and load testing opportunities.

Don't forget to consider test ideas related to how a system recovers from common or typical failures, such as running out of disk space during a transaction. I consult with help desks and system administrators to learn about failures that really happen.

By actively participating in agile planning, you can help the team focus on what matters by considering and scrutinizing a wide variety of great testing alternatives.

by Robert Sabourin
rsabourin@amibug.com

Learning for Agile Testers, Part 1

Your skills are what make you a valuable asset to a team. In part one of this two-part series, we recommend skills every QA professional needs.

by Lisa Crispin and Janet Gregory | lisa.crispin@gmail.com janet@agiletester.ca

What makes a "good tester"? Many testers are unsure what skills they should learn to obtain their desired positions. Some testers just give up and resign themselves to sticking with what they know.

In this series, we answer questions like:

- What do testers need to know in order to add value in software development projects?
- What skills will help them obtain the most rewarding jobs?
- Where can they go to learn what they need?

In part one, we recommend skills that any well-rounded QA professional needs. In part two, we'll delve into more specific technical skills that help testers add value and suggest some ideas on how testers can grow professionally.

In our experience, the best testers can quickly pick up new specific skills, such as creating test plans or writing up bug reports because they have a good foundation of what Isabel Evans calls "thinking" skills [1]. These skills are not tangible in the sense we can say, "I've learned that, I can practice it perfectly now." Some of them are often called interpersonal or "soft" skills. The ability to collaborate is a requirement for agile testers. On traditional teams, often the only people testers talk to are other testers. On agile teams, a tester works closely with customers or the product owner to elicit requirements and uncover hidden assumptions. This is why concepts such as systems thinking—How did we get here? and What changes impact other parts of the system?—are so critical.

Testers collaborate with stakeholders to define acceptance tests and examples of desired system behavior and mis-behavior for each new feature or user story. A tester who knows when and how to get the right people talking can prevent misunderstood or missed requirements. Knowing how to facilitate a specification workshop [2] or a brainstorming meeting when there's no business analyst or other specialist available is a bonus. Testers must feel comfortable working closely with developers to ensure that testing and coding is a seamless process with adequate test coverage.

"Soft" doesn't mean "easy" when we talk about nontechnical skills. For example, how many of us really know how to use problem-solving skills? It is one of those transferable skills that can be applied to test design, debugging, coaching, or teaching.

Janet's story: When I first took my exam for the ASQ Certified Quality Manager, I failed the written part, which was how to address two spe-

cific problems. I believe it was because I didn't remember how to apply my problem-solving skills. I first learned those skills when I took Physics 101 at university, but I didn't apply them on a regular basis. When I failed my exam, I sat back, figured out the root cause, and went back to basics. I retook my exam and followed the principles of problem solving and passed. I then presented what worked for me for the next group of people wanting to take the exam, which reinforced what I had learned.

Providing feedback is a tester's most important function, and it generally needs to be done in a constructive manner. Knowing how to keep the focus on the work, not on the person, is essential. Bug reporting is one way to provide feedback, but there are more important ways. It takes time and practice to learn how to engage colleagues in a positive exchange when talking about negative issues. Learning empathy is essential—think about how would you want to receive unwelcome information? Skills such as the speech evaluations taught by Toastmasters International are transferable and can be essestial to providing feedback on software teams.

Coaching skills enable testers to mentor others. It's much more valuable to guide others in solving their own testing and quality issues than to give them the answers. Observing and listening are critical communication and collaboration skills; use them to know when complaints are legitimate or to encourage shy teammates to speak up. Knowing when and how to listen helps the team grow and improve.

No tester's toolkit is complete without brainstorming tools, such as mind mapping, or root cause analysis tools, like the 5 Whys and fishbone (Ichikawa) diagrams. These can be applied when eliciting requirements and examples from customers or helping the team identify impediments to quality and improve process.

Even if you don't work on an agile team, we recommend that testers understand how they can apply agile values, principles, and practices to their work. Close collaboration with customers and programmers throughout the application lifecycle helps "bake quality in." Breaking tasks into small incremental and iterative "chunks" is one way to get fast feedback into your test cycle. Lisa gives an example of applying agile principles using customer tests.

Lisa's Story: After working on two excellent Extreme Programming teams, I became the director of QA at a company that said it wanted to transition to agile. Though the programming team never embraced agile, my test team found ways to succeed with "agile testing." For example, I asked the development managers about their biggest pain point. They agreed it was requirements—either requirements took too long to get, leaving no time for coding, or there weren't any at all. I suggested we start writing customer tests in place of traditional requirements documents and let the programmers work from those. This worked well on all sides.

Organizational skills are also essential. Knowing how to plan and manage your time, using concepts such as risk-based testing, helps you focus on the right tasks. With so many demands on your time

in the form of meetings, email, instant messages, planning, and tracking activities, it can be hard to do actual testing or learn any other skills your project may require.

Challenge yourself to learn new skills and get out of your comfort zone so you can contribute more value to your team. Continue to improve your skills to enhance your career and have fun while you do it! Get going today by starting a book club with your team, finding a tutorial on the web, or joining a local or global testing community. We've provided some links in the Sticky Notes to help you get started.

In part two, we go into detail on specific skills that benefit testers and we provide more guidance on good ways to learn them. The fact that you're reading this article means you care about your professional development. Your learning journey will lead you and your team in new and exciting directions! **{end}**

Sticky Notes

For more on the following topics go to www.StickyMinds.com/bettersoftware.

- Further reading
- References

Visit **TechWell.com** to read "Why We Need to Learn More." A continuation of the ideas discussed in this article.

index to advertisers

Better Software (USPS: 019-578, ISSN: 1553-1929) is published six times per year January/February, March/April, May/June, July/August, September/October, November/December. Subscription rate is US $40.00 per year. A US $35 shipping charge is incurred for all non-US addresses. Payments to Software Quality Engineering must be made in US funds drawn from a US bank. For more information, contact info@bettersoftware.com or call 800.450.7854. Back issues may be purchased for $15 per issue (plus shipping). Volume discounts available.

Periodicals Postage paid in Orange Park, FL, and other mailing offices. POSTMASTER: Send address changes to Better Software, 340 Corporate Way, Suite 300, Orange Park, FL 32073, info@bettersoftware.com.