



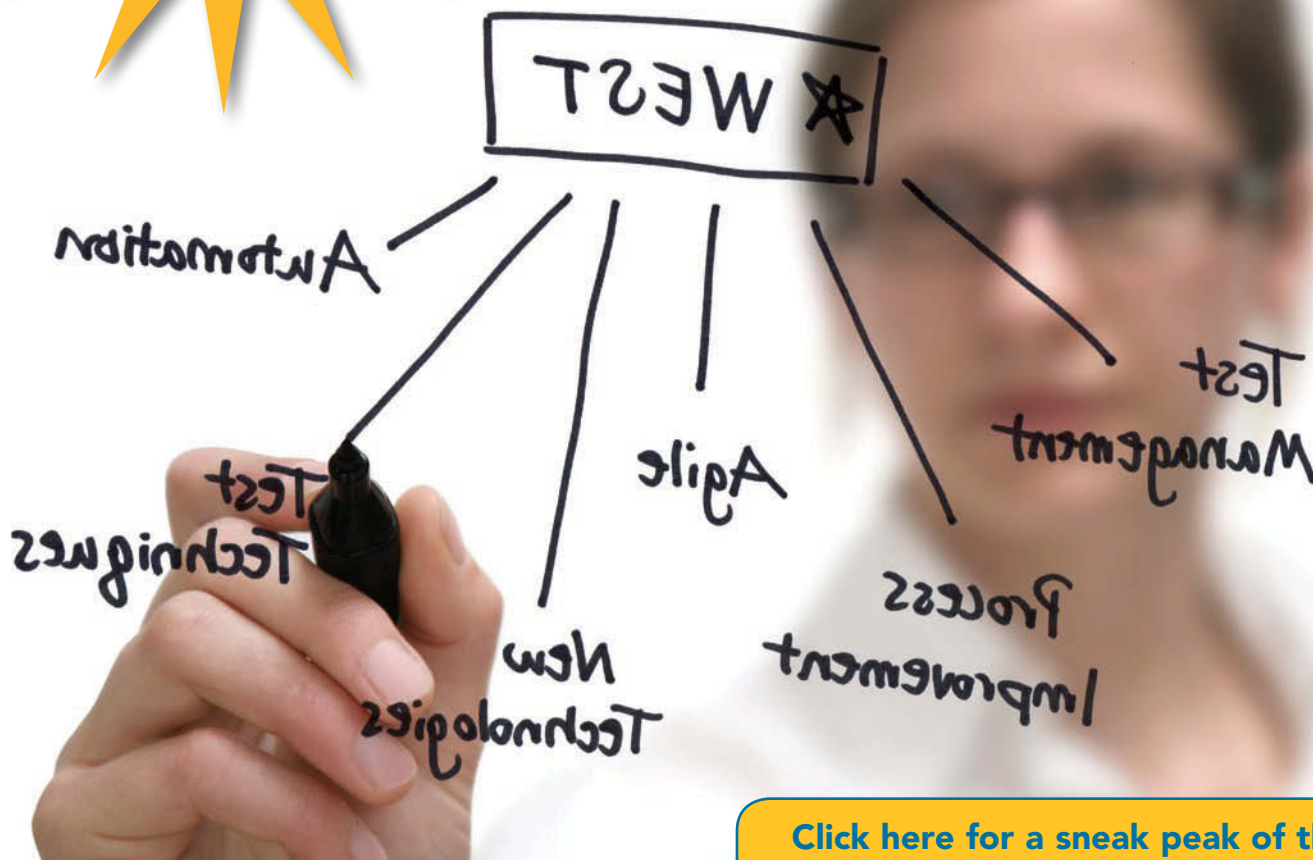
**STAR
WEST**

SOFTWARE TESTING

ANALYSIS & REVIEW

OCTOBER 5–9, 2009
ANAHEIM, CALIFORNIA
DISNEYLAND® HOTEL

The Greatest Software Testing Conference on Earth



[Click here for a sneak peak of the
STARWEST Keynote Speaker Videos](#)

**99% OF 2008 ATTENDEES RECOMMEND
STARWEST TO OTHERS IN THE INDUSTRY**



www.sqe.com/starwest

**MAXIMIZE YOUR DISCOUNTS
AND SAVE UP TO \$600.
GROUPS OF 2 SAVE EVEN MORE!**



**Large-scale Exploratory Testing: Let's Take
a Tour**
James Whittaker, Google



The Marine Corps Principles of Leadership
Rick Craig, Software Quality Engineering



**Moving to an Agile Testing Environment:
What Went Right, What Went Wrong**
Ray Arell, Intel



**The Top Testing Challenges—or
Opportunities—We Face Today**
Lloyd Roden, Grove Consultants



The Irrational Tester
James Lyndsay, Workroom Productions, Ltd.

July/August 2009

\$9.95 www.StickyMinds.com

BETTER SOFTWARE

The Print Companion to

StickyMinds.com

WHY AUTOMATE?
6 good (and not-so-good)
objectives

SOUND THE ALARM!
Burn charts for tracking
project progress

Developing
Applications
for a Wireless
World



**FIND THE BUG INSIDE & WIN
AN AMAZON GIFT CARD!**

Cut Dev Costs... Without Cutting Heads.



*It's tough to look
at your team and
decide who should stay
and who should go.*

Rally's customers are proven to be
50% faster to market and
25% more productive,
so you can avoid cutting heads.
Plus Rally is the only Application Lifecycle
Management provider to ensure
your Agile success with
a money-back guarantee.

**Learn more about our Guaranteed Success Program at
www.rallydev.com**



Scaling Software Agility

© 2009 Rally Software Development Corp

Save time while protecting software quality.



© 2009 Seapine Software, Inc. All rights reserved.

Swiftcover.com cut their testing time in half with TestTrack Studio and QA Wizard Pro, while still providing the quality their customers expect.

Seapine's end-to-end Software Quality Assurance (SQA) solutions help you deliver quality products faster. Start with **QA Wizard Pro** for automated testing and add **TestTrack Studio** for issue tracking and test case management—integrated quality assurance solutions that together reduce testing time, saving you money and improving customer satisfaction.

- Reduce quality assurance costs with automated functional and regression testing.
- Manage test case development, defect assignments, and QA verification with one application.
- Track which test cases have been automated, schedule script runs, and link test results with defects.

“So much of success boils down to time. QA Wizard Pro and TestTrack Studio allow us to be more profitable because we do more in less time. — Test Manager, Swiftcover”

Learn how to test faster while protecting quality. Visit **www.seapine.com/betterswift**

Test your reality with Cognizant. Compete with confidence.

Test data plays the central role in testing. IT organizations handle incredible volumes of complex test data, consuming significant effort and time. Increased outsourcing and distributed environments further add to the complexity in streamlining and managing data for testing.

Cognizant's test data framework ensures consistency and operational efficiency in handling data for testing. Our framework enables accelerated test data generation, localization, scrubbing, masking de-personalization, and cleansing, thereby optimizing complex test environments.

With Cognizant, you get certified continuous business readiness, lower costs, and accelerated results unmatched in the industry.



Cognizant | Testing Services
Passion for building stronger businesses

World Headquarters

Cognizant Technology Solutions
500 Frank W. Burr Boulevard
Teaneck, NJ 07666
Ph: +1 201 801 0233
Fax: +1 201 801 0243
Toll free: +1 888 937 3277
Email: inquiry@cognizant.com



Cover Story

SOFTWARE TO GO 20

The mobile arena is in constant evolution, changing the way we approach software development both from a business and a technical perspective. Taking the time to set your plan can make the difference between success and just a good idea.

by Luis Miguel Carvalho

Features



FEEL THE BURN 26

Burn down charts are a popular project artifact, but too often, people accept the default chart from whatever project management tool they're using. How can we create charts that answer the questions that are really important?

by George Dinwiddie

THAT'S NO REASON TO AUTOMATE! 32

Automating test execution is supposed to give tremendous benefits, but often gives disappointing results. The fault may not lie with the automation itself, but with the objectives you are attempting to achieve.

by Dorothy Graham and Mark Fewster

Columns & Departments

In Every Issue

Mark Your Calendar 4

Contributors 6

Editor's Note 7

10 Things You Might
Not Know About ... 42

Ad Index 44

Better Software magazine—The print companion to StickyMinds.com brings you the hands-on, knowledge-building information you need to run smarter projects and deliver better products that win in the marketplace and positively affect the bottom line.

Subscribe today to get six issues.

Visit www.BetterSoftware.com
or call 800.450.7854.

TECHNICALLY SPEAKING 9

Predicting the Past • by Lee Copeland

Developing an accurate prediction process is complex, time consuming, and difficult. But, basing predictions on causality rather than correlation and learning how to “predict the past” can help us gain confidence in the validity of our work.

CODE CRAFT 10

Testing the Contract Metaphor • by Kevlin Henney

A contract represents a service agreement between two parties, the bounded provision of service by one party to the other. This metaphor also applies to how we can think about the relationship between unit tests and code.

INSIDE ANALYSIS 14

How Agile Practices Reduce Requirements Risk • by Ellen Gottesdiener

Requirements risks are among the most insidious risks threatening software projects. As requirements expert and agile coach Ellen Gottesdiener explains, agile practices can go a long way in mitigating those risks.

TEST CONNECTION 16

Three Kinds of Measurement and Two Ways to Use Them • by Michael Bolton

Are software development and testing sciences subject to the same kind of numerical measurement that we use in physics? If not, what kinds of measurements should we use? How could we think more usefully about measurement?

MANAGEMENT CHRONICLES 18

Avoiding Half-baked Discovery • by Didier Thizy

Software projects have fixed costs that often get overlooked in project planning—setting up development environments, ramp-up, and building frameworks, to name a few. Find out how you can position this with your customer during discovery.

THE LAST WORD 43

Adapting Inspections to the Twenty-first Century • by Ed Weller

Teams working in multiple time zones have limited opportunities for the team meeting, an integral part of the inspection process. This list of requirements and the functions needed to solve this problem should help anyone faced with this problem.

MARK YOUR CALENDAR

TRAINING WEEKS

www.sqetraining.com/public

Testing

September 14–18, 2009

Washington, DC

October 19–23, 2009

San Francisco, CA

November 16–20, 2009

Tampa, FL

Agile Software Development

September 21–25, 2009

Boston, MA

October 12–16, 2009

San Diego, CA

SOFTWARE TESTER

CERTIFICATION

www.sqetraining.com/certification

August 25–27, 2009

Richmond, VA

San Jose, CA

September 1–3, 2009

Albany, NY

Charlotte, NC

September 15–17, 2009

Boston, MA

Minneapolis, MN

Salt Lake City, UT

September 22–24, 2009

Atlanta, GA

Philadelphia, PA



CONFERENCES

STARWEST 2009

Software Testing

Analysis & Review

www.sqe.com/starwest

October 5–9, 2009

Disneyland Hotel

Anaheim, CA

Agile Development

Practices

www.sqe.com/adp

November 9–13, 2009

Rosen Shingle Creek

Orlando, FL

STAREAST 2010

Software Testing

Analysis & Review

www.sqe.com/stareast

April 26–30, 2010

Rosen Shingle Creek

Orlando, FL

Better Software

Conference & EXPO

www.sqe.com/bsce

June 7–11, 2010

Caesars Palace

Las Vegas, NV

BETTER SOFTWARE

Publisher

Wayne Middleton

Vice President of Publishing

Holly N. Bourquin

Editor in Chief

Heather Shanholtzer

Editorial

Managing Technical Editor

Lee Copeland

Editor, StickyMinds.com

Francesca Matteu

Managing Editor, Multimedia

Joseph McAllister

Production Coordinator

Cheryl M. Burke

Design

Creative Director

Catherine J. Clinger

Advertising

Senior Advertising Sales Manager

Shae Young

Production Coordinator

April Evans

Circulation and Marketing

Circulation Coordinator

Jamie Green-Gago

Marketing Coordinator

Stephanie Fender

A PUBLICATION OF
SOFTWARE QUALITY ENGINEERING



CONTACT US

Editors: editors@bettersoftware.com

Subscriber Services: info@better-software.com

Phone: 904.278.0524, 888.268.8770

Fax: 904.278.4380

Address:

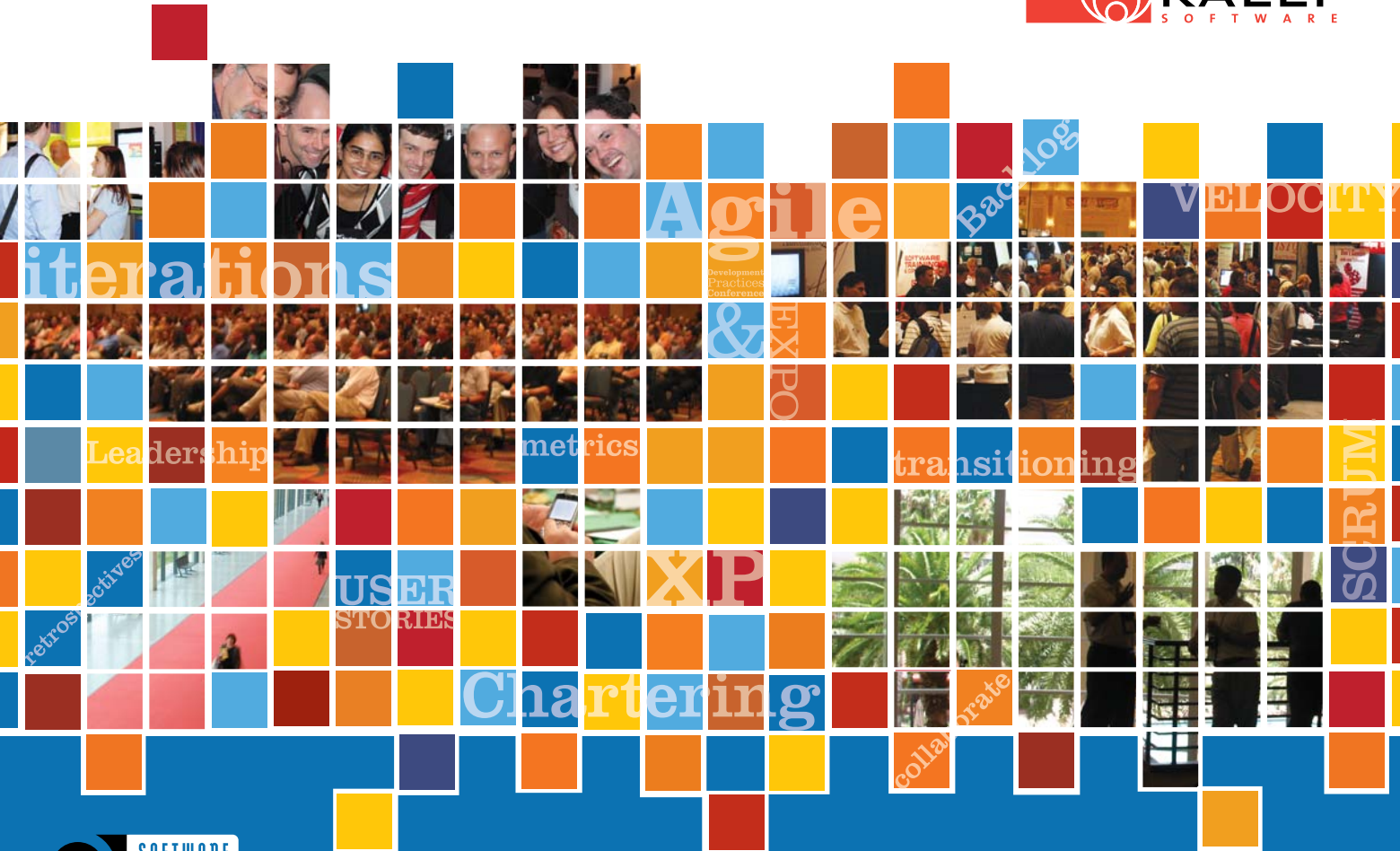
Better Software magazine
Software Quality Engineering, Inc.
330 Corporate Way, Suite 300
Orange Park, FL 32073

Agile Development Practices Conference

November 9-13, 2009
Orlando • Florida
Rosen Shingle Creek



Conference Sponsor



Register Early & Save Up to \$600!

www.sqe.com/adp

KEYNOTES

BY INTERNATIONAL EXPERTS



Lyssa Adkins

CRICKETWING

NAVIGATING CONFLICT ON AGILE TEAMS:
WHY "RESOLVING" CONFLICT WON'T WORK



Jim Highsmith

CUTTER CONSORTIUM

BEYOND SCOPE, SCHEDULE, AND COST:
RETHINKING PERFORMANCE MEASURES FOR AGILE DEVELOPMENT



Alistair Cockburn

HUMANS AND TECHNOLOGY, INC.

AGILE: RESETTling AND RESTARTING



Joshua Kerievsky

INDUSTRIAL LOGIC, INC.

AGILE BRUSHSTROKES:
THE ART OF CHOOSING AN AGILE TRANSITION STYLE



Contributors



SCOTT BAIN is a thirty-plus-year veteran in computer technology with a background in development, engineering, and design. Scott teaches courses and consults on agile analysis and design patterns, advanced software design, and sustainable test-driven development. A frequent speaker at developer conferences such as JavaOne and SDWest, Scott is the author of *Emergent Design: The Evolutionary Nature of Professional Software Development*.



MICHAEL BOLTON lives in Toronto and teaches heuristics and exploratory testing in Canada, the United States, and other countries. He is co-author, with James Bach, of *Rapid Software Testing* and a regular contributor to *Better Software* magazine. Contact Michael at mb@developsense.com.



LUIS CARVALHO is an experienced software engineer in the mobile computing industry where he helps create and deliver innovative mobile solutions. Prior to joining Microsoft, Luis worked at MobiComp as a software architect responsible for improving existing solutions and driving innovation. After Microsoft's acquisition of MobiComp in 2008, he joined the Microsoft My Phone QA team as senior development engineer.



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. Lee 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.



GEORGE DINWIDDIE helps teams address their current impediments through improved engineering practices, enhanced design and testing skills, reduced waste effort, clarified goals, and better communication and teamwork. Kicked off by an early start in television repair, George's career has included electronic hardware development, embedded firmware, and information technology. He has shared his expertise at such venues as the Agile Conference, XP Day North America, the Simple Design and Testing Conference, and regional agile organizations.



MARK FEWSTER has twenty-plus years of experience in software testing. With Grove Consultants, Mark provides consultancy and training in software testing. He has published papers in respected journals, co-authored *Software Test Automation* with Dorothy Graham, and is a popular speaker at national and international conferences and seminars. Mark has served on the committee of British Computer Society's Specialist Group in Software Testing and has contributed to both the ISEB and ISTQB software testing certification schemes.



ELLEN GOTTESDIENER, principal consultant and founder of EBG Consulting, is an internationally recognized trainer, facilitator, speaker, and expert on collaborative requirements development. An agile coach and trainer with a passion for agile requirements, Ellen works with large, complex products and helps teams elicit just enough requirements to achieve iteration and product goals. Author of *Requirements by Collaboration: Workshops for Defining Needs* and *The Software Requirements Memory Jogger*, Ellen speaks at and advises for industry conferences, writes articles, and serves on the expert review board of the International Institute of Business Analysis Body of Knowledge.



DOROTHY GRAHAM has been in software testing for more than thirty years and is co-author of three books—*Software Inspection*, *Software Test Automation*, and *Foundations of Software Testing*. She founded Grove Consultants in 1989, but in 2008 returned to being an independent consultant. Dot was involved with starting testing qualifications in the UK and was a member of the working party that developed the ISTQB Foundation Syllabus. She is program chair for the 2009 EuroSTAR conference. Dot holds the European Excellence Award in Software Testing.



KEVLIN HENNEY is an independent consultant and trainer based in the UK. He provides consultancy and training in programming techniques, software architecture, and development process. Kevlin is co-author of two recent books on patterns, *A Pattern Language for Distributed Computing* and *On Patterns and Pattern Languages*.



DIDIER THIZY is the director of project management at Macadamian Technologies. In the past five years, he has led teams to build more than ten new products end to end and is currently overseeing the development of several new projects in the mobile and Web technology spaces. A PMP-certified project manager, Didier's expertise lies in training agile teams in distributed locations. He regularly contributes to the Macadamian monthly column, "The Critical Path," and maintains a blog at softwarepmp.blogspot.com.



ED WELLER is an SEI-certified High Maturity Appraiser for CMMI appraisals with nearly forty years of experience in hardware and software engineering. Ed is the principal of Integrated Productivity Solutions, a consulting firm focused on providing solutions to companies seeking to improve their development productivity. A regular columnist on StickyMinds.com, Ed can be contacted at edwardfellerIII@msn.com.



FOND FAREWELL

My husband is currently going through the car-buying process. It's never a particularly fun deed, but he is approaching it thoughtfully and pragmatically. He's done his research, saved up a large down payment, and now is trying to negotiate a good deal by working both online and face to face with dealers.

While I'm excited for my husband, who has never owned a new car, I have to admit I'm going to miss his old truck. She's been in the family for years, being passed from family member to family member as needed.

She's been a great truck and has been on countless adventures, including two cross-country road trips and numerous camping and canoeing weekends, but, unfortunately, we've found our vehicle requirements have changed and she no longer meets our needs.

So, while I'm sad to say goodbye, I'm excited that we will soon have a new vehicle that better complements our lifestyle. I can't wait to see where the new truck will take us!

Speaking of fond farewells and new beginnings, I'd like to announce a couple of changes to our editorial lineup. This issue we are saying goodbye to Code Craft, our long-time development-centric department, and adding a fresh voice to our magazine. Our new department, Inside Analysis, is written by experienced business analysts and focuses on industry issues and topics relevant to our analyst audience. But never fear fans of Code Craft, regular contributor Kevlin Henney will be adding his insight and development expertise to our blog roster! Check it out at blogs.stickyminds.com.

Also in this issue, if you've wanted to break into the mobile app arena, but weren't sure how—there's an article for that. In our cover story, "Software to Go," author Luis Carvalho offers some food for thought on how to increase your chance for success when developing software for mobile devices.

In "Feel the Burn," George Dinwiddie introduces the hot topic of burn charts—a popular method of monitoring the progress of work, which can also tell you when your project might have some problems.

Think you are ready to start automating test execution? Before you take the plunge, read "That's No Reason to Automate!" by Dorothy Graham and Mark Fewster. The authors explain that while your objectives might be great for testing, they may not translate to test automation.

As always, I hope you enjoy this issue of *Better Software* magazine. Drop me a note to let me know what you think of our new department.

Happy Reading!

Heather Shanholtzer
HShanholtzer@sqe.com




Automation Design Canvas

Enabling Developers and QA professionals to record automated tests with a point-and-click.

Download your 20-day Trial
www.ArtOfTest.com

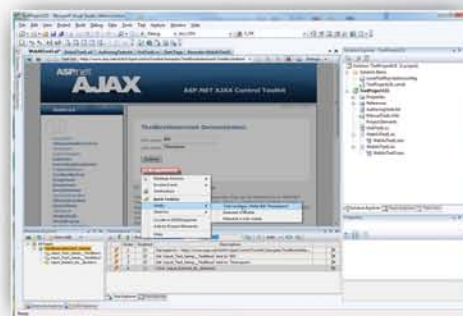
A New Era of Automated Web Testing

Experience unprecedented productivity using our point-and-click interface and build rich automation in minutes vs hours right inside your favorite development environment...  **Visual Studio**

Automation Design Canvas is built from the ground up to support modern web applications that heavily rely on Ajax! Users find it extremely cost-effective for testing ASP.NET applications. Automate scenarios you never thought possible! A quick start guide is all you need to get started automating your tests like a pro.

One team, One System

Promote collaboration by unifying your QA and Development teams under one system and one tool set with Automation Design Canvas. Boost productivity and communication! Automation Design Canvas is fully integrated in Visual Studio Professional & Team System 2008 as well as the entire life cycle management of tests that Visual Studio offers, including: test case management, test execution, source control, and reporting using Team Foundation Server.



Predicting the Past

by Lee Copeland

As a parent, you've probably heard, "Oh, I'll bet he'll look just like you when he grows up." Recently, I found some "face aging" software on the Internet and decided to give it a try. After paying a small fee, I loaded a photo of my five-year-old grandson and moved the age slider bar from five to ten and then to twenty and thirty. I was disappointed with the results. The program appeared to merely overlay the photograph with wrinkled skin. It's not what I thought he'd look like, but how could I be sure it was not accurately predicting his future appearance?

To test this program, I remembered an approach I used decades ago—I attempted to predict the past. (We know that the past is easier to predict than the future.) Back in my old big-iron mainframe days, one of my responsibilities was capacity planning. When a CPU costs millions of dollars, it is vital to accurately predict when it will be necessary to upgrade. Too early, and you waste the organization's money by purchasing unneeded capacity. Too late, and you waste the organization's money through processing inefficiencies. We developed a complex queuing theory model to predict performance based on anticipated load. How can such models be validated? By predicting the past. Design the model, give it yesterday's load data, and see if it correctly predicts yesterday's performance. If so, try last week's data, last month's data, and last year's data. If it predicts well over this range of input, it is reasonable to assume it will predict well into a similar future. If it does not, then modify the model and retest until it does.

So, I found a photo of me in my college days, uploaded it to the program, waited for it to process, and then moved the age slider bar. Again, more wrinkled skin was added, but it didn't resemble the present me. It was clear that since this program could not predict the now, I had no reason to believe it could pre-

dict the future.

How could we use this approach of "predicting the past" to validate our work? In software development, we often are expected to predict the future delivery, future cost, and future quality of the systems we develop. Most organizations do not do that well. Accurate prediction requires a process for prediction and data for that process to operate on. Where do our troubles typically lie?

First, data needed for prediction is often non-existent or of poor quality. Data that is missing, sparse, inaccurate, inconsistent, or ill-defined is not the foundation of accurate predictions.

Second, assuming that adequate data is available, the prediction process itself must be accurate and reliable. These processes (often mathematical algorithms) can fail if they don't take all of the important factors into account. Omitting even one factor may render the predictions worthless.

Finally, outside forces, beyond the scope of the prediction process, can affect the prediction's accuracy. Abandonment of the prediction process when the results are politically unacceptable is common. When the process cannot be abandoned (for example, if its use is subject to audit), finagling or fudging the data to arrive at the desired result, rather than the predicted result, is often done. If adequate data is available, and the prediction process is used, the results are sometimes ignored as hope triumphs over experience.

Developing an accurate prediction process is complex, time consuming, and difficult. This is because we have so little insight into the development

In software development, we often are expected to predict the future delivery, future cost, and future quality of the systems we develop. Most organizations do not do that well.

process itself. The ancient Greek physicians were not allowed to look inside the human body. They measured externally observable data without an understanding of how the system (body) worked internally. It is the same for us today. The system (development process) is essentially unknowable because of the variations of its practitioners. Because of the complexity of the development process, our prediction processes are often based on correlation (easy to measure) rather than cause and effect (difficult to measure).

Correlation merely indicates the strength of the linear relationship between two variables, not a cause-and-effect relationship between them. In a classic study described by the eminent statistician George Box [1], as the number of storks in London increased, the number of human births increased at an almost identical rate. An interesting correlation, but do you still believe that storks bring babies? Where cause and effect cannot be established, prediction processes are bound to fail.

But if you are gathering adequate data and attempting to create an accurate prediction process based on causality, remember to apply the "predicting the past" method to verify your results. Its success will give you confidence that you can predict the future. **{end}**

REFERENCES

- [1] Box, George; William G. Hunter; and J. Stuart Hunter. *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*. Wiley, 1978.

Testing the Contract Metaphor

by Kevlin Henney

Continuing a theme from my previous *Better Software* magazine columns “Programming with GUTs” (July/August 2008) and “GUT Instinct” (May/June 2009), how you partition and name your tests will affect how you think about their role and vice versa. What role do unit tests play? Do they check that the code is right? Or do they define what it means to be right? This is a subtle but important distinction. Unless you already know what *right* is, trying to test code for correctness is not particularly meaningful.

The common focus on tests’ being used to uncover defects means that their complementary role in specifying the code is often overlooked. Indeed, for unit testing, there is a case to be made that the quest for defects is actually a distraction rather than an attraction. [1]

Defining what we expect or require from a piece of code can be likened to a contract. Writing tests in a more contractual style changes their naming, their partitioning, and their readability, as well as their role in development.

Test Functionality, Not Functions

A common, but misguided, style for writing unit tests is to align test methods with methods under test, one for one. By attempting to focus on individual methods in isolation from one another, however, this procedural style diverts attention away from actual object usage. To use an object to achieve an end and to confirm an outcome involves a sequence of method calls, from the initial creation through to the final queries.

The `RecentlyUsedList` C# class from “Programming with GUTs” offered four accessible features: default construc-

```
Constructor
Add
Indexer
```

Listing 1a

tion, size query using a `Count` property, string addition using the `Add` method, and indexing using the subscript operator. Using NUnit, procedural test naming is shown in

listing 1a—not particularly enlightening, and perhaps also a little peculiar: The compiler-generated constructor is sufficient in this case, but the name implies that this is something the programmer must necessarily define. A test style that reflects propositions about the behavior gives a completely different quality of description, as can be seen in listing 1b.

```
InitialListIsEmpty
AdditionOfSingleItemToEmptyListIsRetained
AdditionOfDistinctItemsIsRetainedInStackOrder
DuplicateItemsAreMovedToFrontButNotAdded
OutOfRangeIndexThrowsException
```

Listing 1b



ISTOCKPHOTO

This reads like a list of specific behaviors rather than merely echoing the method names. Without even reading the body of the test methods, we already understand what is

```
CanBeCreated
KnowsWhetherItIsEmpty
KnowsHowManyItemsItHolds
DoesNotHoldDuplicates
ChecksIndexing
```

Listing 1c

expected. The names should be specific and refer to measurable outcomes, rather than list vague objectives. A counterexample would be the test names in listing 1c, which, although not incorrect, are vague and do not really tell us how a recently used list behaves.

Naming is not merely an act of labeling: Adopting a naming style based on propositions about usage and behavior changes the role of the tests and influences the partitioning of test cases.

Required and Ritual Naming Conventions

Test names are influenced by technical constraints and non-technical preferences. Groovy, with its out-of-the-box support for JUnit 3, was used in “GUT Instinct” to define code and tests for an `isLeapYear` predicate. JUnit 3 requires that test method names begin with `test`. This is not necessarily unreasonable, but on its own it is not enough to encourage—and can even discourage—names that reflect specification. In such

```
testThatYearsNotDivisibleBy4AreNotLeapYears
testThatYearsDivisibleBy4ButNotBy100AreLeapYears
testThatYearsDivisibleBy100ButNotBy400AreLeapYears
testThatYearsDivisibleBy400AreLeapYears
```

Listing 2a

cases it is more useful to consider `testThat` to be the prefix, as this naturally invites a proposition to follow it, as can be seen in listing 2a.

With frameworks, such as JUnit 4 and NUnit 2, that use metadata to denote test methods, naming is a little less constrained. Nonetheless, some programmers find ritual phrasing to be helpful, such as prefixing with `testThat` or `requireThat` or, in the case of behavior-driven development (see the StickyNotes for more information), using `should` somewhere in the test name, as shown in listing 2b.

```
yearsNotDivisibleBy4ShouldNotBeLeapYears
yearsDivisibleBy4ButNotBy100ShouldBeLeapYears
yearsDivisibleBy100ButNotBy400ShouldNotBeLeapYears
yearsDivisibleBy400ShouldNotBeLeapYears
```

Listing 2b

```
yearsNotDivisibleBy4AreNotLeapYears
yearsDivisibleBy4ButNotBy100AreLeapYears
yearsDivisibleBy100ButNotBy400AreLeapYears
yearsDivisibleBy400AreLeapYears
```

Listing 2c

The use of `should` can be helpful if specification-like naming is not the norm, but be aware that it can at times look a little indecisive and uncommitted. The word *should* is often used to qualify things as probable, suggested, or desirable but not actually definite, obligatory, or necessary. If you can do without ritual words, a direct, propositional naming style is less noisy and less ambivalent, as can be seen in listing 2c.

As Strunk and White note, “Make definite assertions. Avoid tame, colorless, hesitating, non-committal language.” [2]

The Contract Metaphor

Such a definitive and assertive approach sounds contractual, and with good reason. The tests contribute to the specification of the code, defining the expected interface that can be relied on. As Butler Lampson noted, “an interface is a contract to deliver a certain amount of service; clients of the interface depend on the contract, which is usually documented in the

interface specification.” [3]

The metaphor of contracts is quite a rich one, although it is often interpreted narrowly as no more than *programming by contract*. [4] Programming by contract frames the behavior of individual operations in terms of preconditions—which must be met by the caller if the operation is to execute successfully—and postconditions—which must be met by the operation following execution.

Programming by contract is simply one application of the contract metaphor, one that addresses some kinds of behavior but not others. In practice, the contract metaphor should be taken more broadly. There are many other approaches that are also contractual in nature, even if the word *contract* does not appear in their names. They vary in terms of effectiveness, effort, formality, executability, enforceability, scope, and so on. The style of testing we have been discussing is one such approach.

Returning to leap years, here is a textual definition of the contract:

Given `year`, which is an `int`, `isLeapYear` returns `true` if `year` is divisible by 4 but not by 100, with the exception of years also divisible by 400, otherwise it returns `false`. For example, `isLeapYear(2008)` and `isLeapYear(2000)` both return `true`, whereas `isLeapYear(2009)` and `isLeapYear(1900)` both return `false`.

Looking at the test names in listing 2c, we can see the contract teased out into its constituent parts. The examples form the basis of the assertions within each test method.

```
boolean isLeapYear(int year)
{
    boolean result
    ... // calculate result
    assert result == (year % 4 == 0 && (year % 100 != 0 || year % 400 == 0))
    return result
}
```

Listing 3a

```
boolean isLeapYear(int year)
{
    boolean result = year % 4 == 0 && (year % 100 != 0 || year % 400 == 0)
    assert result == (year % 4 == 0 && (year % 100 != 0 || year % 400 == 0))
    return result
}
```

Listing 3b

Contract and Converge

Would an approach more like programming by contract be as effective in expressing the contract for `isLeapYear` as the test-driven approach? This would entail checking the postcondition within the body of the method rather than driving the method from outside. Assertions would still be used but now to express an invariant rule rather than a specific outcome.

This would look something like listing 3a.

The limitation with this approach becomes obvious when we elaborate the calculation, as shown in listing 3b. The implementation is identical to the desired outcome. For code that is declarative and functional in nature, this is not uncommon: The implementation of a pure function is often the same as the specification of its result.

Given that the logic itself is actually the challenge, duplicating it does little to clarify the specification or to catch defects. Indeed, not only is this approach unlikely to catch any defects, it is likely to propagate them. Making the logic more verbose is unlikely to help; it merely spreads it over more lines of code. It is the essential complexity of the rule for leap years that is the issue, not the accidental complexity of the code. [5]

None of this implies that `isLeapYear` is not bound by a contract, just that the specific technique of programming by contract has little to offer in terms of expressing and checking this particular contract. The test-driven approach turns out to be far more effective for realizing the contract in code: The contract is decomposed into descriptive clauses (the names of the test methods) with illustrative examples attached (the body of each test method).

While there is often a tacit recognition that testing and specification are related, it is one thing to say that you can derive tests from a specification and quite another to say that the tests form part of the specification. **{end}**

REFERENCES

- [1] Feathers, Michael. "The Flawed Theory Behind Unit Testing." michaelfeathers.typepad.com/michael_feathers_blog/2008/06/the-flawed-theo.html
- [2] Strunk Jr., William and E.B. White. *The Elements of Style*, 3rd edition. Allyn & Bacon, 1995.
- [3] Lampson, Butler W. "Hints for Computer System Design." *ACM Operating Systems Review*, October 1983. research.microsoft.com/en-us/um/people/blampson/33-hints/webpage.html
- [4] Meyer, Bertrand. *Object-Oriented Software Construction*, 1st edition. Prentice Hall, 1988.
- [5] Henney, Kevlin. "The Accidental Complexity of Logic." *Better Software* magazine. May, 2008.



Has naming influenced the style of your unit tests? Does a specification-oriented approach change the quality of your tests?

Follow the link on the [StickyMinds.com](http://www.StickyMinds.com) homepage to join the conversation.

Sticky Notes

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

- Behavior-driven development



Find Words that Work at StickyMinds.com Blogs

Software Requirements

Accelerate Your Career & Empower Your Team

Pair two
courses
anywhere,
anytime
and save
\$500!



**NEW FALL 2009
SCHEDULE**



Relevant, Up-to-Date Content

Small Classroom Workshop Environment

Best Practices

World-Class Expert Instructors

**BUILD-YOUR-OWN
TRAINING WEEK**

Maximize the impact of your training by combining courses in one location to create a customized training week. Pair two courses and save up to \$500. For a complete list of courses available, visit www.sqetraining.com or call 888.268.8770 or 904.278.0524 for pairing discount options.



www.sqetraining.com

Mastering the Requirements Process

Build the Right Software the First Time **UPDATED**

Washington, DC

September 14–16, 2009

San Diego, CA

October 12–14, 2009

Requirements Modeling

Use Models to Improve Your Requirements Gathering and Systems Analysis **UPDATED**

Washington, DC

September 17–18, 2009

San Diego, CA

October 15–16, 2009



Let SQE Training come to you. For more information about on-site training courses, contact SQE Training at 904.278.0524 x212 or 233 or 888.268.8770 or email onsitetraining@sqe.com.

How Agile Practices Reduce Requirements Risk

by Ellen Gottesdiener

Every software project carries some risk, but many of these risks can be mitigated. That's true of problems related to product requirements—problems that are often cited as one of highest risks for any type of software project. Whether it is having unclear requirements, lack of customer involvement in requirements development, or defective requirements, these troubles are a major culprit in projects that go awry.

Project teams can make a difference by adopting and implementing agile practices. When implemented correctly, agile practices greatly mitigate the most common risks associated with requirements on software development projects. Adapting the requirements risks I discuss in my book *The Software Requirements Memory Jogger* [1], I will explain how agile practices act to mitigate the risks—and, therefore, provide the business value for which these practices aim.

Risk 1: Unrealistic Customer Expectations and Developer Gold-Plating

This is the risk that your customer's wishes will exceed what your team can deliver or that developers—in their sincere quest to satisfy their customers—will add unnecessary features.

How does agile address these problems? In agile projects, we chop up delivery expectations into short iterations—one- to three-week timeboxes. Each timebox begins with an iteration planning workshop in which the customer decides which work should be delivered.

The process is entirely transparent:

1. The customer states the goal or theme for the iteration.
2. The delivery team members state how much time they have to devote to the effort (i.e., their capacity, usually in work hours).
3. The customer selects the highest-priority requirements from the

backlog (the master catalog of work needed to build the product).

4. The desired requirements are further discussed and elaborated on, as needed.
5. The team estimates and tasks out the work.
6. The team and the customer explore risks and dependences.
7. The team makes an explicit commitment about which requirements will be delivered.

As an analyst and coach, I find that the key to this process is having each work item (also called a *story*) small and sharply defined. If you don't know the completion criteria up front—to assess whether a requirement is “done”—then the customer's expectations might be dashed or delivery team members might make (wrong) assumptions and add extras.

Throughout the iteration, the team checks on expectations by showing completed stories to the customer. At the completion of each iteration, team members show any stakeholders all the completed work in a demonstration and review.

Risk 2: Insufficient Customer Involvement

The most commonly cited project risk is a lack of engagement by customers. A precondition on agile projects is that we require the customer to participate throughout each iteration. As just described, the customer declares the iteration goal and work at the start, reviews completed work during the iteration, and attends the demo or review at the close of every iteration. In addition, the customer must always be available to answer questions about requirements.

When customers are less available, then domain-knowledgeable business analysts act as proxy customers to help with requirements analysis. Thus, business analysts become customers; they are delegated the decision-making authority



ISTOCKPHOTO

about requirements priorities. In other cases, I have seen business analysts act as coaches and aides to customers, helping them define concise and clear requirements, prune the ever-changing product backlog, analyze backlog items to prepare the team for the iteration planning workshop, and document requirements.

No matter who assumes the customer role, it is front and center on an agile project.

Senior-level customer involvement is also crucial, particularly on large, complex projects. These executives set the context for the product development effort by participating in product roadmapping to define the product vision and lay out which features will be delivered over time based on market and technology needs and constraints.

Risk 3: Poor Impact Analysis

It is rare to encounter products with fixed, clear requirements up front. Changes to requirements and shifting priorities can affect the sequence of work, introduce unforeseen rework, or create product defects.

Poor impact analysis involves not understanding the ways that new and evolving requirements affect the set of proposed requirements that make up the *baseline* (the traditional requirements term) or *backlog* (the agile term).

On agile projects, it's OK to change the backlog. Indeed, some teams agree

that any team member can revise the backlog at any time. Other teams allow only the customer or the business analyst to modify the backlog. Whichever arrangement the team agrees to, the point is that team members recognize that the backlog of work is dynamic.

The product backlog is continually analyzed and adjusted. The customer, often with an analyst and perhaps other team members, *prunes* the backlog. When pruning, impact analysis is key: Items are broken down, analyzed for their interdependences, shifted up or down in priority, re-estimated, removed, and reallocated to iterations or releases. This happens weekly on most agile teams. Analyzing the impact of changing requirements is part of the rhythm of successful agile teams.

Risk 4: Scope Creep

The uncontrolled expansion of requirements throughout the project is the highest risk of any software project [2]. In addition, the larger the product, the more requirements grow. Yet, scope creep might actually be considered “normal.”

Most software products present a wicked dilemma: The problem you are trying to solve is not fully understood until *after* it has been solved (i.e., some of the solution space lies within the problem space) [3]. If you cannot know what the solution is until you start to build the product, you benefit by starting to build it in small increments and then obtaining feedback to learn and adapt. This is the essence of agile development.

Some stability is, of course, necessary. Product goals, objectives, target market and users, and a product vision need to be articulated (agile teams do this as part of product and release planning).

It's OK to add new items or stories to the backlog as they arise. Agile teams manage scope creep by continually pruning the backlog.

The project's scope is defined at a high level but is not a binding contract. By working in short delivery cycles on a small subset of requirements, agile teams can better control scope. Every one to three months, they conduct release planning to adapt the requirements delivery plan over a longer time frame.

Risk 5: Defective Requirements

Requirements defects include missing, erroneous, conflicting, or ambiguous requirements, which can lead to a defective product. Even worse, it can lead to building the wrong product. On an agile project, small, concise requirements (stories) are sharply defined once the customer has chosen them from the backlog. Defining story “doneness” is essential. As mentioned earlier, the customer participates in iteration planning and is available throughout each iteration to answer requirements-related questions.

That leaves no wait time during which developers or testers make (wrong) assumptions about requirements. In addition, I like to have the team develop user acceptance tests as soon as work begins on each item. This form of validation is the best way to remove ambiguity from requirements.

A requirements defect also leads to excessive rework—revised code, additional testing, modified documentation, and premature or unnecessary analysis. On agile projects, we do not analyze backlog items until they move to the top of the backlog stack—when they are about to be pulled into an iteration planning workshop. This practice not only prevents us from analyzing requirements that will never be implemented but it also avoids rework caused by analyzing requirements prematurely. Additionally, many requirements are interdependent. When you analyze, build, test, and deliver a requirement, you learn things that will impact your understanding of related requirements. By waiting until the “last responsible moment” to conduct analysis, you are better informed and can tackle your analysis more efficiently.

Risk 6: New Processes and Tools

How do agile teams reduce the risks associated with using new requirements practices and tools? How do teams mitigate the normal risks of *any* change? They minimize these risks through feedback, metrics, and coaching.

Each day, the team shares feedback via a *standup meeting*. In that fifteen minutes, team members state what they

did yesterday, what they plan to do today, and what (if any) impediments they are experiencing. The team also gets customer feedback by showing its customers completed stories as soon as they are finished. The other key feedback mechanism is iteration *retrospectives*, sessions during which team members review self-correcting feedback and identify small, focused adjustments that will help them better integrate changing work practices.

A key metric for agile teams is the burn down chart, showing the rate at which stories or tasks are being completed, measured in hours per day. See this issue's “Getting the Most Out of Burn Charts” for more information.

Real Risk Reduction

Myths abound about how agile practices ignore or avoid good requirements practices and can increase requirements risks. In reality, agile done right decreases common requirements-related risks. Adapting agile practices can enable the team to act in rhythm with the dynamic nature of requirements development and facilitate the delivery of “solutions that meet business needs, goals, or objectives” [4]. **{end}**

REFERENCES

- [1] Gottesdiener, Ellen. *The Software Requirements Memory Jogger: A Pocket Guide to Help Software and Business Teams Develop and Manage Requirements*. GOAL/QPC, 2005.
- [2] Jones, Capers. “Social and Technical Reasons for Software Project Failure.” *CrossTalk*, June 2006, pp. 4-9.
- [3] DeGrace, Peter and Leslie Hulet Stahl. *Wicked Problems, Righteous Solutions: A Catalogue of Modern Software Engineering Paradigms*. PTR Prentice Hall, 1990.
- [4] International Institute of Business Analysis. *A Guide to the Business Analysis Body of Knowledge*, version 2.0, 2009.



How do you balance agile's imperative to define small, concise requirements in each iteration with the need to have a larger view of the entire product's requirements?

Follow the link on the **StickyMinds.com** homepage to join the conversation.

Three Kinds of Measurement and Two Ways to Use Them

by Michael Bolton

People often quote Lord Kelvin: “I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of *science*, whatever the matter may be.”[1] But, few note the sentence that precedes the passage: “In *physical science* the first essential step in the direction of learning any subject is to find principles of numerical reckoning and practicable methods for measuring some quality connected with it.” The missing sentence prompts some questions: *Are software development and testing sciences subject to the same kind of numerical measurement that we use in physics? If not, what kinds of measurements should we use? How could we think more usefully about measurement?*

Gerald M. (Jerry) Weinberg suggests thinking in terms of three broad categories. [2] *First-order measurement*, he says, is what we need to get started—“just adequate to the task of getting something built.” First-order measurement tends to be qualitative, fast, and inexpensive; it generally doesn’t require mechanisms or devices to enhance or extend the observation. In a recent conversation, Jerry told me that first-order measurements “are unobtrusive, or minimally obtrusive, and can be used without a whole lot of fuss. They help give you a lot of important information that can lead to other information or, in the best case, to immediate action if needed.” [3]

First-order measurement is what we’re doing most of the time as we’re driving a car. We look through the windows, listen to the engine, and feel the acceleration and deceleration. We make observations and comparisons without getting hung up on quantification. “The road is dry. It’s cloudy. There’s traffic



on the right and a car up ahead with its brake lights on.” First-order measurement suggests answers to the questions *What seems to be happening?* and *What should I do now?* In this situation, if you feel like you’re driving too fast, you probably are driving too fast. If so, first-order measurement is enough to inform an immediate and appropriate action: slow down.

Because it’s based on ongoing experience and feelings, rather than on careful experiments and controlled data intake, wise use of first-order measurement requires us to consider a number of possible interpretations of the meaning and significance of what we see. Suppose you feel like you’re driving fast, but not *too* fast. Now you observe a set of red and blue lights on the top of the car ahead. The extra data suddenly prompts you to realize that you’re uncertain about your relationship to the speed limit. The situation and first-order measurement prompt a different response in the form of questions: *What else do I need to know?* and *Where should I look?* At this point, you move into *second-order measurement* and refer to the speedometer.

Second-order measurement, says Jerry, is the kind of measurement that engineers use to tune relatively stable systems, making them cheaper, stronger, lighter, more reliable, faster—or slower, if that’s what’s desired. Second-order measurement focuses on questions like *What’s really happening?* and *How is it changing?* tending to be more quantitative, subject to more refined models, and generally busier than first-order measurement. It is often assisted by external instruments to supplement or refine direct sensory intake. In particular, metrics—mathematical functions that relate objects or events to numbers via a model—are second-order measurements.

Back in the car, second-order measurement is the kind of information that you obtain from looking at the dashboard. You note that your speed is forty-three miles per hour and that the posted limit is thirty-five miles per hour. Your quantitative, second-order measurement tells you that you’re above the legal limit. The apprehensive feeling in your gut, triggered by the combination of police car and the second-order measurement, informs a decision to slow down.

What of *third-order measurement*? That, says Jerry, is the kind of precise, highly quantitative measurement that supports the physicist's search for new natural laws. It helps us answer the question *What happens?* in a universal and general sense. But third-order measurement can be precise only because it tends to be about very simple systems (such as two interacting masses) or very simple models of complex systems (in which we choose to ignore many dimensions of the system, but analyze a very small number of dimensions very thoroughly). Perhaps most significantly, third-order measurement emphasizes and depends upon keeping messy human traits—variability, subjectivity, and values—out of the way. As noted in an important paper by Cem Kaner and Walter P. Bond, [4] using metrics and higher-order measurement wisely depends on *construct validity*—critical rigor in evaluating the models and the functions that form the basis for the measurement.

In Rapid Testing, we define a *control metric* as any metric that drives a decision. Some development groups standardize the decision to ship the product when it contains a low-enough threshold number of high-severity bugs. Others consider a program adequately tested if there's one positive and one negative test per "requirement" (meaning per line in a requirements document). Still others deem a test group "successful" if there is a low-enough percentage of rejected bug reports. By contrast, an *inquiry metric* is one that prompts a question: We have three open high-severity bugs—*What's the story there?* Jim and Mark are two days behind where we thought they'd be—*Do they need help?* The program managers are deferring a lot of problem reports—*Are the problems insignificant, or do we need more training because we don't understand the product?*

One of my recent clients rated the quality of its products and customer satisfaction with a basket of five second-order metrics. Each measurement collapsed months of work and tons of data into a single number. "Better" numbers earned praise; "worse" numbers earned a reprimand, so management meetings dragged on while people tried to explain

changes from last month's numbers—especially when things had gotten worse. At this company, schedules frequently slipped and shipments were often delayed. Yet when I asked testers the simple question: *What slows you down?* I got a wealth of information. They told me about broken and buggy builds, inadequate test environments, excessive emphasis on scripts that were out of date by the time the product arrived, and a lack of information about real customer needs. They also said they were wasting time collecting data that wasn't being used to help speed up development or testing, and they offered dozens of ways in which the numbers could be gamed.

A different client, also working on one-year project cycles, focused on questions like: *What happened this week? What did we get done? What problems did we run into?* Managers used personal contact—direct observation of and conversation with people—as their primary approach to assessing the project's status. They took a good number of quantitative measurements, but used them only as indicators to refine their initial assessments and to inform new first-order questions. The team made rough long-term estimates and more precise short-term estimates, dividing two-week cycles into tasks of two days or less, with clear deliverables that signaled completion. When tasks weren't finished in the estimated time, no one was punished; instead, everyone considered what he hadn't understood earlier, what he had learned, and what might inform a better estimate next time. Team members didn't collect metrics on things that weren't immediately interesting and important to them. They were interested in understanding the situation and optimizing the quality of the work, not in the appearances afforded by the metrics. They emphasized the game and the season over the box scores. And they consistently shipped high-quality products on time.

They did use one—and only one—control metric. When the amount of open problems exceeded a certain number, they stopped working on new features and fixed problems until the list was comprehensible and manageable again.

Jerry observes that in software engineering we seem obsessed with higher-order measurements. Why? He suggests that decisions about quality are political and emotional, based on discussions and decisions about whose values count and how much they count relative to one another. [5] Such issues are often distasteful to people who want to appear rational and "scientific," so we try to avoid those issues with appeals to higher-order measurement.

Each new software project involves a human context—interaction between different sets of clients, developers, tasks, and problems to solve, with high variability, contending values, and small sample sizes. In those environments, third-order measurement isn't achievable; it's an expensive distraction. That leaves us with cycles of first- and second-order inquiry measurement—not physics, but easily good enough to build and tune our systems. {end}

REFERENCES

- [1] Thomson, William (Lord Kelvin). "Electrical Units of Measurement." Popular Lectures and Addresses I (London, 1981-94).
- [2] Weinberg, Gerald M. *Quality Software Management, Vol. 2: First-Order Measurement*. Dorset House Publishing, New York, 1993.
- [3] Weinberg, Gerald M. Personal correspondence with the author, May 18, 2009.
- [4] Kaner, Cem and Walter P. Bond. "Software Engineering Metrics: What Do They Measure and How Do We Know." 10th International Software Metrics Symposium. Chicago, IL, 2004. www.kaner.com/pdfs/metrics2004.pdf
- [5] Weinberg, Gerald M. *Quality Software Management, Vol. 1: Systems Thinking*. Dorset House Publishing, New York, 1991.



What's your experience with observation and measurement in your organization?

▼

Follow the link on the StickyMinds.com homepage to join the conversation.

Avoiding Half-baked Discovery

by Didier Thizy

“Let’s get to the bottom line,” said Joe, the potential client, as he put his laptop bag on the boardroom table, just missing the plate of fresh blueberry muffins I’d picked up at my favorite bakery. “I’ve seen your proposal for adding ten features to the Widget 2.0 application, priced out at \$100,000.”

My spidey sense went tingly. “You have some changes?”

“We only have \$50,000. So we’ll take the best five features and call it a deal.”

I took a deep breath. This wasn’t going to be easy. I got the feeling that Joe had recently had a bad encounter with a software outsourcing contractor—maybe someone who didn’t take the strategic view that my company did. However symbiotic we tried to be, Joe saw us as parasites.

“Five features will cost you \$75,000,” I said, bracing for the backlash.

“Explain,” Joe said, crossing his arms.

I fell back on Stephen R. Covey’s habit five—*Seek first to understand, then to be understood*. I had to look at it from his point of view. I gestured to the plate of muffins. “I can see why that seems odd. You can get a dozen blueberry muffins for \$5.00, and six for \$2.50. Half as many; half the price. So if ten features cost \$100,000, that’s \$10,000 per feature. Therefore, your budget should cover five features.”

Joe nodded. “Exactly. I just want fewer muffins.”

“The problem is that software and muffins aren’t quite the same. For starters, the bakery ovens are already on.”

“Your ovens aren’t on?”

I ignored the hint of sarcasm in his voice. “Not in the same way. The bakery is in the business of baking bread, buns, cinnamon rolls—lots of other stuff, too. So, the baker gets to work in the morning, turns on the ovens, and starts cooking. For us, it’s different. A project is a discrete entity, not just another

batch. It’s like we have to turn the oven on each time we want to make muffins.”

“By ‘turn the oven on,’ what kind of things are you talking about?”

“Fixed project costs. You can cut features all you want, but we still have to set up the development environment, source control repository, and continuous integration system and configure the servers. We have to configure the UI, application, and data-level frameworks on which we’ll build the features. We have to agree on the overall UI look and feel. And that’s just the beginning.”

“Hmm.” Joe looked thoughtful.

“Then we have to find the bakers and the recipe—put together and train the team. Even if team members already know the technologies, they have to get up to speed on your application. They can’t start writing meaningful code on day one without any research into what’s already been done, the coding conventions, even just reading the code.”

“True.”

“There’s ongoing stuff throughout the project. Having meetings with you and other stakeholders, for example. Cutting half the features might cut back, but it won’t cut half the meetings. And team members have to have their meetings to talk to each other and stay coordinated throughout the project.

“Let me show you.” I grabbed a napkin from under the muffin plate and scribbled a long rectangular box on it. “Let’s say that setting up the project



ISTOCKPHOTO

takes \$25,000, leaving \$75,000 for new features.” I drew a line one quarter of the way through the box and wrote project costs in the small part.

“If you reduce your budget to \$50,000, all you have left is \$25,000 for new features.” I cut the box in half and scribbled out the unused part. “At best, you might get three or four of the ten you originally wanted.”

Joe frowned at the paper, seeing my point.

“Also, you’re assuming all the muffins are the same size. But in reality, one feature might take significantly more time than another.”

“That’s true.”

“I have a suggestion that could help identify ways we could get you the biggest bang for your buck,” I said. “We like to call it a ‘discovery phase,’ and we’ve found it’s useful to project man-

STORY LINES

- Before beginning an estimate, schedule a short discovery phase to help you understand project priorities and investigate areas of risk. It will reduce the uncertainties and make it easier and faster to prioritize features and tasks.
- When negotiating the schedule with stakeholders, remember that cutting features out of a project doesn't cut time out of the fixed costs.
- Discovery phases give you checkpoints where you and your stakeholders can re-assess whether you're a fit to work together and reset expectations before it's too late.

agers looking to hire a development partner. It's a short engagement where we work together to investigate the right mix of features, design, process, and team composition."

Joe eyed me skeptically. I could almost hear his thoughts. *Sounds like a great deal ... For you.*

"So," I asked. "When you say 'best five features,' what do you mean by 'best'?"

"I know which ones I want," Joe said.

"Are you going for the most cost effective? Most innovative? Or maybe the most time sensitive?"

"Kind of a combo," Joe replied, enigmatically.

"We research which features would interest your users most. We can look into cost-saving strategies. For example, we could architect and plan the project, then hand off to a global development team. Or maybe find some third-party or open source software that would reduce the cost."

Across the table, Joe seemed to open up. "You'd do that stuff?"

"Of course, we'd like it better if you'd contract the whole project with us, Joe, but we're not out to make you overspend your budget. We want your project to succeed almost as much as you do."

"I doubt that," he said.

"More than 60 percent of our revenue comes from repeat business. If we gouge our customers, that 60 percent would go away. So yeah, we want to help you, and we want the project to be a success. Then you'll come back to us."

Joe eyed me thoughtfully. "You know, I don't think the last guys we worked with thought that way."

"I got that feeling," I said, sliding the plate across the table. "Muffin?" **{end}**

Do your customers, stakeholders, or boss want it all but have trouble prioritizing features and functionality? How do you guide them through that process?

Follow the link on the StickyMinds.com homepage to join the conversation.

More time... Less stress...
eLearning is the perfect solution for training at your own pace.

Try our self-paced eLearning delivered right to your desktop.

No need to leave your desktop with eLearning. eMastering Test Design gives you a classroom experience with the convenience of self-paced instruction. People learn in different ways so we've combined the audio, video, text, graphics, examples, questions and exercises for the best web-based delivery possible.

Mastering Test Design: The Art and Science of Creating Test Cases

Classroom Value with the Convenience of Self-paced Instruction

- Same valuable information as the two-day classroom course

Test Practitioner with Travel, Time and Budget Constraints?

- Save money and time. Take this eLearning course from your own desktop

Course Features

- **24/7 access for 70 days:** Unlimited access for a full 70 days.
- **Access to expert test consultants:** Email testing experts during business hours
- **Exercises:** Help you immediately apply your new learning.
- **Assessment questions:** Help you evaluate your learning.
- **Reinforce your learning:** Questions linked to content allow you to reinforce your learning.
- **Powerful multimedia format:** Professionally narrated audio. "Step-into-the-classroom" video clips. Downloadable templates, Web links for additional learning, and hot links to a complete glossary of terms.
- **15 professional education contact hours/PDUs:** Professional education credits toward the PMI.

Take \$100 off the course fee AND get a \$50 Amazon.com Gift Card! Use promo code: BSMTD
Offer ends Aug. 31, 2009



www.sqe.com

TAKE THE DEMO TODAY AND SAMPLE THE COURSE CONTENT ! www.sqetraining.com/eLearning/EMTDDemo



Developing Applications for a Wireless World

by Luis Miguel Carvalho





In the technology sector, there is a constant push to be the first to market with the next cool thing. Mobile devices are today's new and exciting battleground. As with all things new, there is a lot of research and planning, platforms changing, and standards winning and losing the fight for survival. As a result, tools, techniques, and development languages are always evolving.

My first Symbian OS application took me almost two months to develop. The software development kit (SDK) had just been released, the development tools were a work in progress, and community help was practically nonexistent. I remember during the first weeks going through the SDK C++ header files to find out whether some APIs were available because there was no complete code in the development environment. Even worse was finding out that sometimes header files had API signatures but they were not implemented.

The Symbian platform has matured, and its SDK and tools have evolved. What took two months to develop six years ago now can be done in two weeks.

Working in the mobile software business, I have learned a few lessons that I would like to share to help you develop software for mobile devices. Some lessons I learned from others, but most are gleaned from personal experience. I don't consider these ideas to be eternal truths, and neither should you. Take them for what they are—guidelines that will evolve as mobile technologies progress.

Set the Target

The first thing you need to do is set your goal. This seems obvious, and it's certainly not mobile specific. But this is where most projects start to fail, and it is even more common in the mobile arena. Competition between hardware manufacturers, operating systems, mobile operators, developer bases, and distribution networks is at full strength in the mobile industry. This segmentation is both an opportunity and a deadly trap. Many startups can position themselves in the market either by creating a new

niche or by bringing specialized offers into existing fields, but if a strong plan and clear definition of the target are not in place, you can lose your way very quickly.

WHERE DO YOU WANT TO BE?

You might think all mobile applications are network agnostic, because the communication stacks of service gateways and devices transparently take care of all the magic of transferring bits between both ends. In most cases this is true, but do not make decisions based on this assumption.

No one is capable of testing in all possible network environments. So, what to do? Define the target audience for your application. The best choice is to start right in your own country. If the goal is more international, identify the countries you are going after next. Don't make this a long-term plan. In ten years, technologies will be different and you will be adjusting to new realities.

Make a plan identifying your target geographical areas, investigate mobile operators in that area, and research the most popular mobile devices. Once you have a short list, then you have a base from which to start making decisions.

HOW ARE YOU PLANNING TO DISTRIBUTE YOUR APPLICATION?

If you already have an established company with a Web site and the application is targeted for your existing customer base, then making it available for download is your best choice—less cost and quicker market availability.

If your application is tightly connected with services that mobile operators (AT&T, Sprint, Vodafone, T-Mobile, etc.) want to promote—such as messaging, phone management and

protection, games, and high data-consuming applications—you may want to sell directly to mobile operators. They will own your solution, promote it themselves, and share profits with you.

If you have broad goals and want to get increased visibility across international markets, you can choose to partner with established distribution networks. Distribution networks can range from platform specific—such as Microsoft Windows Mobile Catalog or Apple's App Store—to specialized companies with established relationships with mobile operators. Specialized companies can help cut the costs and gain broader visibility when moving to international markets. You can search online for these companies, but it's better to attend specialized conferences such as GSMA Mobile Works Congress and meet company representatives directly.

WILL YOU NEED MOBILE OPERATORS' INVOLVEMENT FOR DISTRIBUTION?

If your strategy is to sell directly to mobile operators, you need to plan how you're going to approach them. Depending on their size, mobile operators can be slow-moving machines, and opening doors is not a simple task.

If you choose this path, expect a lot of hard work (unless you golf with the CEOs) to go through the selection phase and get on their short list. Sharpen your presentation skills so that you make a good impression when the opportunity to show your application arises. And remember: Like you, mobile operators are in business to make money, so be sure you make a clear case on how your applications can help them achieve this.

WHICH MOBILE DEVICES ARE YOU TARGETING?

A number of different mobile operating systems are available. Windows Mobile, Symbian, iPhone, Brew, Palm, Blackberry, Mobile Linux, and Android are examples. Supporting all of the available technologies would give you an enormous advantage when it comes to distribution, but it's a bit unrealistic to set as an initial goal.

You need to decide on which devices to focus. By now, you probably know your target audience. Looking at which

phones mobile operators sell in that area gives an idea of what devices you need to target. Subscribe to specialized press and go to mobile conferences in your area, or visit international conferences for a wider perspective.

Depending on the type of operating systems you choose to support, there are technical requirements to follow and maintain throughout your application lifecycle. From development languages like C++, Objective C, .NET, J2ME, and Java to hardware specifications for screen sizes and touch capability, keyboards, sound, cameras, and accelerometers, you will need to have broad technical expertise.

Create both a short- and a long-term plan. Your short-term plan should focus on one operating system and the devices that support it. Make sure this covers the biggest slice of your target audience. If you want to make mobile operators happy, ensure they sell or plan to sell those phones. In the long-term plan, make a more elaborate list of what devices you want to support over the next two years. This short- and long-term approach will allow you to focus on getting a first version for specific devices while keeping your engineering team in touch with the fact that you want your solutions to work on a broader range of devices in the future.

IS THERE ANY MANDATORY CERTIFICATION OR APPROVAL PROCESS YOU MUST FOLLOW?

Most people don't think about this until it's too late. For applications that must be installed on a device, some operating systems enforce strict rules, requiring a certification or approval process. One example is the Symbian Signed service for Symbian devices. Not following these rules means your application won't work. With Windows Mobile, the annoying "not trusted" message appears during installation if you do not sign the application with a valid certificate. This generally leaves customers concerned.

From the verification processes to buying signing certificates, there will be costs involved. These costs increase if you fail the certification process or later submit new versions of the application.

Be sure to include them in your budget.

TIME FOR THE BITS ...

Some sort of master plan is already forming in your mind. Your plan is set and ready to make you rich. There's just one small detail in the middle called implementation. Yep, that still needs to happen, unless you are just selling ideas. Let's consider some technical aspects that I believe are important when planning the development roadmap for what is going to be the next great mobile product.

To Install or Not to Install?

By now you probably know your answer to this question, but just in case, here are some additional things to consider:

WEB BASED

Web-based applications can be faster to create and, if well planned and developed, will bring your product to a lot of mobile devices present and future. Every single phone today has a Web browser included, and mobile browser technology is improving with every version released. Some devices even come with more than one browser. There are devices rolling out with Opera Mobile preinstalled in addition to Mobile IE for Windows Mobile and S60 Browser for Symbian. HTC Touch is one such example.

Yet, there are still some things to consider. A connection must be available for the user to access your Web-based application. Even if this is becoming less of a problem, it still pays to ensure you cover all angles. For example, if your target customers are people who travel a lot, consider that sometimes they will not have access to any network and won't be able to use the product you are providing.

The application must adapt seamlessly to mobile-specific screen requirements; otherwise, it will result in a poor user experience. Just serving your regular Web pages probably won't work, even in today's most advanced browsers. Create an application that detects mobile devices and adapts your Web pages to smaller screens. This was a lot more difficult to do when phones had tiny screens and some didn't even support

“Your application must be made to be used by fingers—and those fingers can belong to a ten-year-old or to a wrestler.”

colors. Now, you may get by just by adapting to VGA screens. To confirm if this is true, look at the target devices in your plan and test your adapted Web site on those devices.

LOCAL

Local applications are installed on the devices. This is a good choice if you want to provide a better user experience with richer menus and interactive options that Web browsers don't provide. It also will be advantageous if the application uses local device information, such as contacts or calendar, or improved integration with device applications like the file explorer, images gallery, or camera. Depending on the devices your application will support, you may need to add some logic that will enable adaptation to different screen sizes, intermediate layers to separate presentation logic from device hardware-specific implementation, and adaptation modules to adjust common logic to device-specific languages. Some platforms—such as Windows Mobile, Symbian third and fifth editions, and iPhone—make these tasks simpler than other platforms, either because they only have one device in the market or because the operating system is kept consistent across different devices. One example of a good idea gone bad is J2ME. Although J2ME applications were to be widely supported by Java-enabled devices, in reality, there were many extensions added to the J2ME standards by different manufacturers. These extensions range from access to hardware—such as screen, sound, and camera—to contact management or image manipulation. As a result, it is difficult to make a single J2ME application work on multiple phones.

Wireless, but Net-less

So, what does this mean to mobile applications? When thinking about the workflow of the application and how all the pieces come together to make an incredible user experience, we must

always—and I really mean always—consider the “what if it goes wrong” scenarios. When working with slow or intermittent connections, there is a chance that you won't be able to perform online operations as smoothly as expected. From the beginning, the application design must take these possibilities into consideration.

For example, suppose your application monitors user travel information. Whenever the user takes a picture or creates a note, it sends that information to a Web site together with a tag containing GPS information of where the information was created. The user then can go to the Web site to see a map with all the places he visited and the photos or notes shown in the correct locations. What if a photo is taken or a note written, but the upload to the Web site is not possible? Photos and notes are already stored in the device, so they are not lost, and upload can be retried later. What about the GPS information? You could attach the GPS location to the photo, store it in the mobile device, and transmit it later when connectivity is restored. The best thing is to store a mapping between each photo or note in the device and its GPS info. The main idea is to think right from the start about what could go wrong. Solutions will flow from there.

Screens and More Screens

When moving from the desktop world, the first thing you need to adjust is how you conceive user interfaces. Screen dimension and resolution are hardware features that definitely influence the way you will plan and implement your application. Even within the same family of devices, screen characteristics change.

SMALL SCREENS, BIG IDEAS

The one thing mobile screens have in common is that they are smaller than desktop monitors. Everything about the screen design needs to be planned care-

fully so that every pixel is well used and contributes to a richer user experience. This is an area where size definitely matters. The smaller the screen, the more clever user interfaces must be.

The first thing you might be tempted to do is wrap your creative brain around the best possible way to fit all the cool features your application supports in an amazing and intuitive interface. Think about the most important features you're offering your client and minimize the visual clutter around them. Keep your client focused on the features that will benefit him the most. If your application is also a channel for making money—for example, by showing advertisements—make sure this is taken into consideration. Main features must be quick to access and simple to use. Advanced features—ones that add small value and are used by only a small percentage of your clients—are better off in a subarea of the application, or, in a more radical approach, you can choose to remove them completely.

Also important is that the perception of time is different when using a mobile application rather than a desktop. Users are less inclined to wait for screens, transitions, and operations to execute when looking at a mobile screen. On the desktop, they can easily multitask, leaving less room for boredom and suspicions that something went wrong. Never leave your clients bored, and never leave them unsure whether or not a selected action has completed.

DIFFERENT SCREENS, BIG HEADACHES

Now that you are thinking about smaller screens, a new variable adds a larger complexity into the mix. Besides being smaller, there are dozens of different screens available across mobile platforms. From square to rectangular formats and older 96x12 resolutions to newer 240x160, 240x320, 240x240, 640x480, and many others, screens are not all the same.

This is another difficulty for developers who need to create cross-device, cross-platform applications. Hopefully you have your plan already set so you know which devices you need to target in the short and long term. If you find that you must support different screen resolutions, gather the engineering experts and discuss the possibilities. If all your devices support a common development language like C++ or J2ME, create a common base for the application in which you abstract the graphical interface from business logic. This can help isolate a lot of your code when porting from one platform to another. Modular and cross-platform architectures might take longer to create, but in the long run, they will help you support additional devices.

One bit of advice is to use native UI controls such as edit controls, lists, and buttons whenever possible. This reduces your ability to create Flash- or Silverlight-like interfaces, but you get controls that scale to different size screens and look and feel like other applications for that device.

TOUCH SCREENS

From Windows Mobile to iPhone and from Palm to Nokia, every device needs a touch screen. I believe touch screens are an intuitive form of interacting with a device, but for them to work there must be a good combination of great tactile hardware and well-crafted applications. Not all touch screens do their job, which can make using touch devices a nightmare. And, not all application interfaces are easily adapted to the reality of touch. Imagine a combination of bad choices, such as an application that squeezes in a number of options on the main screen installed on a device with a low sensitivity touch screen. Having to press hard on the screen and still missing the right option leads to a poor user experience.

The user's natural tendency is to use his fingers to "touch" the screen. Making controls like button and list entries that require mandatory use of a stylus is not a good idea. Your application must be made to be used by fingers—and those fingers can belong to a ten-year-old or to a wrestler.

Most touch devices actually support multiple input methods—typically the touch screen and a keypad or keyboard. If you're a developer, you might be thinking about how to support both. The good news is that if you stick to native controls, this is taken care of for you—yet another point to keep you on this path.

Testing

Just like in any other software life-cycle, testing must be present throughout definition, development, and release phases.

From the start, it pays to validate all interface choices and assumptions. You can use a faster, low-tech solution such as making paper prototypes of all the screens, or a more high-tech prototype that will run on one target device without any functional logic. After you have a prototype ready, create a focus group representing your target user segment to validate it. Collect all feedback using surveys and video. This user feedback probably will be the most important information to use when finalizing the definition phase, just before going into full-speed development. You also can have more sessions throughout different development phases for intermediate validations.

Whatever the development model used, ensure unit testing is part of the process. Once created, these tests can run automatically to perform regression validation on every build of your application. This will require time overhead, but you'll be thankful once regression tests start finding bugs before they reach your clients. You can add more elaborate automatic tests to validate complex scenarios not checked by unit tests.

Automatic tests, as important as they are, will never replace end-to-end human testing. Make sure you reserve both people and time to run full end-to-end tests in all alpha, beta, candidate, and release versions. Your budget must include money to buy devices to be used in tests and possibly SIM cards for data plans. Try to establish agreements with hardware manufacturers or mobile operators for loaner equipment and services.

Don't get caught in the overtesting trap. Learning when to stop is as im-

portant as making testing a part of your work. Deadlines and budget are common factors that delimit the time you have to test your application. Testing beyond a certain point can delay delivery of an application and cause you to miss the ideal time to market.

Make a list of application features, detailing each of the components involved. For each component, note the test cost, time to fix bugs, and customer impact. Take these costs into consideration and mark each feature as high, middle, or low priority for testing. It is easy to have too many high-priority features. If this is the case, redo the exercise until the priority levels are evenly distributed across features.

Based on this list, you can create your testing plan, focusing first on high-priority features and moving down the scale. High-priority features should have a high-quality bar set: no failed test cases, high code coverage, and daily automated test runs. If necessary, low-priority features can have quality bar values decreased. One possible result is that early in the decision process you find that certain features cannot be implemented because there will be no time to test them during the development cycle.

Creating mobile applications has some unique characteristics but still is based on common computer science and programming paradigms. Although this article focuses on the mobile perspective, some of these ideas are not mobile exclusive. Others are specific to mobile development planning. But, lessons learned can, without a doubt, be applied to the process of planning and creating desktop, backend, automobile, and industrial embedded systems.

Whether you plan to bring an existing application to mobile phones, create new standalone or Internet-connected applications or games, or adapt Web portals to fit mobile screens, I hope the guidelines presented here help you through the process of bringing the next big thing into the wireless world. **{end}**

AGILE SOFTWARE DEVELOPMENT TRAINING

Accelerate Your Career & Empower Your Team

Pair two courses anywhere, anytime and save \$500!



NEW FALL 2009 SCHEDULE



Relevant, Up-to-Date Content

Best Practices

Small Classroom Workshop Environment

World-Class Expert Instructors

BUILD-YOUR-OWN TRAINING WEEK

Maximize the impact of your training by combining courses in one location to create a customized training week. Pair two courses and save up to \$500. For a complete list of courses available, visit www.sqetraining.com or call **888.268.8770** or **904.278.0524** for pairing discount options.



Improve your skills and help your organization increase its performance through targeted high-value training. Delivered by top software consultants, training through SQE Training is one of the best investments you can make to meet your business objectives.

AGILE SOFTWARE DEVELOPMENT TRAINING WEEK LOCATIONS

September 21-25, 2009

Boston, MA

October 12-16, 2009

San Diego, CA

November 8-10, 2009

*Orlando, FL**

*Only Scrum Master Certification is available in Orlando, FL.

Choose from 4 specialized training courses:

THREE-DAY COURSES (Monday - Wednesday)

- Scrum Master Certification
- Design Patterns Explained

TWO-DAY COURSES (Thursday - Friday)

- Practical Test-Driven Development
- User Stories and Estimation in Agile Development



Let SQE Training come to you. For more information about on-site training courses, contact SQE Training at 904.278.0524 x212 or 233 or 888.268.8770 or email onsitetraining@sqe.com.

FEEL THE

BURN

GETTING THE MOST
BURN CHARTS OUT OF

BY GEORGE DINWIDDIE



“HOW’S your project going?” Isn’t this the first question people ask? It’s natural and necessary to have

some idea how a project is proceeding. Some people want to know if it will be done on time. Others have fixed the delivery date and want to know how much functionality the system will have. Some people are looking for an early warning sign that changes need to be made. Everyone wants to peer into the future for reassurance that they’ll get what they want.

Burn charts are a simple method to monitor the progress of the work. They provide an easy to comprehend, visual representation of project progress. They can be visually extrapolated to make predictions about delivery date for fixed scope, or scope for a fixed delivery date, as shown in figures 1a and 1b. If this

visual extrapolation is difficult because of the shape of the line, then the chart is telling us other things about how the project is going. This may be a difficulty in delivering working software or a difficulty in deciding what the software should be. We’ll take a closer look at some of these scenarios a little later. First, we’ll examine some of the variations of these charts.

Measuring Work Remaining

If you’ve got a shovel and you’re spreading a pile of dirt, the size of the remaining pile shows you how much work you’ve got left to do. As you take shovelfuls off the top of the pile and spread them around, the height of the pile goes down over time. If we plot the height of the pile over time, we’ll see something like figure 2.

This is an example of a burn down chart. At each increment of time, we plot the work remaining. We can see the progress toward our final goal, which is to have no work remaining. In effect, we “burn down” the planned amount of work until there’s none left. If the rate of progress is at all consistent, then we can easily predict when the work will be finished. If we’ve got a fixed deadline, we’ll see if we’re going to meet that deadline.

Ways of Measuring Work

There are various ways of measuring work. For a physicist, work is measured in joules. If your manager asks how much work is left to do on your project, and you reply with a value in joules ... well, let’s hope your manager has a sense of humor.

BY HOURS, DAYS, WEEKS

A more common way of measuring work is with units of time. I do eight

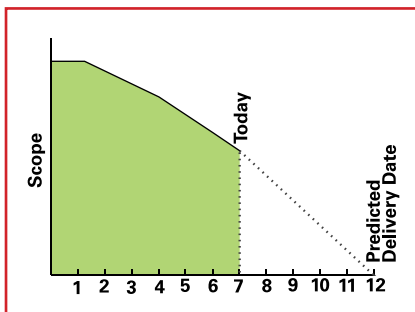


Figure 1a

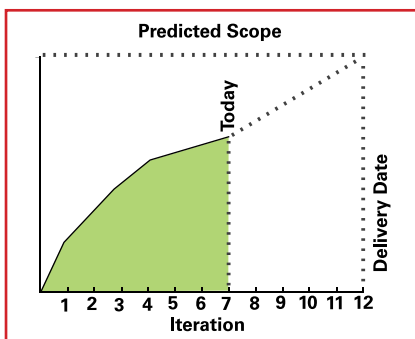


Figure 1b

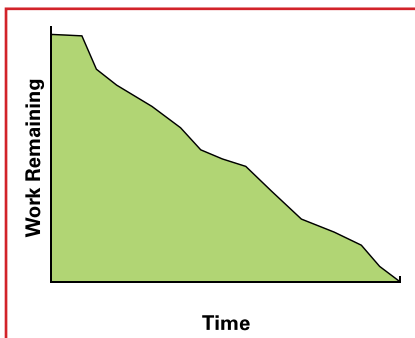


Figure 2

hours of work for a day's pay. This is a simple and easily understood measure. It's just that it does not measure what you may think it does.

Suppose I start work on time but then go for coffee while waiting for my computer to boot up. In the cafeteria, I run into my boss, and we spend forty-five minutes talking about company policies. After that, I log onto my email to catch up. Some of the emails require detailed responses, and it takes a couple of hours to collect the information. Now, down to work. I start up my IDE, start up my local server, re-deploy the code I'm writing, and start the application. I try the scenario that my current user story addresses. Oops, time for lunch. Right after lunch I've got a two-hour meeting. OK, where did I leave off? Oh yeah, here it is. I start to code, but I notice people are leaving. The workday is over and I've done eight hours of "work," but I haven't accomplished very much.

Of course, not every day is this bad, but time has an amazing tendency to slip away unnoticed. Not every hour corresponds to the same amount of work. It doesn't take much for hours—or days, weeks, months—to become a highly inaccurate way of measuring work.

People are not very accurate in making predictions, particularly about the future. If we're doing a repetitive task, such as spreading dirt from a big pile, then we might estimate how long it will take to spread the rest of the pile based on the amount we've done so far. Or, if we're doing a task that is essentially the same as one we've done before, we use that experience to estimate how long this instance will take.

With a complex task such as software development, we've got bigger problems. The work doesn't proceed at a steady pace. We're zipping along and suddenly run into an issue that stops us in our tracks. There's something we don't know, and we have to figure it out before we can continue. This makes a mess of our prediction for how long the task will take. With no apparent progress while we are discovering, we can't know when the stuck period will end.

This doesn't mean that you can't succeed by estimating in hours and days. By estimating in time units and checking

your actuals against your estimates, you will get better at estimating. At the individual level, there's some merit to that. But with a group of people, the chance of getting really good at estimating is lessened, the cost of doing so is increased, and the payoff is uncertain. The goal of a software development team is building valuable software. It's not clear that better estimating will increase either the quantity or quality of the software produced, but it is clear that the estimation ability will be disturbed every time there's a change in the team.

Adherents of Scrum recommend that the team estimate each task and then daily re-estimate the remaining amount of work on each task in order to calculate the total hours of work remaining. [1] This may provide more precision, but it creates a lot of overhead work and doesn't produce the software any faster. And, by taking up time and energy, it delays delivery.

BY STORY POINTS

I recommend a different approach. Divide the work into user stories—small but functional slices of the application (see the StickyNotes for more information). Assign each user story a simple, relative estimate. For example, assign each of the simple stories one point. Assign the stories that seem about twice as hard two points. Estimate your work in story points and track the number of story points left to do. You don't "get credit" for a story point until it is completely done—coded, tested, acceptable to the customer, and ready for potential deployment. This gives you a reliable indication of work accomplished—what you really want to know—rather than effort expended.

Don't worry about improving your estimating accuracy. In a large project, there's probably enough random perturbation that you'll never achieve the accuracy you would like. Remember, our desired output is working software, not accurate estimates. The estimates just have to be consistent enough to use for planning purposes. While our estimates may be off by 50 percent or even 100 percent, if we expect to accomplish the same estimated amount next iteration as we did this iteration, we'll be on target.

Variable Scope

The simple burn down chart uses the zero point of the y-axis as the goal line. This works fine as long as the goal doesn't change.

If you don't keep the scope of work constant, then from time to time you're likely to get a burn down chart like that shown in figure 3. What happened here? Why did the amount of work remaining increase rather than decrease? Did more

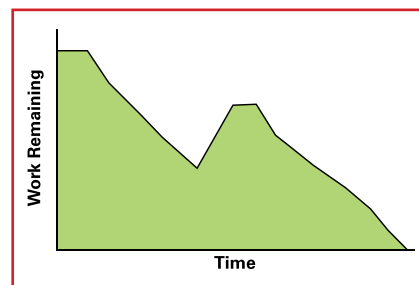


Figure 3

work get added? Within a single iteration, this shouldn't happen but sometimes does. Over a longer period of time, it's generally expected that new work will be added.

Did unforeseen tasks come to light? If you're burning down hours and re-estimating the remaining work on each task, as frequently recommended in Scrum books and articles, then you might get such a thing.

Did a story move backward? Perhaps it was thought to be done, but a new scenario was discovered and the story was handed back to the developers for more development. Or, perhaps the testers found a bug that had escaped development.

When the burn down chart curves upward like this, you no longer can predict when the work will be done. If the scope changes, we can no longer trust the goal line. If the measure of work remaining proves unreliable, then we're losing the ability to predict when the current scope of work will be done. We can't extrapolate the burn down chart to the goal line.

BURN BELOW ZERO

One way to separate changes in scope and progress is to use a burn down chart with a variable floor, as shown in figure 4. In other words, the work remaining is plotted as before, but the goal line

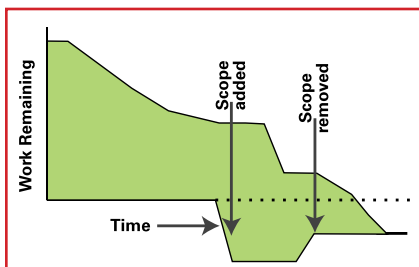


Figure 4

may vary from the zero point. If scope is added, the goal line moves below the zero point. If scope is removed, the goal line moves up. We can see our continued progress, but the intersection with the goal changes. Or, to reach completion sooner, we may reduce scope and bring the goal closer (see the StickyNotes for more information).

This seems to me a little complicated to draw, especially if the scope changes very much. It also presupposes that you know the goal line at the very start, which I find difficult to do. I generally find that the addition of new story points continues well into development, if not to the very end. I prefer not to use this chart, but people have used it effectively when the scope of work is changed during an iteration. I also prefer not to do that, but there are times when it's appropriate—when the team has underestimated and is running out of things to do, the team has overestimated and clearly needs to cut scope, or an emergency occurs.

BURN UP

A good way to indicate variable scope is to use a burn up chart. [2] This is like a burn down chart turned upside down.

Instead of tracking work remaining, a burn up chart tracks work completed. The goal line can be moved, and the difference between it and the work completed will give you an estimate of the work remaining. In the burn up chart in figure 5, you can see the addition of scope over time. Periodically, the goal takes a step upward. Each of the upticks in the goal line may represent a major feature's being defined as stories and estimated. Or, the upticks may indicate scope that was assumed but is now being made explicit. The scope of work also may be adjusted downward to meet a particular release date or because the least important

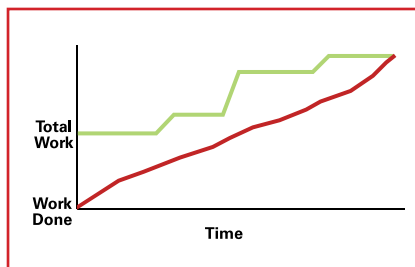


Figure 5

functionality is being trimmed.

For a fixed scope of work, a burn up chart doesn't match the simplicity of a burn down chart. But for variable scope, as most projects are over any extended period of time, the burn up chart gives a clear indication of the progress so far and a visual prediction of the finish date.

My preference when doing iterative software development is to use a burn down chart for an iteration, which has a short time span and a fixed scope, and a burn up chart for release planning or other longer-term project planning, as shown in figures 6a and 6b.

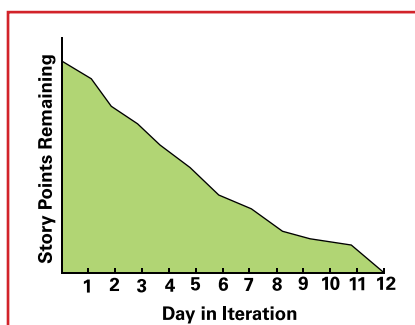


Figure 6a

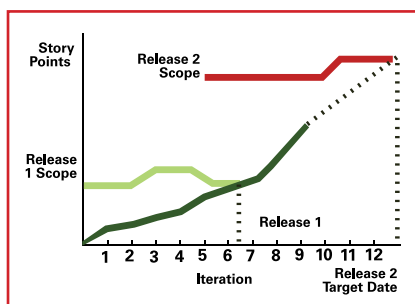


Figure 6b

Reading the Charts

A burn down or burn up chart does more than track progress. By examining the graph, we can make inferences about how the work is progressing. Figures 7a through 7e show some examples of burn down charts tracking progress in iterative development.

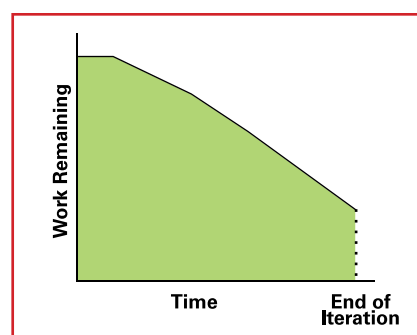


Figure 7a

TOO MUCH WORK

Figure 7a shows a burn down chart that doesn't reach the goal line. At the end of the iteration, there's work left unfinished. This may be an indicator of an unusual problem in this iteration, but if it happens frequently, I would interpret it as a sign of overcommitment.

Frequently, software development teams are overly optimistic about their capabilities. They know how to do the things that they see need to be done. Often they don't remember or consider the times they got stuck, either on a hard problem or waiting for some external dependency. They may ignore the time spent doing things other than creating code. And when they come up a little short at the end, they may commit to even more work during the next iteration to catch up.

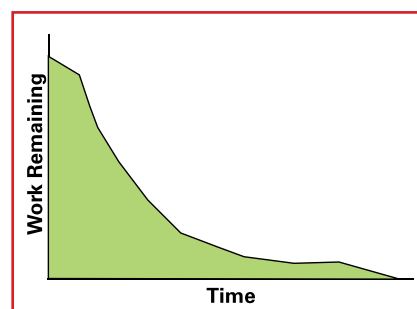


Figure 7b

SANDBAGGING

Some software development teams go in the other direction. Not wanting to miss their commitments and disappoint their stakeholders, they are conservative in their commitments. Figure 7b shows a steady progress and then slacking off when reaching the goal seems assured. Or, this chart could mean that the later stories were estimated low, relative to the

THE RULE OF THREE

Gerald M. Weinberg, in his book, *The Secrets of Consulting*, states the Rule of Three: If you can't think of three things that might go wrong with your plans, then there's something wrong with your thinking. This rule is applicable to much more than planning. It particularly applies to reading burn down charts. The possibilities that I offer are common ones I've seen, but they are offered without any knowledge of the context of your project. You'll need to discern what your burn down charts might be telling you. After you have an answer, try to think of at least two more. Then, armed with these possibilities, try to check it out to see which, if any, reflect the reality around you.

work they required. Or, it could mean that technical debt incurred in the earlier stories slowed down the development of later ones. It's often the case that a single graph of the iteration could be telling one of several stories. You may need to look at other information to clarify your understanding.

Another possibility: A team that never misses its commitment is a team that isn't pushing the boundaries. A high-performing team should always be testing where the line is between too much and too little work. This is much like a sailor who heads closer to the wind until the sail just starts to luff and then bears off to achieve maximum speed. The wind and waves are always changing, so this test needs to be done repeatedly and continually. So it is with software development. Try to push for just a little more, but be sensitive for signs that you're tempted to cut corners or leave something not quite done. Then, back off slightly.

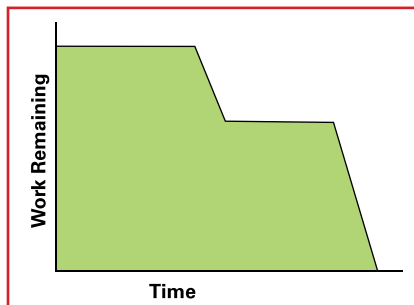


Figure 7c

STORIES TOO BIG

Figure 7c shows a picture where there's no indication of progress for a long time; then, suddenly, there's a big chunk accomplished all at once. The most likely cause for this is "big stories." Often, teams accustomed to a serial project lifecycle—with requirements definition, design, implementation, and then testing—have a difficult time breaking work down into small but valuable stories. Instead, they do "mini-waterfalls" on larger chunks of work, and they break the work down by architectural layers instead of by functionality. As a result, they bundle a lot of work together into one unit.

Doing work in large chunks makes real progress hard to see. Some people

try to counter this by estimating progress on unfinished pieces of work. It's easy to fool yourself about the amount of progress, though. There is a danger of creating the old situation of having done 90 percent of the work but the 10 percent remaining takes 90 percent of the time. It's much more reliable to judge progress by functionality that's easily tested to be complete or not. I like to see progress every day. Otherwise, I think I need to look for a bottleneck that's halting progress. This is just a small instance of developers "going dark" and no one else knowing what's happening.

Large stories also prevent the product owner from effectively steering the development of the product. Such a story may contain many little details that are "nice to haves" rather than "must-haves." If they're all lumped together with must-have functionality, the product owner must accept or reject the story altogether. This will mean putting these nice to haves into the product ahead of must haves in other stories.

What really works, once you learn how to do it, is to create very thin slivers of functionality. I call these *vertical slices* because they slice through the architectural layers from the user interface all the way down to the database. (This is the typical description of layers for business systems, but other types of software will have different layers.) This has been called "a walking skeleton" [3] or "firing tracer bullets through the application." [4] Each thin sliver doesn't do much, but it works all the way through. In addition to these slices of functionality, there may be stories that only modify existing functionality.

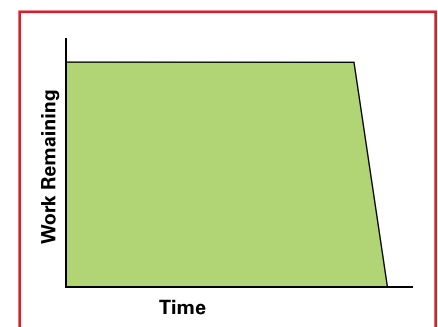


Figure 7d

BIG WORK IN PROGRESS

Figure 7d could be telling us that there's one huge story, the worst case of

**“Unfinished work has a cost (the work done so far)
but no value (we can’t deliver it to the customer).**

**It’s far better to finish a small amount of work than
to start a large amount.”**

“stories too big.” Or, it could be telling us that all of the stories are being worked simultaneously. Either of these is an indication that a lot of work is in progress at one time. Another alternative, that the developers were doing other work rather than the requested stories, would also produce a burn down chart like this.

What’s the problem with this? One issue is that you’re not sure of completing the stories. What if you reach the end of the iteration and all of the stories are “90 percent done,” but there’s no working business value to deliver to the product owner?

Unfinished work has a cost (the work done so far) but no value (we can’t deliver it to the customer). It’s far better to finish a small amount of work than to start a large amount. By dividing the story into small, functional slices, we can see the progress being realized. If we then start work on all of those small stories at once, we lose that advantage. It’s better for the team to swarm over each story to get it done and then start on the next one. The goal should be to have as few stories in progress at a time as you can work on productively.

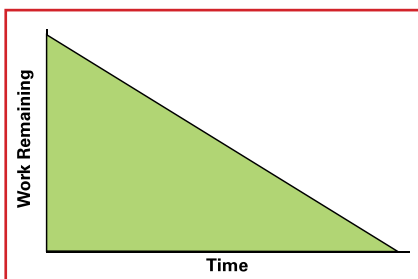


Figure 7e

COOK THE BOOKS

One of the most insidious burn down charts is the one where the progress line is arrow straight from start to finish, as shown in figure 7e.

While this *could* happen occasionally, it most likely means that someone is cooking the books. He may be reporting the progress that he perceives people want to see rather than an honest measure of how the project is proceeding. This is especially easy to do when tracking progress in task hours rather than story points. It also can be done by cutting corners on stories and claiming they’re done when the code is still a mess internally.

Why would this happen? There may be a number of contributing factors:

- The person (or software) drawing the burn down chart may mark a line to be able to tell easily if development is lagging behind or getting ahead of the average expected rate. Such a line can be interpreted by others as the “ideal” rate.
- The team may feel that it is being judged on the basis of the chart, so team members want it to look good.
- It may be that progress is being measured more by effort than by accomplishment. Systems that automatically calculate time remaining on a task are prone to this.

When a “good looking” burn down chart becomes the focus, then the ability to use it to understand and manage development is lost.

It’s Your Choice

Burn down and burn up charts offer considerable variety in the ways they can display progress so it can be understood at a glance. They’re highly adaptable to your context—you can track work with whatever measurement makes sense in your environment. You can track it over

short or long periods of time—or both. Do you have a fixed scope? Then a burn down is a good fit. Do you have a well-understood scope with relatively small adjustments? Either the variable floor burn down chart or a burn up chart would be appropriate. Are you working in a continuous flow, continuing to add to the backlog as you develop? Then a burn up chart is probably the best choice.

The important point is that your burn chart should reflect an objective reality, not wishes and hopes. It should be built with data that is measured, not estimated. Post your chart prominently where you will look at it every day. In this way, it can tell you at a glance whether your wishes and hopes are likely to be realized. If it tells you something else, it will give you the clues you need to take some corrective action. **{end}**

REFERENCES

- [1] Schwaber, Ken. “Work Burn Down.” www.controlchaos.com/about/burndown.php
- [2] Brewer, John. “Burn-down vs. Burn-up Charts.” tech.groups.yahoo.com/group/extreme_programming/message/81856
- [3] Cockburn, Alistair. “Walking Skeleton.” alistair.cockburn.us/Walking+skeleton
- [4] Hunt, Andy and Dave Thomas. “Tracer Bullets and Prototypes.” Artima Developer, 2003. www.artima.com/intv/tracer.html

Sticky Notes

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

- User stories
- Variable floor burn down charts



THAT'S NO REASON TO AUTOMATE!

Why Good Objectives Are Critical
to Test Execution Automation

by Dorothy Graham
and Mark Fewster

“WHY AUTOMATE?” This seems such an easy question to answer; yet many people don’t achieve the success they hoped for. If you are aiming in the wrong direction, you will not hit your target!

This article explains why some testing objectives don’t work for automation, even though they may be very sensible goals for testing in general. We take a look at what makes a good test automation objective; then we examine six commonly held—but misguided—objectives for test execution automation, explaining the good ideas behind them, where they fail, and how these objectives can be modified for successful test automation.

Good Objectives for Test Automation

A good objective for test automation should have a number of characteristics. First of all, it should be *measurable* so that you can tell whether or not you have achieved it.

Objectives for test automation should *support* testing activities but should not be the same as the objectives for testing. Testing and automation are different and distinct activities.

Objectives should be *realistic* and *achievable*; otherwise, you will set yourself up for failure. It is better to have smaller-scale goals that can be met than far-reaching goals that seem impossible. Of course, many small steps can take you a long way!

Automation objectives should be both *short* and *long term*. The short-term goals should focus on what can be achieved in the next month or quarter. The long-term goals focus on where you want to be in a year or two.

Objectives should be regularly *revised* in the light of experience.

Misguided Objectives for Test Automation

OBJECTIVE 1: FIND MORE BUGS

Good ideas behind this objective:

- Testing should find bugs, so automated testing should find them quicker.
- Since tests are run quicker, we can run more tests and find even more bugs.

- We can test more of the system so we should also find bugs in the parts we weren’t able to test manually.

Basing the success of automation on finding bugs—especially the automation of regression tests—is not a good thing to do for several reasons. First, it is the quality of the *tests* that determines whether or not bugs are found, and this has very little, if anything, to do with automation. Second, if tests are first run manually, any bugs will be found then, and they may be fixed by the time the automated tests are run. Finally, it sets an expectation that the main purpose of test automation is to find bugs, but this is not the case: A repeated test is much less likely to find a new bug than a new test. If the software is really good, automation may be seen as a waste of time and resources.

Regression testing looks for unexpected, detrimental side effects in unchanged software. This typically involves running a lot of tests, many of which will not find any defects. This is ideal ground for test automation as it can significantly reduce the burden of this repetitive work, freeing the testers to focus on running manual tests where more defects are likely to be. It is the testing that finds bugs—not the automation. It is the testers who may be able to find more bugs, if the automation frees them from mundane repetitive work.

The number of bugs found is a misleading measure for automation in any case. A better measure would be the percentage of regression bugs found (compared to a currently known total). This is known as the defect detection percentage (DDP). See the StickyNotes for more information.

Sometimes this objective is phrased in a slightly different way: “Improve the quality of the software.” But identifying bugs does nothing to improve software—it is the *fixing* of bugs that improves the software, and this is a development task.

If finding more bugs is something that you want to do, make it an objective for measuring the value of testing, not for measuring the value of automation.

Better automation objective: Help tes-

ters find more regression bugs (so fewer regression failures occur in operation). This could be measured by increased DDP for regression bugs, together with a rating from the testers about how well the automation has supported their objectives.

OBJECTIVE 2: RUN REGRESSION TESTS OVERNIGHT AND ON WEEKENDS

Good ideas behind this objective:

- We have unused resources (evenings and weekends).
- We could run automated tests “while we sleep.”

At first glance, this seems an excellent objective for test execution automation, and it does have some good points.

Once you have a good set of automated regression tests, it *is* a good idea to run the tests unattended overnight and on weekends, but resource use is not the most important thing.

What about the value of the tests that are being run? If the regression tests that would be run “off peak” are really valuable tests, giving confidence that the main areas of the system are still working correctly, then this is useful. But the focus needs to be on supporting good testing.

It is too easy to meet this stated objective by just running any test, whether it is worth running or not. For example, if you ran the same one test over and over again every night and every weekend, you would have achieved the goal as stated, but it is a total waste of time and electricity. In fact, we have heard of someone who did just this! (We think he left the company soon after.)

Of course, automated tests can be run much more often, and you may want some evidence of the increased test execution. One way to measure this is using equivalent manual test effort (EMTE). For all automated tests, estimate how long it would have taken to run those tests manually (even though you have no intention of doing so). Then each time the test is run automatically, add that EMTE to your running total.

Better automation objective: Run the most important or most useful tests, employing under-used computer resources when possible. This could be partially

measured by the increased use of resources and by EMTE, but should also include a measure of the value of the tests run, for example, the top 25 percent of the current priority list of most important tests (priority determined by the testers for each test cycle).

OBJECTIVE 3: REDUCE TESTING STAFF

Good ideas behind this objective:

- We are spending money on the tool, so we should be able to save elsewhere.
- We want to reduce costs overall, and staff costs are high.

This is an objective that seems to be quite popular with managers. Some managers may go even further and think that the tool will do the testing for them, so they don't need the testers—this is just wrong. Perhaps managers also think that a tool won't be as argumentative as a tester!

It is rare that staffing levels are reduced when test automation is introduced; on the contrary, more staff are usually needed, since we now need people with test script development skills in addition to people with testing skills. You wouldn't want to let four testers go and then find that you need eight test automators to maintain their tests!

Automation supports testing activities; it does not usurp them. Tools cannot make intelligent decisions about which tests to run, when, and how often. This is a task for humans able to assess the current situation and make the best use of the available time and resources.

Furthermore, automated testing is not *automatic* testing. There is much work for people to do in building the automated tests, analyzing the results, and maintaining the testware.

Having tests automated does—or at least should—make life better for testers. The most tedious and boring tasks are the ones that are most amenable for automation, since the computer will happily do repetitive tasks more consistently and without complaining. Automation can make test execution more efficient, but it is the testers who make the tests themselves effective. We have yet to see a tool that can think up tests as well as a human being can!

The objective as stated is a management objective, not an appropriate objective for automation. A better management objective is “Ensure that everyone is performing tasks they are good at.” This is not an automation objective either, nor is “Reducing the cost of testing.” These could be valid objectives, but they are related to management, not automation.

Better automation objective: The total cost of the automation effort should be significantly less than the total testing effort saved by the automation. This could be partially measured by an increase in tests run or coverage achieved per hour of human effort.

OBJECTIVE 4: REDUCE ELAPSED TIME FOR TESTING

Good ideas behind this objective:

- Reduce deadline pressure—any way we can save time is good.
- Testing is a bottleneck, so faster testing will help overall.
- We want to be quicker to market.

This one seems very sensible at first and sometimes it is even quantified—“Reduce elapsed time by X%”—which sounds even more impressive. However, this objective can be dangerous because of confusion between “testing” and “test execution.”

The first problem with this objective is that there are much easier ways to achieve it: run fewer tests, omit long tests, or cut regression testing. These are *not* good ideas, but they would achieve the objective as stated.

The second problem with this objective is its generality. Reducing the elapsed time for “testing” gives the impression we are talking about reducing the elapsed time for testing as a whole. However, test execution automation tools are focused on the execution of the tests (the clue is in the name!) not the whole of testing. The total elapsed time for testing may be reduced only if the test execution time is reduced sufficiently to make an impact on the whole. What typically happens, though, is that the tests are run more frequently or more tests are run. This can result in more bugs being found (a good thing), that take time to fix (a fact of life), and

increase the need to run the tests again (an unavoidable consequence).

The third problem is that there are many factors other than execution that contribute to the overall elapsed time for testing: How long does it take to set up the automated run and clear up after it? How long does it take to recognize a test failure and find out what is actually wrong (test fault, software fault, environment problem)? When you are testing manually, you know the context—you know what you have done just before the bug occurs and what you were doing in the previous ten minutes. When a tool identifies a bug, it just tells you about the actual discrepancy at that time. Whoever analyzes the bug has to put together the context for the bug before he or she can really identify the bug.

In figures 1 and 2, the blocks represent the relative effort for the different activities involved in testing. In manual testing, there is time taken for editing tests, maintenance, set up of tests, executing the tests (the largest component of manual testing), analyzing failures, and clearing up after tests have completed. In figure 1, when those same tests are automated, we see the illusion that automating test execution will save us a lot of time, since the relative time for execution is dramatically reduced. However, figure 2 shows us the true picture—total elapsed time for testing may actually increase, even though the time for test execution has been reduced. When test automation is more mature, then the total elapsed time for all of the testing activities may decrease below what it was initially for manual testing. Note that this is not to scale; the effects may be greater than we have illustrated.

We now can see that the total elapsed time for testing depends on too many things that are outside the control or influence of the test automator.

The main thing that causes increased testing time is the quality of the software—the number of bugs that are already there. The more bugs there are, the more often a test fails, the more bug reports need to be written up, and the more retesting and regression testing are needed. This has nothing to do with whether or not the tests are automated or manual, and the quality of the software

is the responsibility of the developers, not the testers or the test automators.

Finally, how much time is spent maintaining the automated tests? Depending on the test infrastructure, architecture, or framework, this could add considerably to the elapsed time for testing. Maintenance of the automated tests for later versions of the software can consume a lot of effort that also will detract from the savings made in test execution. This is particularly problematic when the automation is poorly implemented, without thought for maintenance issues when designing the testware architecture. We may achieve our goal with the first release of software, but later versions may fail to repeat the success and

may even become worse.

Here is how the automator and tester should work together: The tester may request automated support for things that are difficult or time consuming, for example, a comparison or ensuring that files are in the right place before a test runs. The automator would then provide utilities or ways to do them. But the automator, by observing what the tester is doing, may suggest other things that could be supported and “sell” additional tool support to the tester. The rationale is to make life easier for the tester and to make the testing faster, thus reducing elapsed time.

Better automation objective: *Reduce the elapsed time for all tool-supported*

testing activities. This is an ongoing objective for automation, seeking to improve both manual and existing automated testing. It could be measured by elapsed time for specified testing activities, such as maintenance time or failure analysis time.

OBJECTIVE 5: RUN MORE TESTS

Good ideas behind this objective:

- Testing more of the software gives better coverage.
- Testing is good, so more testing must be better.

More is not better! Good testing is not found in the number of tests run, but in the *value* of the tests that are run. In fact, the fewer tests for the same value, the better. It is definitely the quality of the tests that counts, not the quantity. Automating a lot of poor tests gives you maintenance overhead with little return. Automating the best tests (however many that is) gives you value for the time and money spent in automating them.

If we do want to run more tests, we need to be careful when choosing which additional tests to run. It may be easier to automate tests for one area of the software than for another. However, if it is more valuable to have automated tests for this second area than the first, then automating a few of the more difficult tests is better than automating many of the easier (and less useful) tests.

A raw count of the number of automated tests is a fairly useless way of gauging the contribution of automation to testing. For example, suppose testers decide there is a particular set of tests that they would like to automate. The real value of automation is not that the tests are automated but the number of times they are run. It is possible that the testers make the wrong choice and end up with a set of automated tests that they hardly ever use. This is not the fault of the automation, but of the testers’ choice of which tests to automate.

It is important that automation is responsive, flexible, and able to automate different tests quickly as needed. Although we try to plan which tests to automate and when, we should always start automating the most important tests first. Once we are running the tests,

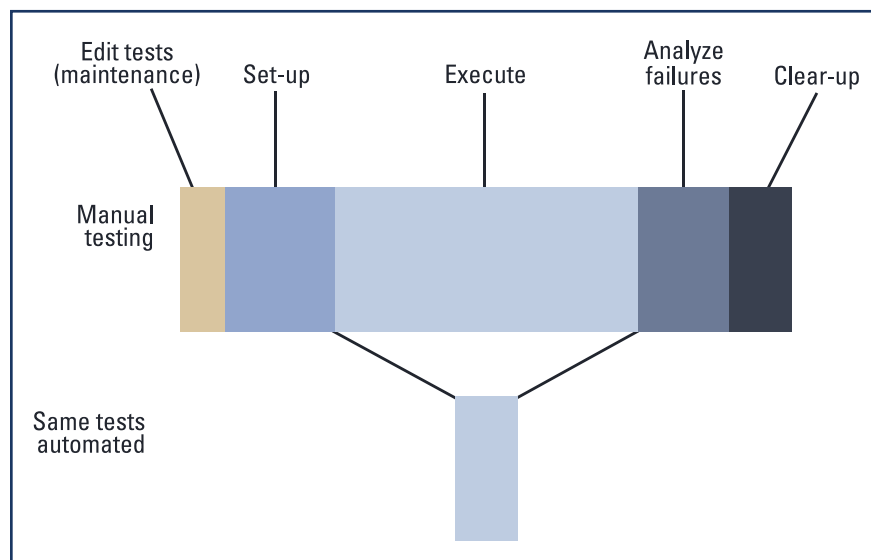


Figure 1

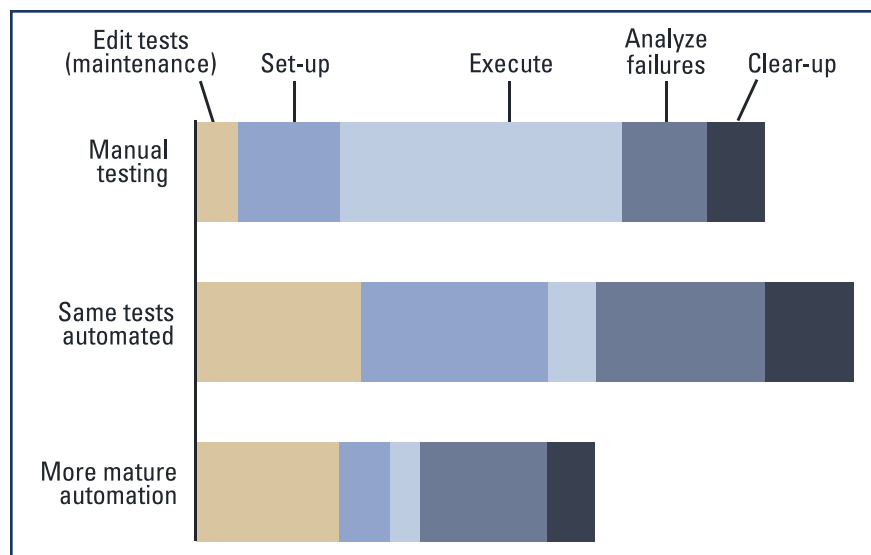


Figure 2

the testers may discover new information that shows that different tests should be automated rather than the ones that had been planned. The automation regime needs to be able to cope with a change of direction without having to start again from the beginning.

During the journey to effective test automation, it may take far longer to automate a test than to run that test manually. Hence, trying to automate may lead, in the short term at least, to running fewer tests, and this may be OK.

Better automation objective: Automate the optimum number of the most useful and valuable tests, as identified by the testers. This could be measured as the number or percentage automated out of the valuable tests identified.

OBJECTIVE 6: AUTOMATE X% OF TESTING

Good ideas behind this objective:

- We should measure the progress of our automation effort.
- We should measure the quality of our automation.

This objective is often seen as “Automate 100 percent of testing.” In this form, it looks very decisive and macho! The aim of this objective is to ensure that a significant proportion of existing manual tests is automated, but this may not be the best idea.

A more important and fundamental point is to ask about the quality of the tests that you already have, rather than how many of them should be automated. The answer might be none—let’s have better tests first! If they are poor tests that don’t do anything for you, automating them still doesn’t do anything for you (but faster!). As Dorothy Graham has often been quoted, “Automated chaos is just faster chaos.”

If the objective is to automate 50 percent of the tests, will the right 50 percent be automated? The answer to this will depend on who is making the decisions and what criteria they apply. Ideally, the decision should be made through negotiation between the testers and the automators. This negotiation should weigh the cost of automating individual tests or sets of tests, and the potential costs of maintaining the tests, against the value

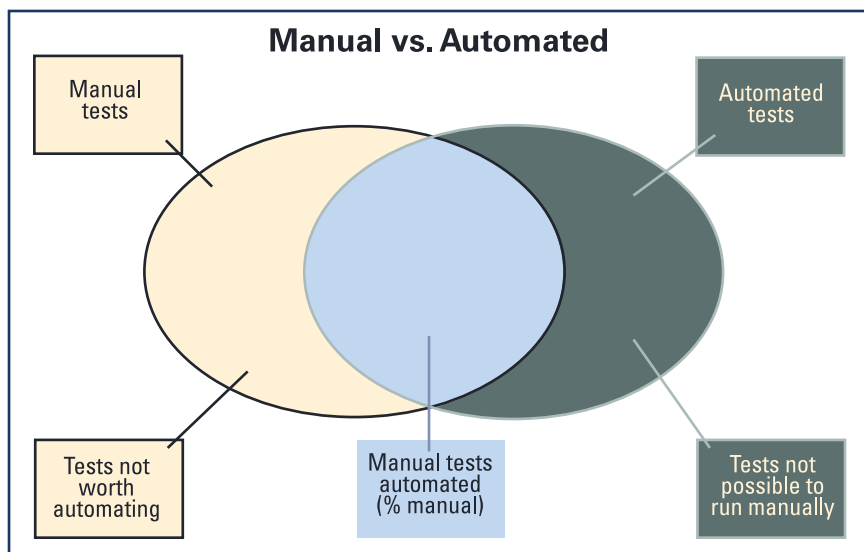


Figure 3

of automating those tests. We’ve heard of one automated test taking two weeks to build when running the test manually took only thirty minutes—and it was only run once a month. It is difficult to see how the cost of automating this test will ever be repaid!

What percentage of tests *could* be automated? First, eliminate those tests that are actually impossible or totally impractical to automate. For example, a test that consists of assessing whether the screen colors work well together is not a good candidate for automation. Automating 2 percent of your most important and often-repeated tests may give more benefit than automating 50 percent of tests that don’t provide much value.

Measuring the percentage of manual tests that have been automated also leaves out a potentially greater benefit of automation—there are tests that can be done automatically that are impossible or totally impractical to do manually. In figure 3 we see that the best automation includes tests that don’t make sense as manual tests and does not include tests that make sense only as manual tests.

Automation provides tool support for testing; it should not simply automate tests. For example, a utility could be developed by the automators to make comparing results easier for the testers. This does not automate any tests but may be a great help to the testers, save them a lot of time, and make things much easier for them. This is good automation support.

Better automation objective: Automation should provide valuable support to testing. This could be measured by how often the testers used what was provided by the automators, including automated tests run and utilities and other support. It could also be measured by how useful the testers rated the various types of support provided by the automation team. Another objective could be: *The number of additional verifications made that couldn’t be checked manually.* This could be related to the number of tests, in the form of a ratio that should be increasing.

What are your objectives for test execution automation? Are they good ones? If not, this may seriously impact the success of your automation efforts. Don’t confuse objectives for testing with objectives for automation. Choose more appropriate objectives and measure the extent to which you are achieving them, and you will be able to show how your automation efforts benefit your organization. {end}

Sticky Notes

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

- Dorothy Graham’s blog on DDP and test automation
- *Software Test Automation*



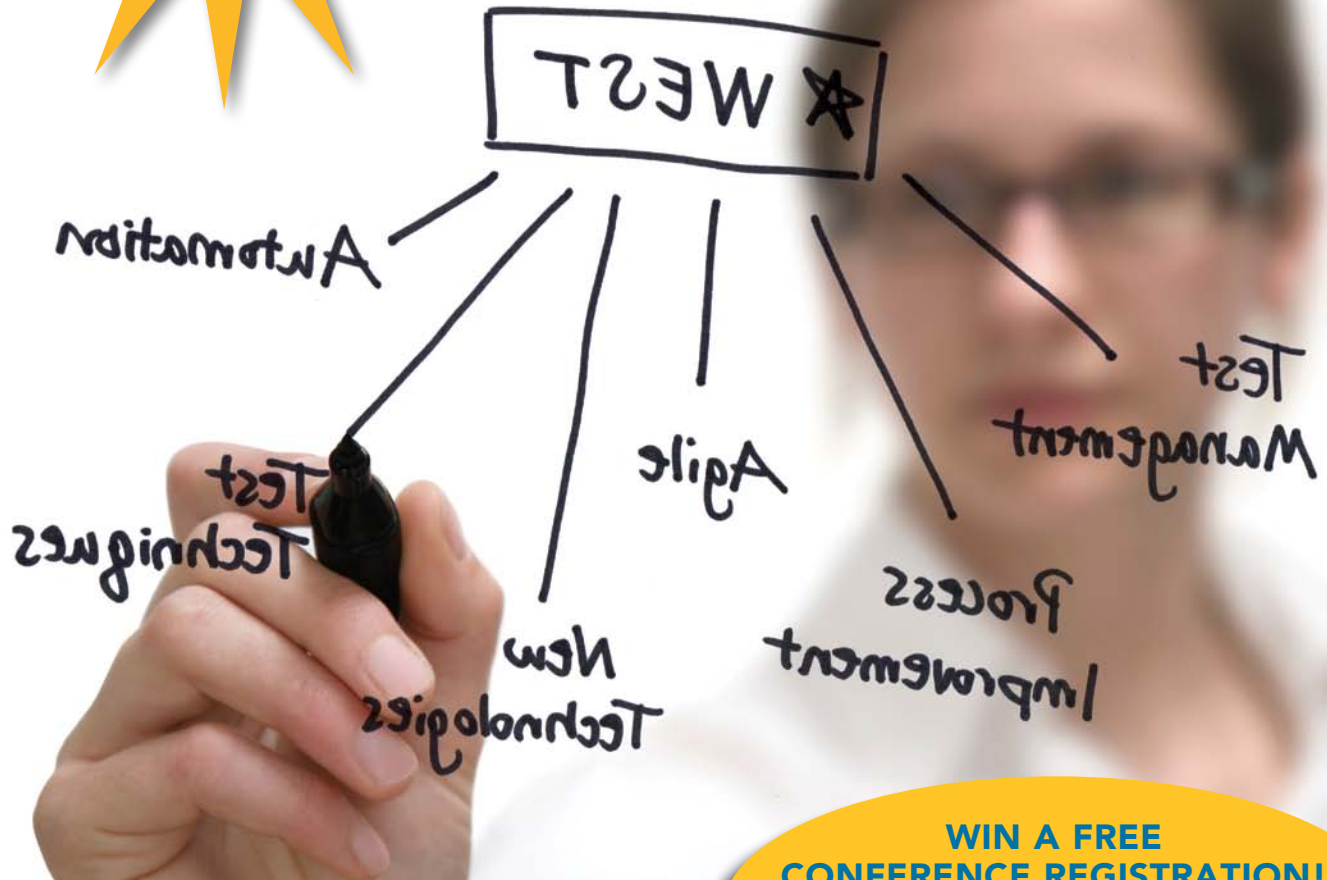
**STAR
WEST**

SOFTWARE TESTING

ANALYSIS & REVIEW

OCTOBER 5-9, 2009
ANAHEIM, CALIFORNIA
DISNEYLAND® HOTEL

The Greatest Software Testing Conference on Earth



**WIN A FREE
CONFERENCE REGISTRATION!**

Go to www.sqe.com/starwest for a chance
to win 1 of 5 free registrations
to STARWEST 2009!

You Can't Afford to Miss STARWEST 2009

- Participate in the broadest and deepest range of learning sessions available
- Cultivate relationships with peers and join the tester community
- Visit the largest testing EXPO anywhere
- Gain insight from top industry experts and speakers
- Network with peers and industry leaders

**99% OF 2008 ATTENDEES RECOMMEND
STARWEST TO OTHERS IN THE INDUSTRY**

www.sqe.com/starwest

MAXIMIZE YOUR DISCOUNTS AND SAVE UP TO \$600.
GROUPS OF 2 SAVE EVEN MORE!



New for 2009!
**Testing & Quality
Leadership Summit**



BUILD YOUR CONFERENCE!

Conference schedule includes pre-conference tutorials, keynote presentations, concurrent sessions, summit sessions, and more!



MONDAY – TUESDAY

Choose from 24 half-and full-day tutorials that allow you to learn in-depth about specific topics.



Popular Tutorials Include:

Test Automation: The Smart Way
What We Can Learn from Big Bugs that Got Away
Transition to Agile Development: A Tester's View
Risk-based Testing: Focusing Your Scarce Resources
Essential Test Management and Planning

Becoming a Trusted Advisor to Senior Management
Whittaker on Testing
Measurement and Metrics for Test Managers
Making Test Automation Work in Agile Projects
And Many More!

WEDNESDAY – THURSDAY

5 Keynote Presentations
35 Concurrent Sessions
Networking EXPO
Special Events
Bonus Sessions
...and More!



Covering topics like:

Top Testing Challenges Today
Practical Test Design Techniques
Agile Testing Practices
Free and Cheap Test Tools
Leadership in Testing
Exploratory Testing

Test Automation Frameworks
Essential Test Management
Testing Web 2.0 Applications
Test Process Improvement
ROI of Test Automation
Future Trends in Software Testing

VISIT THE EXPO!

Visit Top Industry Providers Offering the Latest in Testing Solutions

TOOLS • SERVICES • TECHNIQUES • DEMOS



KEYNOTES BY INTERNATIONAL EXPERTS



Large-scale Exploratory Testing: Let's Take a Tour

James Whittaker, Google



The Top Testing Challenges—or Opportunities—we Face Today

Lloyd Roden, Grove Consultants



The Marine Corps Principles of Leadership

Rick Craig, Software Quality Engineering



The Irrational Tester

James Lyndsay, Workroom Productions, Ltd.



Moving to an Agile Testing Environment: What Went Right, What Went Wrong

Ray Arell, Intel

"This was a fantastic conference. The trainers/facilitators were top-notch."

— Ladona Laporte, Software Process Technologies

FRIDAY

Testing & Quality Leadership Summit

New for 2009! Add a fifth day to your conference event by attending the Testing & Quality Leadership Summit Thursday evening and Friday. Join senior leaders from the industry to gain new perspectives and share ideas on today's software testing issues.



Maximizing the Value of Testing to the Business

Marc René, Director, Billing Strategy, Business Architecture and Sourcing, MetLife Auto and Home

Changing the Software Test Culture Step by Step

Tom Wissink, Information Systems and Global Services, Lockheed Martin

The ROI of Testing and Quality: A Business Executive's View

Jeffery Payne, CEO, Coveros, Inc.

The Future of Testing—How Testing and Technology Will Change

Joachim Herschmann, Director of Product Management, Borland Software



WAYS TO SAVE ON YOUR CONFERENCE REGISTRATION



Special Early Bird Offer!

Receive up to \$300 off the regular conference registration fees if payment is received on or before September 4, 2009.

Buy One Get One Half Off

Register two people at the same time and save half off the second registration. The 50% savings will be taken off the lower of the two registration amounts. To take advantage of this offer, please call the Client Support Group at 888.268.8770 or 904.278.0524 or email them at sqeinfo@sqe.com, and reference promo code BOGO.

Groups of 3 or More Save 25%

Register a group of three or more at the same time and save 25% off each registration. To take advantage of this offer, please call the Client Support Group at 888.268.8770 or 904.278.0524 or email them at sqeinfo@sqe.com, and reference promo code GRP3.

PowerPass Discount

PowerPass holders receive an additional \$100 off their registration fee.

Alumni Discount

STARWEST alumni receive up to an additional \$300 discount off their registration fee.

Certification Training + Conference + Leadership Summit

If you attend the "Software Testing Certification" Training Course, the Conference, and the Leadership Summit, you save an additional \$500.

Check out all the Ways To Save at:
<http://www.sqe.com/starwest/WaysToSave.aspx>

Please Note: We will always provide the highest possible discount and allow you to use the two largest discounts that apply to your registration.



THE EXPO October 7-8, 2009

Visit Top Industry Providers Offering the Latest in Testing Solutions

Looking for answers? Take time to explore this one-of-a-kind EXPO, designed to bring you the latest solutions in testing technologies, software, and tools. To support your software testing efforts, participate in technical presentations and demonstrations conducted throughout the EXPO. Meet one-on-one with representatives from some of today's most progressive and innovative organizations.

For Sponsor/Exhibitor news and updates, visit www.sqe.com/STARWEST.

Conference Sponsors:

Borland®
THE OPEN ALM COMPANY



Cognizant

twist™
TEST AUTOMATION

Media Sponsors:

**BETTER
SOFTWARE
MAGAZINE**

Methods & Tools

queue
tomorrow's computing today

CoDe
www.code-magazine.com

artima

**IEEE
computer
society**

StickyMinds.com

www.sqe.com/starwest



REGISTER EARLY AND SAVE UP TO \$600!

Parasoft SOA Quality Solution

MONROVIA, CA—Parasoft's latest release, Parasoft SOA Quality Solution, allows organizations to create, execute, and maintain end-to-end test suites that can be initiated from rich user interfaces, through the logic in the message layer, through the implementation component, to the database (or mainframe) and back, validating the entire business process.

Features of Parasoft SOA Quality Solution include:

- **Comprehensive, integrated quality platform for complex distributed environments**—Offers significant new capabilities to help teams achieve their quality goals in light of the complexities associated with today's heterogeneous enterprise environments.
- **End-to-end testing**—Helps teams continuously validate all critical aspects of complex transactions, which may extend through Web interfaces, backend services, ESBs, databases, and everything in between.
- **Team workflow and task management**—Establishes a sustainable workflow where quality tasks are automatically generated, assigned, and distributed to the appropriate team members, making the entire team more productive.
- **Increased visibility into enterprise service bus**—Allows visualizing and tracing the intra-process events triggered by tests, facilitating rapid diagnosis of problems directly from the test environment, and enabling teams to continuously validate whether critical events continue to satisfy functional expectations as the system evolves.
- **Next-generation performance testing**—Provides customers a centrally managed load test configuration and execution environment, which is better aligned with how teams and roles are structured within an organization. Existing end-to-end functional tests are leveraged for load testing, removing the barrier to comprehen-

sive and continuous performance monitoring.

- **Platform awareness**—Features built-in, vendor-specific awareness to TIBCO, Progress Sonic, Oracle/BEA, IBM, Software AG web-Methods, and other platforms. This reduces the learning curve for organizations defining complex tests and understanding how tests pass through heterogeneous systems.

Visit www.parasoft.com/quality for more information

Web Performance Load Tester version 3.6

DURHAM, NC—Web Performance, Inc. has released the most recent version of its industry-leading Web site performance testing application. Web Performance Load Tester version 3.6 includes a number of new features that make it even easier to use and more intuitive for its non-programmer user base.

One of the application's most appealing new features is its use of visual displays and video demonstrations that let users with zero programming experience create test cases, run load tests, and analyze the results. The new Click-to-Configure feature lets users configure dynamic text entry fields simply by importing a list of established values. New Performance Goal Analysis instantly pinpoints the exact system location causing performance problems, while its User Capacity Analysis lets users know exactly how many visitors a site can handle given its current system architecture.

Features also include expanded compatibility with today's leading Web applications including .NET and Ajax, user-level analysis tools and scenario builders, new test cases and customization tools, improved reports, advanced metrics, and usability improvements among others.

Visit webperformanceinc.com for more information.

iTKO LISA version 4.6

DALLAS, TX—iTKO has announced the general availability release of their flagship LISA product suite version 4.6. iTKO's LISA product suite has achieved

success by helping iTKO customers lower the cost of quality across their application lifecycle while improving agility, shortening release cycles, and eliminating critical development and testing constraints.

LISA Version 4.6 contains several improvements to help organizations more quickly realize the benefits of infrastructure cost savings, reduced project risk, faster time-to-market, and greater software quality:

- **Third-generation service virtualization for rapid simulation of complete test environments.** LISA enables automated modeling and simulation of inaccessible or unavailable IT resources into virtual test environments for development and testing purposes, providing realistic behaviors and stateful transactions just as a production environment would.
- **Enhanced UI testing support for new RIA/Web 2.0 application delivery models.** LISA accelerates the process of testing and ensuring quality for today's rich Internet-based applications, including detailed requirements validation for Java Swing, Flash, AJAX, Flex, and ActiveX.
- **Increased governance and policy validation support for distributed composite applications, including SOA, BPM, Cloud, and others.** LISA supports the enforcement and verification of policies and service level agreements by validating business outcomes and results using event-based triggers such as service publishing or validating continuously while in production.

LISA 4.6 also includes several usability enhancements based on customer feedback and expanded platform and standards support including Eclipse, CORBA, SAML 2.0, REST-based validation approaches, and others.

Visit www.itko.com for additional information.

10 Things You Might Not Know About

Sustainable Test-driven Development (TDD)

by Scott Bain

- 1 **A TEST SHOULD BE PROVEN TO FAIL FOR A WELL-UNDERSTOOD, NARROWLY DEFINED REASON.** If this is not proven, then it is possible that a test will never fail, and thus the behavior it defines is not guaranteed.
- 2 **A TEST SHOULD NEVER FAIL FOR A REASON OTHER THAN ONE FOR WHICH IT WAS INTENDED.** When a test fails—yet the behavior it was designed to cover is working properly—then the test misleads us as to the source of the failure. Often this is due to unexpected or undesirable coupling in the production code.
- 3 **MULTIPLE TESTS SHOULD NOT FAIL FOR THE SAME REASON.** Multiple test failures for a single cause create maintenance problems when new features are added to the system, when existing features are removed, or when there are changes for performance, security, or any other reason. Agility requires that we embrace change, and large numbers of test failures due to a single change inhibit this.
- 4 **EACH TEST SHOULD MAKE A UNIQUE BEHAVIORAL DISTINCTION IN THE SYSTEM THAT NO OTHER TEST MAKES.** This will ensure adherence to points 1, 2, and 3.
- 5 **TESTS MUST BE WRITTEN WITHIN A WELL-UNDERSTOOD SCOPE.** For example, if we are testing that an algorithm properly calculates tax, we do not need to write a test to ensure that the code being tested does not also reset the system clock, format the disk, or do anything else outside the scope of the problem. We test for errors, not malice.
- 6 **A TEST TESTS ALL THOSE THINGS THAT ARE IN SCOPE BUT ARE NOT UNDER THE TEST'S CONTROL.** Therefore, if we wish to test a single, narrowly defined thing, then everything else in scope must be brought under the test's control. We use techniques such as mocking, shunting, endo-testing, dependency injection, etc., to accomplish this. Developers must be well trained in these techniques if the test suite is to remain maintainable as the project matures.
- 7 **THE ROLE OF A CODE COVERAGE TOOL IN TDD IS TO VERIFY, AFTER A TEST IS REMOVED, THAT THE SYSTEM IS INDEED STILL FULLY COVERED.** For example, when a test is found to duplicate the distinction of another test, one of them must be removed. If code coverage reveals an uncovered path, then either the test was not actually a duplicate or there is “dead” (unused) code in the system.
- 8 **THE SPECIFICATION OF THE TESTS IS HOW WE CONDUCT ANALYSIS.** The resulting test suite, along with the scenario descriptions that it satisfies, becomes the functional specification of the system. This has advantages over traditional specification documents in that the suite is written in concrete language, which is verifiable by the computer without human intervention.
- 9 **THE LEVEL OF COMPLEXITY IN TESTING IS AN INDICATOR OF THE QUALITY OF THE DESIGN OF THE CODE BEING TESTED.** For example, when test classes are significantly larger than the classes they test, this can indicate that the tested class does too many things. Or, a test that requires a large number of instances can indicate excessive or unwanted coupling in the system. Therefore, testability is a driver of good design.
- 10 **TEST SUITES MUST RUN FAST SO THEY CAN BE RUN FREQUENTLY WITHOUT EXCESSIVE COST.** Therefore, all dependencies on the database, the GUI, the network, etc., that would make tests run slowly must be mocked for testing. These external dependencies should be tested, as well, but they can be tested by integration tests that are not run frequently and, therefore, are not expected to run as fast.

Adapting Inspections to the Twenty-first Century

by Ed Weller

The global workforce is here to stay. Project teams are spread across multiple time zones with few or no overlapping workday hours. When team members are located in Australia, China, India, Europe, and across all US time zones, it becomes unrealistic to hold the team meeting part of the inspection process. So, how can we adapt team meetings to our global reality?

Before we change a working process to meet the new environment, we need to know how and why it works in the current environment. To fully understand the problem, we need to look at why the team meeting was included in the inspection process. Why not just distribute the work product, have people review it, and gather comments?

Twentieth Century Process

The team meeting in the inspection process has evolved into two variants, often denoted as the *Fagan method* [1] and the *Gilb method*. [2] In the Fagan method, after individual preparation, the team meets to read through the work product, noting and recording defects. In the Gilb method, after individual preparation, the team meets to record defects without reading the work product. In both methods, team consensus (ideally) results in an agreed upon defect list for the author to fix, with relatively few open issues. There have been numerous analyses of the effectiveness and efficiency of the contrasting meeting methods, but my experience has been that either method works—as long as you pay attention to the fundamental requirements of the process.

However, when you consider that the number one reason inspections become ineffective is the lack of adequate individual preparation, the team meeting in the Fagan method provides a safety net. The reading of the work product slows the review rate and allows the team to find many of the defects that would have been noted in individual prepara-

tion. Additionally, those team members who consistently do not identify defects in the team meeting are subject to some amount of peer pressure to improve their performance. In either method there is a process step that calls for the moderator to postpone the inspection until sufficient preparation effort has been spent.

Adapting to the Twenty-first Century

Considering the above points, we can build a list of requirements that enable inspections across multiple time zones without the need for a face-to-face or teleconferenced team meeting.

- A. Record potential defects found by each team member
- B. Record effort spent by each team member
- C. Replace the “consensus” step in the team meeting
- D. Maintain the peer pressure that exists in the current method

There are commercially available tools as well as proprietary tools that help us accomplish these requirements. Let's look at the steps involved with using these tools to aid in inspections.

1. Record each potential defect and its location in the work product to allow for sorting and identification of duplicate defects. **Satisfies requirement: A**
2. Record the person identifying the defect and effort spent. **Satisfies requirements: B, D**
3. Note whether the defect severity is “major,” “minor,” or “issue.” (Issue is often allowed as a choice when the reviewer cannot determine the impact.) **Satisfies requirements: A**
4. Describe the defect. **Satisfies requirements: A, C**
5. Optionally, hide the defect list until all inspectors have completed their preparation, so the moderator can make all defects

visible for team review. **Satisfies requirement: D**

6. Include an “accept,” “reject,” or “duplicate” disposition by the author. **Satisfies requirement: C**
7. Insert a resolution field to describe the fix (or reason for rejecting the defect) and the reply by the defect originator if he does not agree with the rejection. **Satisfies requirement: C**
8. Provide a closed indicator. **Satisfies requirement: C**

Steps one through four are straightforward. As team members review the work product, they make entries (add records) into the tool with the location, description, and their judgment of the severity. Step five is optional; you may want to hide the list to prevent “coat tailing” (adding defects by copying others already in the log) or more positively, to ensure as many views of a potential defect as possible. The counterargument is that it is a waste of time to enter multiple descriptions of the same defect. I can see valid reasons for either approach and leave it to your organization's culture to decide which will work best for you. Hiding defects until all team members have completed review is one way of keeping peer pressure in place.

Steps six and seven start when all reviewers have completed their preparation. The author reviews the defect list; marks defects accept, reject, or duplicate (the location field is critical for this step!); and enters a short description of the resolution. “Short” could mean as little as “OK” or “fixed,” or in some cases, a description of a complex fix that allows the team to review the author's solution to the defect.

After completing steps six and seven for all defects, the tool notifies the moderator to check for any “rejects.” If there are none, the team has reached consensus (Satisfies requirement: C),

and the updated work product enters the normal acceptance or verification step in the inspection process, using the closed field (step eight) to indicate the defect is closed. If the author rejects any defects, the team is asked to review and agree or disagree with the author. This can be accomplished by adding notes to the resolution field. If the team members agree with the author's reasoning, the rework can be accepted. If there is still disagreement, at this point I'd recommend a teleconference to quickly resolve the problem—talking just works better than typing.

Advantages and Disadvantages

This may seem to be a lot of work—recording defects, recording responses, and logging effort. My reply would be that yes, there is effort spent in all these activities, but the steps are essential to the correct and effective execution of the inspection process in a globally distributed environment. In any collaborative review process, the reviewers' comments need to get back to the author. Do we do this with handwritten scrawls on paper, comments in an electronic version of the paper, or in a reusable form in a database that is visible to all reviewers? I've used the latter process on multiple

occasions and have found that the time it takes to record the data is a small fraction of the actual review time. If we look at the development process from end to end, rather than sub-optimizing one small step in the process, the leverage provided by early defect identification and removal clearly offsets the cost of recording this data.

As a side benefit of eliminating the face-to-face meeting, we solve one of the thorny issues with the inspection process—the delays caused by scheduling three or more people into a face-to-face meeting. It also allows a larger number of reviewers to contribute to a review without making the face-to-face meeting unmanageable. Note that the metrics used to evaluate preparation rate may need to change if you have a large number of reviewers looking at parts of the work product.

By understanding the dynamics of the inspection process, we can identify a set of critical requirements that must be met to enable inspection teams to operate in widely divergent time zones and maintain the core principles of the inspection process. We have enabled multiple people to participate in a “virtual meeting” without having to meet face to face. By eliminating the face-to-face or teleconference team meeting, team

members can review and comment on a work product during their normal work hours. An essential part of the process—peer pressure to prepare properly—can be maintained using a defect log. **{end}**

REFERENCES

- [1] Radice, Ronald A. *High Quality Low Cost Software Inspections*. Paradoxicon Publishing, 2001.
- [2] Gilb, Tom and Dorothy Graham. *Software Inspection*. Addison-Wesley Professional, 1994.

HAVE THE LAST WORD!

If you have a point to make or a side to take on issues and trends that affect the industry, we want to hear from you.

We are looking for insightful, thought-provoking commentary for possible use as **The Last Word**.

Please send an abstract to
editors@bettersoftware.com.

Index to Advertisers

Agile Development Practices	www.sqe.com/ADP	5
ArtOfTest	www.ArtOfTest.com	8
Cognizant	www.cognizant.com	2
Hewlett-Packard	www.hp.com/go/software	Back Cover
Rally Software	www.rallydev.com/bsm	Inside Front Cover
Seapine	www.seapine.com	1
SQE Training—Agile	www.sqetraining.com/Agile	25
SQE Training—eLearning	www.sqetraining.com/eLearning	19
SQE Training—Requirements	www.sqetraining.com/Engineering	13
STARWEST 2009	www.sqe.com/STARWEST	37–40
StickyMinds.com Blogs	blogs.stickyminds.com	12
TechExcel	www.techexcel.com	Inside Back Cover

Display Advertising advertisingsales@sqe.com

All Other Inquiries info@bettersoftware.com

Better Software (USPS: 019-578, ISSN: 1553-1929) is published seven times per year January/February, March, April, May/June, July/August, September/October, November/December. Subscription rate is US \$39.99 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. Entire contents © 2009 by Software Quality Engineering (330 Corporate Way, Suite 300, Orange Park, FL 32073), unless otherwise noted on specific articles. The opinions expressed within the articles and contents herein do not necessarily express those of the publisher (Software Quality Engineering). All rights reserved. No material in this publication may be reproduced in any form without permission. Reprints of individual articles available. Call for details. Periodicals Postage paid in Orange Park, FL, and other mailing offices. POSTMASTER: Send address changes to Better Software, 330 Corporate Way, Suite 300, Orange Park, FL 32073, info@bettersoftware.com.

Wiki-powered

Get everyone on board



DevSpec

Wiki-powered requirements management

Implementing a requirements management solution can be challenging if it's not intuitive and easy to use. Experience the DevSpec difference:

Easy

- DevSpec's Wiki interface makes adoption easy

Powerful

- Fully-definable graphic workflow
- Completely customizable user interface

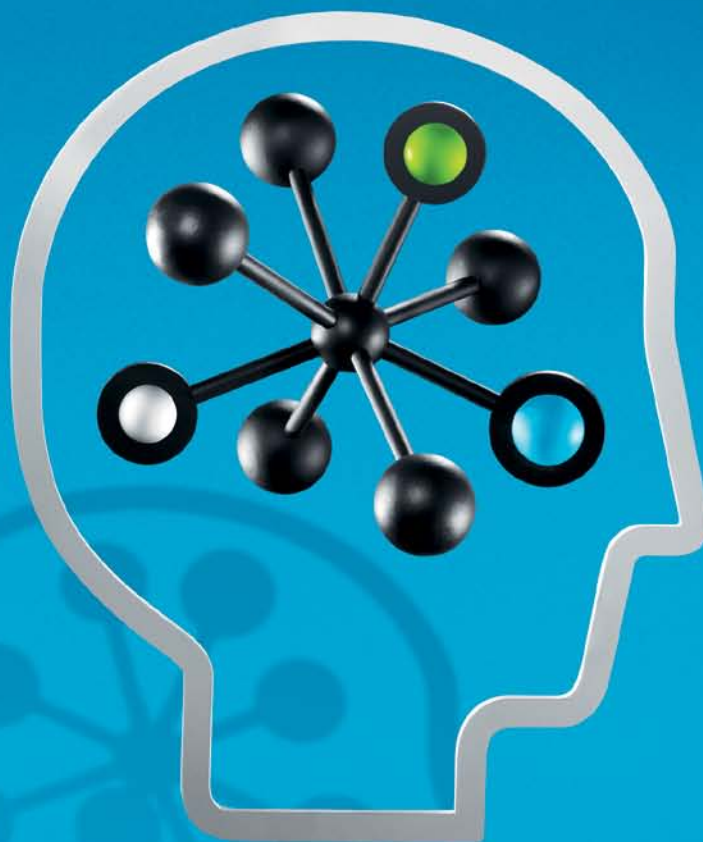
Full Traceability

- Integrated with DevTrack for requirement-driven development
- Integrated with DevTest for requirement-driven testing
- Integrated with Community so your customers are part of your requirements team

TechExcel www.techexcel.com



© 2009 TechExcel, Inc., All rights reserved.



ALTERNATIVE THINKING ABOUT APPLICATION LIFECYCLE MANAGEMENT:

Computers Don't Run Your Apps. People Do.

Alternative thinking is looking beyond the development cycle and focusing on customer satisfaction. Because the real application lifecycle involves real people – and the customer's perception is all that matters in the end.

HP helps you see the big picture and manage the application lifecycle. From the moment it starts – from a business goal, to requirements, to development and quality management – (and here is the difference) – all the way through to operations where the application touches your customers.

HP ALM offerings help you ensure that your applications not only function properly, but perform under heavy load and are secure from hackers. (Can't you just hear your customers cheer now?)

Technology for better business outcomes. hp.com/go/alm

