

# 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.

Avoid cutting heads with the only  
Application Lifecycle Management provider  
to ensure your Agile success.

**Learn More about our Guarantee Success Program  
at [rallydev.com](http://rallydev.com)**



Scaling Software Agility

© 2009 Rally Software Development Corp

# BETTER SOFTWARE

**ANALYSIS I.O.U.**  
Tips for responsible  
borrowing

**WALK THE TALK**  
Mapping effective  
user stories

The Print Companion to **StickyMinds.com**

## COUNTDOWN TO AGILITY

**TOP  
TEN** 10

CHARACTERISTICS  
of an **AGILE**  
ORGANIZATION



**2009 SALARY SURVEY  
RESULTS INSIDE!**



# 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.

Avoid cutting heads with the only  
Application Lifecycle Management provider  
to ensure your Agile success.

**Learn More about our Guarantee Success Program  
at [rallydev.com](http://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**



# Managing Mobile Application Quality for a 24 X 7 Enterprise



As the mobile user base grows exponentially and mobile devices become “anywhere, always-on” Web-enabled personal gadgets, product development challenges for device manufacturers and application developers increase by the day.

Our dedicated mobile testing lab is working tirelessly with device makers and software developers to achieve and then exceed desired product quality levels. By testing the functional and non-functional requirements encompassing usability, security and adaptability, we increase the performance of devices and operating systems that run mobile applications, augmenting revenue and profit potential for the entire value chain.

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](mailto:inquiry@cognizant.com)



## Cover Story

### COUNTDOWN TO AGILITY 18

Jean Tabaka believes in the power of an entire agile organization. These ten characteristics of an agile organization may seem counter to market success, but she explores why they are wholly embedded in twenty-first century business success. *by Jean Tabaka*

## Features

ONLY  
In the  
Digital  
Edition



## Columns & Departments

### In Every Issue

Contributors 7

Editor's Note 8

Product Announcements 37

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](http://www.BetterSoftware.com)  
or call 800.450.7854.

### TELLING BETTER USER STORIES 24

While the idea of a user story is simple on the surface, there are challenges to working with them. User story mapping is a useful way to organize, decompose, and prioritize user stories. *by Jeff Patton*

### PREPARE TO SUCCEED 30

Another pair of eyes will find bugs, but code reviews are traditionally time consuming and painful. Learn how modern, lightweight techniques make code reviews effective and practical. *by Jason Cohen*

### TECHNICALLY SPEAKING 11

**Outside the Strike Zone** • *by Lee Copeland*

In a counterpoint to his previous Technically Speaking column, Lee explains why holding fast to one's beliefs is not necessarily a bad thing.

### INSIDE ANALYSIS 12

**Managing Your Analysis Debt** • *by Mary Gorman and Ellen Gottesdiener*

What is your project's analysis debt load? What's the difference between good and bad analysis debt? What are causes and remedies for such debt? Mary Gorman and Ellen Gottesdiener explore the concept of analysis debt and consider strategies for prudent investing.

### TEST CONNECTION 14

**Constructing the Quality Story** • *by Michael Bolton*

Knowledge doesn't just exist; we build it. Sometimes we disagree on what we've got, and sometimes we disagree on how to get it. Hard as it may be to imagine, the experimental approach itself was once controversial. What can we learn from the disputes of the past? How do we manage skepticism and trust and tell the testing story?

### MANAGEMENT CHRONICLES 16

**Par for the Course** • *by Patrick M. Bailey*

What can happen over a game of golf? You learn what you don't know, you learn more about what you do know, and you learn to listen to what others know. See how two managers and a caddy team up for some valuable lessons about staying out of the rough.

### THE LAST WORD 43

**QA Is Not Evil** • *by Chris McMahon*

A software tester re-examines the role of software testers in quality assurance work, helping implement software development processes. If testers are knowledgeable, helpful, and supportive, they may be in the best possible position to help the team improve its software development process.



### TRAINING WEEKS

[www.sqetraining.com/public](http://www.sqetraining.com/public)

#### Testing

**November 16–20, 2009**

Tampa, FL

**March 22–26, 2010**

San Diego, CA

**April 12–16, 2010**

Boston, MA

### SOFTWARE TESTER

#### CERTIFICATION

[www.sqetraining.com/certification](http://www.sqetraining.com/certification)

**November 16–18, 2009**

Tampa, FL

**December 1–3, 2009**

Phoenix, AZ

**February 16–18, 2010**

Toronto, ON

**February 23–25, 2010**

Mountain View, CA

Atlanta, GA



### CONFERENCES

#### STAREAST 2010

##### Software Testing Analysis & Review

[www.sqe.com/stareast](http://www.sqe.com/stareast)

**April 25–30, 2010**

Rosen Shingle Creek  
Orlando, FL

#### Better Software Conference

[www.sqe.com/bsc](http://www.sqe.com/bsc)

**June 6–11, 2010**

Caesars Palace  
Las Vegas, NV

#### Agile Development Practices | West

[www.sqe.com/adpwest](http://www.sqe.com/adpwest)

**June 6–11, 2010**

Caesars Palace  
Las Vegas, NV

#### STARWEST 2010

##### Software Testing Analysis & Review

[www.sqe.com/starwest](http://www.sqe.com/starwest)

**September 26 –  
October 1, 2010**

Hilton San Diego Bayfront  
San Diego, CA

#### Agile Development Practices | East

[www.sqe.com/adpeast](http://www.sqe.com/adpeast)

**November 14–19, 2010**

The Rosen Centre  
Orlando, FL

# 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](mailto:editors@bettersoftware.com)

Subscriber Services: [info@better-software.com](mailto: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



**Take quality to the next level**



## **DevTest Studio**

The integrated solution for defect tracking, test management and automated testing

### **DevTrack**

Use DevTrack to track defects/issues

- Track each issue through a definable workflow
- SCM integration – track fixes against their source code deliverables
- Deploy a resolution across multiple releases, versions and products
- Reporting and metrics to illustrate the entire defect lifecycle

### **DevTest**

Use DevTest to manage your testing

- Create a central repository for your test cases, knowledge items and automation scripts
- Schedule releases and test cycles using a wizard-driven interface
- Execute test assignments and submit defects from the same interface
- Track results with real-time dashboards and reports

### **TestLink**

Use TestLink to automate your testing

- Add automated tests to the DevTest test library
- Schedule automated tests along with manual tests
- Launch automated tests from the DevTest interface
- Track automation results with real-time dashboards and reports

**TechExcel**

[www.techexcel.com](http://www.techexcel.com) | 1-800-439-7782



# SOFTWARE TESTING ANALYSIS & REVIEW

# 2010



THE GREATEST SOFTWARE TESTING CONFERENCE IN THE UNIVERSE (WE THINK!)

APRIL 25-30, 2010  
ORLANDO, FLORIDA  
ROSEN SHINGLE CREEK



**MAXIMIZE YOUR DISCOUNTS  
AND SAVE UP TO \$400.  
GROUPS OF 2 SAVE EVEN MORE!**

[www.sqe.com/stareast](http://www.sqe.com/stareast)

## Choose from a full week of learning

### SUNDAY

Software Testing Certification Training (3-days)

### MONDAY-TUESDAY

26 Tutorials, STF Training continues

### WEDNESDAY-THURSDAY

5 Keynotes, 35 Concurrent Sessions, Networking EXPO, Bonus Sessions, EXPO Reception and more

### FRIDAY

Testing & Quality Leadership Summit





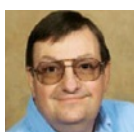
**PATRICK M. BAILEY** teaches information systems at Calvin College in Grand Rapids, MI. He has more than twenty years of experience as a software developer and manager with his experience evenly divided between new development and maintenance organizations. Patrick regularly speaks about IT leadership subjects at non-profit-oriented conferences. His first paying job was as a caddy when he was twelve. Email Patrick at [pmb4@calvin.edu](mailto:pmb4@calvin.edu).



**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](mailto:mb@developsense.com).



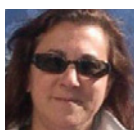
**JASON COHEN** is the founder of Smart Bear Software, maker of the Code Collaborator tool for peer code review. In conjunction with Cisco Systems, Jason conducted the world's largest study of lightweight peer code review and published the findings in his book, *Best Kept Secrets of Peer Code Review*.



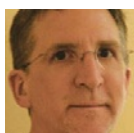
**LEE COPELAND** has more than thirty years of experience in the field of software development and testing. He has worked as a programmer, development director, process improvement leader, and consultant. Based on his experience, Lee has developed and taught a number of training courses focusing on software 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](mailto:lcopeland@sqe.com).



**MARY GORMAN**, CBAP<sup>TM</sup>, is senior associate at EBG Consulting and helps project teams explore, analyze, and build robust business and system requirements models. Mary serves on the Business Analysis Body of Knowledge committee of the International Institute of Business Analysis and is the leader of the elicitation subcommittee. She can be reached at [mary@ebgconsulting.com](mailto:mary@ebgconsulting.com) and [ebgconsulting.com](http://ebgconsulting.com).



**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.



**CHRIS MCMAHON** is an experienced software tester. He lives in a small town deep in the Four Corners area of southwest Colorado. He is a dedicated telecommuter committed to building and growing high-performing distributed agile software development teams. Email him at [christopher.mcmahon@gmail.com](mailto:christopher.mcmahon@gmail.com).



**JEFF PATTON** has designed and built software for the past fifteen years. He currently works as an independent consultant, agile process and product design process coach, and instructor. Jeff is founder and list moderator of the agile-usability Yahoo! discussion group, a columnist with StickyMinds.com and IEEE Software, a Certified Scrum Trainer, and winner of the Agile Alliance's 2007 Gordon Pask award for contributions to agile development.



**JEFFERY PAYNE** is CEO and founder of Coveros, Inc. Prior to Coveros, he was chairman of the board, CEO, and co-founder of Cigital, Inc. Jeffery has been a keynote and featured speaker at CIO and business technology conferences and frequently testifies before Congress on issues of national importance, including intellectual property rights, cyber-terrorism, and software quality. You can reach Jeffery at [jeff.payne@coveros.com](mailto:jeff.payne@coveros.com).



An agile coach with Rally Software, **JEAN TABAKA** specializes in creating and mentoring agile software teams. Bringing more than twenty-five years of experience in software development to the agile plate, Jean is a Certified ScrumMaster, Certified Scrum Trainer, Certified Professional Facilitator, and author of *Collaboration Explained: Facilitation Skills for Software Project Leaders*.





### AGILITY: IT'S NOT JUST FOR SOFTWARE

Fresh off my trip to Anaheim for our STARWEST 2009 conference, I find myself reflecting on the week with a mixture of pride and appreciation. We have a fairly small staff at Software Quality Engineering, so it always amazes me to see the way the company pulls together to produce such a large event.

Everyone from the CEO to the newly hired former intern happily jumps into the fray and pitches in. It's not unusual to see VPs running microphones, editors directing lunch traffic, and managers passing out morning announcements. This level of cooperation and dedication is indicative of an organization with a strong mission and core set of values. And, based on what I've learned from this month's cover story, it's also very agile.

In "Countdown to Agility," Rally's Jean Tabaka reveals the core values and top ten characteristics that she feels are essential to building a great agile organization. Included in this list are ideas to help you deliver your product consistently and still maintain your staff's work/life balance, hire managers who act as both servant and leader, and contribute to the community while remaining profitable.

Also in this issue, find out how to implement user story mapping as a way to organize, decompose, and prioritize user stories, as well as keep the conversation about your product alive throughout its construction.

Don't miss our exclusive digital edition content. The 2009 *Better Software* magazine/StickyMinds.com Salary Survey results will only be published in the digital edition this year, so make sure to check your inbox.

As always, I hope you enjoy this issue of *Better Software* magazine. If you are attending the Agile Development Practices 2009 conference in Orlando, stop by the Interactive Lounge and tell me how you've put our magazine to work for you.

Happy Reading!

Heather Shanholtzer  
HShanholtzer@sqa.com

# Implementing HP Quality Center Software

## A Customer Perspective

### “Best Practices for Quality Center”

*On-demand presentation now available*

Putting our  
imprint on  
technology  
since 1921

Time is of the essence, resources are few, deadlines loom, and leveraging what works becomes critical. Take a short break and join HP Software and Avnet on a customer journey. Learn about the “best practices” around HP Quality Center Software and how implementing new testing tools or process starts with thorough thought and planning. We’ve learned that sometimes the best tools alone aren’t enough; well-defined processes aren’t enough; and insightful plans aren’t enough. So, whether you’re an HP customer or a perspective customer ~ join us for this informative webinar and ensure your requirements, questions, and concerns are understood and addressed from a lifecycle approach to quality.

**[hpsoftware@avnet.com](mailto:hpsoftware@avnet.com)**





New  
for 2010

## Two Co-located Conferences

*Attend one conference and receive full access to both events*

# BETTER SOFTWARE CONFERENCE



## Mark Your Calendar Now!

**June 6-11, 2010**

*Las Vegas, Nevada*

*Caesars Palace*

[www.sqe.com/adpwest](http://www.sqe.com/adpwest)

[www.sqe.com/bsc](http://www.sqe.com/bsc)



# Outside the Strike Zone

by Lee Copeland

Ted Williams was the greatest hitter the game of baseball has ever known. Williams knew exactly where the strike zone was and steadfastly refused to swing at pitches that were even a little bit outside. “A great hitter should walk three times for every strikeout,” Williams said. “Otherwise, he’s swinging too often at the pitcher’s pitch.” He believed that his refusal to swing at bad pitches gave him more opportunities to swing at pitches he could hit, and, in hitting those pitches, he benefited himself and his team.

But, as great as Williams was, baseball writers, and even some of his own teammates, criticized Williams for his approach to hitting. They said he would have hit even better and helped his team more if he had swung every once in a while at pitches just outside the strike zone. Williams responded to this criticism in his book, *The Science of Hitting*. “As soon as you start going for the pitch that’s one or two inches off the plate, you’ve automatically widened the strike zone for the pitcher. Before you know it, you’re swinging at pitches three or four inches off. Where does it end?” And that is the vital question—where does it end?

The word “orthodox,” from Greek *orthodoxos* (“having the right opinion,”) from *orthos* (“right, true, straight”) + *doxa* (“opinion, praise,” related to *dokein*, “to think”), means adhering to the accepted, traditional, or established principles, especially in religion and software development. The concept of and need for orthodoxy is pervasive in nearly all forms of organized monotheism, although orthodox belief is generally not emphasized in polytheistic religions. The same can be said about practitioners of software development who can be divided into two groups: the “one-true-way” school and the “context-sensitive” school. The one-true-way school is heavily invested in the need for orthodoxy.

The subjects of orthodoxy are couched in commandments, manifestos,

creeds, writings, and traditions. In “Fiddler on the Roof,” Tevye reminds us that “because of our traditions, every one of us knows who he is and what God expects him to do. Without our traditions, our lives would be as shaky as a fiddler on the roof!” Tevye’s traditions define roles and responsibilities. They provide him an anchor in the swirling winds of change.

In exactly the same way, the Agile Manifesto and the writings defining Scrum provide an anchor. They define our roles and responsibilities. They tell us what to do and what not to do. They promise to guide us to repeatable success.

In my last column, I criticized the Scrum process for its rigid requirement that “every Scrum sprint must finish by delivering new product functionality that is integrated, fully tested, and potentially shippable. [1] This requirement eliminates the possibility of having special-purpose sprints for stabilization, hardening, integration testing, regulatory compliance testing, etc.—sprints that could be very useful for enhancing product quality. Previously, I wrote “Scrum seems to have become less agile in its emphasis on following the rules, regardless of development context or value achieved.”

The adherents to Scrum are simply holding fast to their orthodox beliefs, parrying aside attempts to “liberalize” the rules. They have defined Scrum’s “strike zone” and are holding to it. Like Ted Williams, they are concerned about

“The adherents to Scrum are simply holding fast to their orthodox beliefs, parrying aside attempts to “liberalize” the rules. They have defined Scrum’s “strike zone” and are holding to it.”

where it ends if they wander outside their boundaries. And for that, they are to be commended.

However, there is peril in orthodoxy. Unwavering allegiance to dogma and authority can lead us astray if the dogma is not beneficial or the authority is mistaken. Recent history is replete with examples of failed dogma—from business models to investment strategies to war-fighting tactics to energy policies.

The context-driven school of software testing offers some excellent guidance, applicable to all facets of software development—“the value of any practice depends on its context; there are good practices in context, but there are no *best* practices; projects unfold over time in ways that are often not predictable, and only through judgment and skill—exercised cooperatively throughout the entire project—are we able to do the

right things at the right times.” [2] These ideas embody the spirit of agile development: “valuing interactions over process, responding to change over following a plan.” These ideas also see “agile” as an adjective, not a noun. {end}

## Sticky Notes

For more on the following topic go to [www.StickyMinds.com/bettersoftware](http://www.StickyMinds.com/bettersoftware).

■ References

Read more articles  
by Lee Copeland on  
[StickyMinds.com](http://StickyMinds.com)

# Managing Your Analysis Debt

by Mary Gorman and Ellen Gottesdiener

We recently heard about an agile project that implemented four user stories. The first story began, “As a sales associate ...”; The second story began, “As a sales rep ...”; and the third and fourth stories began, “As a sales consultant ...” These stories and the resulting software successfully delivered the customer’s understanding.

Now, flash forward three iterations. The customer has had an epiphany: These three users are really only one—the sales rep. So the stories, rules, data, and interface experiences are overlapping and conflicting, and they need to be refactored. This change cost the team almost two iterations of work.

## Technical Debt in Software Projects

Ward Cunningham coined the metaphor of technical debt in 1992. “Shipping first-time code is like going into debt,” he said. “A little debt speeds development so long as it is paid back promptly with a rewrite. The danger occurs when the debt is not repaid.” [1]

For large software projects, using debt is often a wise financial strategy. But incurring debt is always a risk, especially if it is high-interest debt and you’re not paying close attention to the cost. The same is true of technical debt, and it applies not only to code but also to architectural design [2] and even to requirements analysis.

## Analysis Debt

Analysis debt results when a team:

- Defines and implements requirements in a manner that limits or disables future integration of various customer views, extensibility, scalability, or reuse
- Uses requirements practices that make it harder to make changes later, require extra effort to clean up, and generally cost more
- Invests too much in analysis too early



ISTOCKPHOTO

Analysis debt can be intentional or unintentional. For example, customers may *intentionally* limit requirements for the next release, which can lead to rework in subsequent releases to “fix” the resulting product. On the other hand, a domain subject matter expert may *unintentionally* provide inadequate product details or make poor choices of what requirements to deliver.

No development method is immune to analysis debt. On traditional (waterfall) projects, teams can overinvest in analysis of unnecessary requirements. Later, the team finds itself reprioritizing and removing requirements.

On agile projects, teams are addicted to delivery (in a good way!). Producing working, tested software of value as soon as possible pays off for the business. Yet, it can have the unintended consequence of incurring analysis debt.

## Good Debts and Bad

Like cholesterol, analysis debt comes in good and bad versions. Some examples of good analysis debt are:

- Delaying detailed requirements analysis of low-priority requirements
- Delaying scheduling and analyzing volatile requirements, saving wasted time and effort in the face of uncertainty
- Pretending that user requirements dependencies don’t exist, so that you deliver business-crucial features while allowing the delivery team time to learn how to deliver efficiently

Recognizing the causes of bad analysis debt can help you prevent or, at least, actively manage it. Table 1 shows common forms of analysis debt we have observed, along with suggested remedies.

## Responsible Analysis Debt

The stakeholders of analysis debt—the business customer, the delivery team, and the senior managers who advise the customer and team—must collaboratively decide when, why, and how to incur analysis debt. Everyone should be



Analysis Debt Causes	Analysis Debt Remedies
<b>Scope Definition</b>	
Poor scope definition of key domain objects, resulting in excess rework of the application architecture	Define a shared business vision of the product.
Conflicting, localized views of the business domain	Build organic analysis models (personas, domain models, business policies) to help all stakeholders understand the product and communicate clearly.
Knowledge gaps among multiple subject matter experts who own isolated portions of the product backlog	Collaborate on a shared product backlog with a single, overarching (“uber”) customer or product owner.
Focus on implementing software, ignoring the business activities needed to support released product (the operational workflow)	Evaluate the entire value stream, including how work is done, as a part of product roadmapping.
All requirements considered must-haves	Define users or personas and rank their market value. Prioritize backlog items based on market research and product positioning. Frequently reprioritize backlog.
Inconsistent use of business terms and business rules	Define standard business terms in a shareable glossary. Localize terms as needed. Identify reusable business rules.
<b>Team Flow</b>	
Stalled development due to unanalyzed or vague backlog items	Plan work-ahead analysis of next small set of high-priority requirements. Build acceptance tests and start that process before iteration planning.
Stalled development due to changed business needs that are inconsistent with completed analysis	Define requirements details as close to development time as possible.
Backlog requirements too large to deliver in an iteration or a reasonable timeframe to get return on investment	Slice requirements into minimally marketable features for a release; then right-size them for incremental delivery across iterations.
Stalled project due to analysis paralysis	Build acceptance tests, timebox analysis on each story, force-rank requirements, and limit the number of backlog items you allow to be worked on at any time.
User experience models not in sync with data and business rules analysis	Build prototypes iteratively and factor in cross-cutting users, data, and rules.
<b>Financial</b>	
Business customer surprised by the need to pay off analysis debt	Make analysis debt visible. Add backlog items to address some of the debt in the form of work-ahead analysis, product roadmapping, and analysis spikes.

Table 1: Analysis debt causes and remedies

cognizant of the factors that justify the cost of analysis debt for your product and market—for example, earning short-term revenue, delivering differentiating features to a swiftly moving market, or being first to market.

In some cases, it may be prudent to incur the cost of analysis debt if the realized income outweighs the risks and cost of future nimbleness. Be sure to incorporate the expected life of the product into your analysis-debt assessment. Most important, be sure that the consequences of deferring the cost of repaying analysis debt (rework, repair, and waste) are transparent to everyone.

The final accounting is clear: Analysis debt will compound the longer you go without paying it off. Be a prudent in-

vestor! Implement strategies to protect against unintentional analysis debt, and carefully and actively manage your intentionally incurred analysis debt. {end}

#### How do you detect and mitigate analysis debt on your projects?

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

#### Sticky Notes

For more on the following topics go to [www.StickyMinds.com/bettersoftware](http://www.StickyMinds.com/bettersoftware).

- References
- Recommended links

# Let your voice be heard!



Take the  
**Better Software**  
magazine reader  
survey today.

[WWW.SQE.COM/GO?BSM09SURVEY](http://WWW.SQE.COM/GO?BSM09SURVEY)

## BETTER SOFTWARE MAGAZINE

# Constructing the Quality Story

by Michael Bolton

What we know about a product's quality isn't inherent in the product; our knowledge is constructed by us. In today's world, we often construct knowledge by means of experiments that we call "tests"; yet, at one point in history, the experimental approach was both new and controversial. That controversy is outlined in a book, *Leviathan and the Air-Pump: Hobbes, Boyle, and the Experimental Life* by Steven Shapin and Simon Schaffer.

The story begins in 1659, when Robert Boyle and his colleague Robert Hooke finished building a version of the most sophisticated and complex scientific instrument of its time, the air pump. While trying to learn about air, Boyle realized something that we now regard as commonplace: If you want to understand how a system is affected by something, get rid of that thing. In 1660, at roughly the same time as the Royal Society was established, Boyle published the results of several experiments in which he removed the air from the "receiver"—a chamber in the air pump—and observed the effects on animals and objects that he had placed inside. More significantly, his writings discussed how a community could arrive at a *matter of fact*—something upon which everyone could agree without dispute. Boyle proposed three main points. First, the advance of scientific knowledge would depend upon instruments that would simultaneously extend human observation and remove human subjectivity. Second, experiments should be performed in front of groups of people who could observe the proceedings and the experimenter, and bear witness to accounts of what had happened. Third, a style of recording and writing should be adopted so that anyone else (with sufficient skill and funding, presumably) could reproduce the experiment.

Thomas Hobbes, a prestigious natural and political philosopher of the day, had serious objections to the ex-



perimental approach. He didn't object to experiments per se, but at the same time, he didn't believe that they presented conclusive evidence of anything. For one thing, air pumps were very unreliable. Although Hooke and Boyle spent years trying to improve theirs, it leaked persistently. Hobbes also dismissed the idea that an experiment proved anything definitively—other than the fact that the air pump didn't work well. That was easy to demonstrate because of the leaks and the fact that different experimenters obtained different results. Hobbes also believed that the air pump couldn't work to produce universal knowledge. Even in the event that the air pump did work here and now, how do you *know* that it will work somewhere else later? You can't, unless or until you actually try it, said Hobbes, so the "knowledge" that you create isn't forever and always. Experimental knowledge is always provisional.

Hobbes also pointed out that the air pump experiments and the theories that they were intended to prove had a kind of circularity to them. The air pump had its effects because Boyle's theories were true, and Boyle's theories were true because the air pump proved them. Hobbes may not have been the first philosopher to notice this loop in science (which, at the time, wasn't yet called science), but he was vocal about it, and he certainly

wasn't the last to raise the problem.

Hobbes's principal issue was that people would not agree on the evidence of experiments if their interests were at odds. His own objection alone was proof of that. Hobbes believed that true knowledge should be based upon axioms and reasoned analysis that is derived from them—the foundation of his philosophy was geometry. He had alternative and (at the time) plausible theories to explain Boyle's observations. Hobbes had a stake in maintaining his beliefs and wasn't going to give them up based on the seventeenth-century equivalent of a product demo.

It didn't help Hobbes's argument that his attacks were exceptionally nasty and personal. Boyle had widespread support from his colleagues in the Royal Society, and both Hobbes and his point of view were marginalized. The experimental approach, though imperfect, proved sufficiently useful to help generate new explanations for the way the world works, and the objections of Hobbes and several other critics were largely forgotten by history. Yet, Hobbes should be credited for sharpening the scientific method. Boyle and his colleagues were compelled to acknowledge and deal with Hobbes's criticisms, which led to refinement of the equipment on the one hand, and more explicit and more guarded truth claims about experiments on the other.

All of this occurred 350 years ago. What can we learn from it? A surprising amount, I think.

When we're testing, we're constructing knowledge. That knowledge takes the form of two parallel stories. There's a story about the product—what it is, what it does, how it does it, how it works, and how it might fail. The second story—the testing story—is about how we arrived at the product story. The testing story has a structure based on what we decided to test, the oracles that we used, the extent to which we covered our models of the product, and the techniques we applied. Building that structure is the process of test design. The testing story also has a narrative in which we describe how we configured, operated, observed, and evaluated the product; that's the process of test execution. Testing is a process of composing, editing, narrating, and justifying those stories. Our tests, our product demos at the end of a development cycle, and our careful accounts of the most interesting tests—whether delivered in conversation or in writing—can be traced right back to the staging of experiments, witnessing, testimony, and reporting that were part of Boyle's protocol.

Boyle's insight of removing something to understand its effects remains an important testing technique. Want to find interesting bugs? Learn about strengths and weaknesses of the system by depriving it of something it needs. Delete or rename a file, and observe how the system handles the situation. (In *How to Break Software Security*, Herbert Thompson and James Whittaker describe finding an important security bug in Microsoft Internet Explorer using this exact technique.) Want to find out about initial state problems? Clean out the database (remember to make a backup copy first) and see how the system deals with empty tables. Unplug the network cable, remove a registry key, or shut down a process on which the program depends.

Hobbes's criticism of the experimental apparatus reverberates in today's testing tools. Many of the popular tools on the market are like the air pump—expensive, complicated, finicky, sometimes erratic, and in need of continuous

maintenance. It still seems a good idea to remain highly skeptical of tools that are simultaneously expensive and unreliable.

Our knowledge of our products can be both extended and limited by our models and by our test tools. We need to beware of the risk of believing that the product works because our tools show it works, and believing that our tools work because they show the product works.

Social issues—how we relate to one another, how we build credibility, and how we manage trust—are at the center of testing. We can demonstrate something about a product with a test, but a test result on its own doesn't determine the quality of the product. If testers and programmers can't agree on the meaning or the significance of a test result, there's no need to argue. Go to a higher power to end the disagreement—someone who has the authority to make quality-related decisions about the product. Hobbes's political philosophy, expressed in *Leviathan*, was that a strong, central authority was crucial to maintaining civil peace. The ideal is clearly for teams to work by consensus, but when disagreements arise, it's important for testers and programmers alike to recognize that our clients are ultimately the ones in charge.

Finally, just as there are controversies in science, there are controversies in testing. Our ideas about testing are continuously being refined and shaped by our experiments, experiences, and observations. If we're to be excellent testers, we should continue to question and critique widely accepted beliefs as Hobbes did, and we should respond to those critiques as Boyle did. {end}

**What do you know about history that you could connect with testing today?**

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

### Sticky Notes

For more on the following topics go to [www.StickyMinds.com/bettersoftware](http://www.StickyMinds.com/bettersoftware).

- References
- Video

# SALARY SURVEY

Better Software Magazine  
StickyMinds.com

## 2009

**BETTER SOFTWARE  
MAGAZINE  
DIGITAL EDITION  
EXCLUSIVE!**

Check out this  
month's digital  
edition to see the  
results of the 2009  
*Better Software*  
Magazine &  
*StickyMinds.com*  
Salary Survey.





# Par for the Course

by Patrick M. Bailey

Each step kicked up glimmering droplets of early morning dew, which soaked into the shoes of the two golfers and Willy, the caddy doing double duty.

“Nice!” said Bud Sawyer, director of support for Link2People Software, when he got a birdie on the first hole.

“I feel lucky for my bogey,” replied Janet Knight, Bud’s new boss, as they approached the second tee.

Janet was in her third week as director of software development and support, and when Bud heard she was an avid golfer, he was anxious to invite her to his favorite course. Bud had more than golfing in mind. During eighteen holes of golf, he hoped to interject a few concerns. His opportunity came sooner than expected.

“So, how’s work, Bud?”

Bud wondered if it was too soon to bring out his big concerns, but he thought “why not?” She obviously wanted to leverage the time, too.

“Not bad,” Bud said as he bent over and measured the height of the tee with two fingers. An almost ceremonial silence reigned as Bud took a practice stroke and addressed the ball, followed by the sound of the metal club’s sweet spot connecting with the ball. Ping!

“Sorry to see the slice,” Janet said. She then completed a drive, leaving her ball centered and visible on the fairway.

“So, work is ‘not bad?’” Janet picked up as the three proceeded down the fairway. “I was hoping you would say it is great.”

“I would, but we had another system rollout two weeks ago.” Bud gripped his club’s handle with its shaft resting over his shoulder as they walked.

“Problems?”

“Big time.”

“Like what?”

This was Bud’s big chance. “It’s a repeat of the usual. The code transition to level two support was fine, thankfully.”

“That’s a problem?”



ISTOCKPHOTO

“Yes and no,” Bud said, squinting to protect his eyes from the early morning sun. “I’m always tickled the way my support team finds improvements for the code after it’s been in production.”

With somewhat amused curiosity, Janet pursued this. “And that’s a problem?”

“Not really, I guess. But there are so many things we fix. It wasn’t a big deal after the first system the support team took on, but by the fifth or sixth, it added up.”

They stopped at Janet’s ball. “This is horrible.”

“Well, the code isn’t quite *that* bad.”

“I’m talking about my ball’s lie,” Janet said. “This is a horrible position to be in.” She realized the fairway was practically a semicircle, not just a simple dogleg to the right. “I should have looked at the course map. Argh!”

Bud greeted her frustrated look with a wink. Again, another hole Bud had mastered. Janet remained upbeat and did the best she could.

Throughout the course, Bud enumerated his concerns with Janet’s encouragement. He wasn’t sure he was getting through, though, because whenever

Willy suggested a club, she usually declined the suggestion. Bud wondered if this was a sign of how she did business.

“It boils down to two things,” Bud said toward the end of the game. “There is concern about the quality of the code written by the development teams and the lack of needed tools available to the maintenance team.”

“What’s wrong with the code quality? Do we have a problem with testing?”

“Testing is fine. The developers are testing exactly what they designed. It just seems like in many spots we’re always finding a *better* design. One example I can think of was this crazy, homemade queue used in lieu of the operating system’s resources. I understood the developer’s concerns, but once it was in production, it created more problems than it addressed. Fortunately, the code was constructed in such a way that we could easily plug in a completely different module to handle the message passing.”

“Oh, I heard about that,” Janet said. “I saw the help desk calls went down considerably when the change went in.”

“Exactly. That’s one example where we find things in a production environment that we think should be obvious,

## STORY LINES

- **Managers should motivate maintenance developers by reminding them that they know more about the system than the original developers did.**
- **Developers should engage and support team members when forming the vision of the final system. Maintainers must understand the constraint of a “clean piece of paper.”**
- **Listen to those living with the system every day. They understand the tools they need.**

and then we have the extra pressures of downtime to worry about.”

“Sounds like we need to get the developers to talk with your folks.”

She was listening!

Janet smiled and added, “I only wish I had a help desk call for my golf game today.”

The three arrived at the eighteenth tee. Only 108 yards, the end was near. Bud felt victorious in more ways than one. Besides having twelve fewer strokes overall than Janet, he felt he expressed his concerns at an appropriate professional level that didn’t border on whining.

As they waited for the group ahead of them to finish putting, Janet asked about the second issue.

“Tools,” Bud said. “It seems maintenance people don’t get development tools without a lot of budgetary concerns.”

“Well, why would you need as many tools as new development?”

“Mostly, it’s because we are the ‘buck-stops-here’ department. We own about twenty times more lines of code than what comes out with a new product. We’re also dealing with system interaction issues in our testing environment.”

Janet nodded, elevating Bud’s confidence to a new level. It may have been the reason his drive easily made the 108 yards, making a hole in one. “Yes!”

Janet congratulated Bud. “Well, you’re certainly not making it easy for me.” Bud wondered if there was double meaning to this.

“This is a short hole,” Janet said to Willy. “Probably needs a five iron.”

“May I make a suggestion?” Willy asked.

“Of course.”

“You might want to use a seven iron.”

“Thanks for the suggestion, but I feel better about my five.”

To Janet’s dismay, she finished the last hole with a disappointing four strokes—the drive itself overshot the hole.

Janet looked at Willy. “You were probably right about the seven iron. How did you know?”

“Given the power of your swing and what I know about the hard turf on this hole, it seemed more appropriate. I’ve caddied for a lot of people here, for a long time, and I get a pretty good idea of each situation. You did OK, though, for being out here the first time. Mr. Sawyer had the advantage from the beginning. He’s played these holes over and over again and knows every nook and cranny.”

“Sounds a lot like the world of maintenance developers,” Bud said as he puffed up his chest.

“On the other hand,” Willy continued, “we really don’t know how well Mr. Sawyer would do on a course he’s never seen before.”

Janet shot a glance at a humbled Bud. They shared a smile, tipped Willy, and headed to work. **{end}**

**Who has the most challenging job, those doing new development or those doing software maintenance?**

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

### Better Software Magazine Publisher's Statement October 2009

Published in Issue 11-7 November 2009

UNITED STATES POSTAL SERVICE

STATEMENT OF OWNERSHIP, MANAGEMENT, AND CIRCULATION

1. Publication title: Better Software
2. Publication number: 0019-578
3. Filing date: October 2009
4. Issue frequency: January, March, April, May, June, July, September, October, November, December
5. Number of issues published annually: 10
6. Annual subscription price: \$59.00
7. Complete mailing address of known office of publication: Software Quality Engineering, 330 Corporate Way Ste. 300, Orange Park, FL 32073-6214, Clay county
8. Complete mailing address of headquarters or general business office of publisher: Software Quality Engineering, 330 Corporate Way Ste. 300, Orange Park, FL 32073-6214, Clay county
9. Full names and complete address of Publisher, Editor, and Editor in Chief: Publisher: Wayne Middleton, Software Quality Engineering, 330 Corporate Way Ste. 300, Orange Park, FL 32073-6214, Clay county. Editor: Holly Bourquin, Software Quality Engineering, 330 Corporate Way Ste. 300, Orange Park, FL 32073-6214, Clay county. Editor in Chief: Heather Shanholtzer, Software Quality Engineering, 330 Corporate Way Ste. 300, Orange Park, FL 32073-6214, Clay county.
10. Owner: C. Wayne Middleton, Software Quality Engineering, 330 Corporate Way Ste. 300, Orange Park, FL 32073-6214, Clay county.
11. Known bondholders, mortgagees, and other security holders owning or holding 1 percent or more of total amount of bonds, mortgages, or other securities: None.
12. NOT NON PROFIT; DO NOT NEED TO INCLUDE IN STATEMENT
13. Publication title: Better Software
14. Issue date for circulation data below: October 2009
15. Extent and Nature of Circulation:
  - a. Total number of copies (Net press run), 21,695. Actual number of copies of single issue published nearest the filing date: 18,616
  - b. Paid and/or requested circulation. (1.) Paid/requested outside-county mail subscriptions stated on form 3541. Average number of copies each issue during preceding 12 months: 17,877. Actual number of copies of single issue published nearest the filing date: 14,430 (2.) Paid in-county subscriptions stated on form 3541. Average number of copies each issue during preceding 12 months: 0. Actual number of copies of single issue published nearest the filing date: 0. (3.) Sales through dealers and carriers, street vendors, counter sales, and other non-USPS paid distribution. Average number of copies each issue during preceding 12 months: 1,119. Actual number of copies of single issue published nearest the filing date: 840 (4.) Other classes mailed through the USPS. Average number of copies each issue during preceding 12 months: 0. Actual number of copies of single issue published nearest the filing date: 0.
  - c. Total paid and/or requested circulation [Sum of 15b. (1), (2), (3), and (4)]. Average number of copies each issue during preceding 12 months: 18,996. Actual number of copies of single issue published nearest the filing date: 15,270.
  - d. Free distribution by mail (Samples, compliment, and other free). (1.) Outside-county as stated on form 3541. Average number of copies each issue during preceding 12 months: 202. Actual number of copies of single issue published nearest the filing date: 199. (2.) In-county as stated on form 3541. Average number of copies each issue during preceding 12 months: 0. Actual number of copies of single issue published nearest the filing date: 0. (3.) Other classes mailed through the USPS. Average number of copies each issue during preceding 12 months: 7. Actual number of copies of single issue published nearest the filing date: 6. (4.) Free or Nominal Rate Distribution Outside the Mail (Carriers or Other Means). Average number of copies each issue during preceding 12 months: 2340. Actual number of copies of single issue published nearest the filing date: 3141.
  - e. Total Free or Nominal Rate Distribution [Sum of 15d (1), (2), (3) and (4)]. Average number of copies each issue during preceding 12 months: 2,629. Actual number of copies of single issue published nearest the filing date: 3,346.
  - f. Total distribution (Sum of 15c. and 15f.). Average number of copies each issue during preceding 12 months: 21,695. Actual number of copies of single issue published nearest the filing date: 18,616.
  - g. Copies not distributed. Average number of copies each issue during preceding 12 months: 0. Actual number of copies of single issue published nearest the filing date: 0.
  - h. Total (Sum of 15f. and g.). Average number of copies each issue during preceding 12 months: 21,695. Actual number of copies of single issue published nearest the filing date: 18,616.
  - i. Percent paid and/or requested circulation (15c. divided by 15f. times 100). Average number of copies each issue during preceding 12 months: 87%. Actual number of copies of single issue published nearest the filing date: 82%.
16. Publication of Statement of Ownership. Publication is printed in the November 2009 (11-7) issue.

I certify that the statements made by me above are correct and complete.

Wayne Middleton, Publisher

# COUNTDOWN TO AGILITY

TOP **10**  
TEN



ISTOCKPHOTO

**In May 2009**, *Outside* magazine recognized Rally Software as one of the top ten companies to work for in America. That spawned a conversation between Ryan Martens, Rally's founder, and me about how agile creates great companies.

Rally holds six core values: Make and meet commitments, give back to the community, balance work and life, base decision making on theory, respect people, and create your own reality. As we grow and change, we continuously evaluate how faithful we are to these values. With these core values, we seek continually to strengthen our corporate culture.

Based on our conversation about these core values, Ryan and my brainstorm led to these ten characteristics we think are essential to building a truly great agile organization.





# CHARACTERISTICS *of an* **AGILE** ORGANIZATION

**by JEAN TABAKA**

## 10. Work/Life Balance and Consistent Delivery

“Work/life balance?” people ask. “Can there be real commitment to product release dates while still embracing this balance?” Yes. Agile organizations seek people who understand the power of consistent product delivery and embrace the reality of their lives: commitment to company success and commitment to non-work goals.

Agile companies believe that consistent delivery relies on each individual’s ability to maintain a realistic, sustainable life balance. These companies create long-term profitability and ensure employee morale doesn’t fizzle in the process. Here’s how:

**Encourage personal commitment and discipline**—Throughout your organization, empower teams of disciplined individuals dedicated to their personal

growth and to the continuous improvement of the company. Apply this guidance across all departments and all divisions. Personal balance and sustainability promote long-term organizational commitment.

**Create consistent product delivery through discipline**—A discipline of shorter release cycles helps team members commit based on past performance. They track their velocity. And through this trending, teams learn naturally to absorb team member personal needs (maternity leave, injuries, soccer practice) while committing to release plans. In this way, agile teams create a trend of feature completion that defines consistent delivery.

## 9. Servant and Leader

Imagine a CEO job interview that asks, “Can you lead with a vision and

then entrust your teams to create the best solutions? Are you prepared to congratulate your organization when things are going well? Can you turn to yourself during difficult times and ask, ‘How am I not serving my employees?’”

Imagine the project manager who says, “I do everything I can to support my team in making its commitments. I don’t make decisions for them, I don’t commit for them, and I don’t determine their tasks or make task assignments. However, I relentlessly seek to support the team’s commitments.”

Agile organizations ask this of their full spectrum of managers. They act as servant leaders. They serve by leading and lead by serving. Servant leaders build team consensus for the most powerful commitments to company success. This servant role stretches beyond ScrumMaster or agile coach; it is em-

braced throughout the organization. In these collaborative environments, the agile organization only retains managers who can truly serve the team.

How can managers be both a servant and a leader? Here are three of Robert Greenleaf's ten characteristics of a servant leader [1]:

**Goal setting**—Servant leaders hold a greater purpose and aim beyond themselves. With their personal vision, they encourage employees to strive toward a greater purpose beyond profits alone. These leaders respect employees, value customers, and maintain healthy relationships with competitors, suppliers, and the larger community.

**Ability to listen**—Servant leaders create more informed visions through more insights by more inquiry. They seek the wisdom of the team. In return, the agile organization creates more commitment to this servant leader's vision.

**Tolerance of imperfection**—Agile organization leaders let go of their personal sense of perfection. Instead, they ask what their teams think is the best, most informed action. These servant leaders seek innovative solutions outside their personal perspectives alone. They hold and continually re-evaluate their overall vision—the true north for the organization—without ego, based on organizational feedback.

## 8. Sustainable and Successful

This characteristic of a great agile organization relates very closely to characteristic No.10. At the company level, agile organizations emphasize sustainable and stable growth and product growth while still keeping an eye on real-time success.

**Stability**—Agile organizations seek stable, consistent team delivery over individual heroism as a means to success. These companies eliminate multitasking for “getting the most” out of people. Rather, employees are encouraged to single task: complete one item of work before switching to another. Agile organizations eliminate overly matrixed project teams, that is, project teams are not made up of people working on many other projects. With heroics, multitasking, and project matrixing removed,

the stable environment is the key to real success.

**Sustainable pace**—Closely related to stability for an agile organization is sustainable pace. This translates to sustainable regular feature delivery that relies on short feedback loops and small increments, all managed in a regular cadence. This sustainable feature delivery supports continuous improvement on success and innovation.

**Redefining success**—For agile organizations, success is no longer about following a plan. Success is delivering value to the customer. Maintaining sustainable pace and stability, all divisions of the company focus on customer value. Success also means keeping an eye on innovation, creating faster learning, and continually responding to customers, risks, and market shifts. As a result, new metrics and tools are invoked to maintain this agile perspective on success.

## 7. Contributing to the Community and Maintaining a Profitable Company

Agile organizations have a strong focus on community: the professional community, the local community, and the global community. They believe that giving back to the community is essential to being a profitable company. If someone asks, “What are you in business to do?” and you respond, “To make 22 percent pre-tax profit,” you are not at the right level of maturity. You are not a true agile organization.

Great agile organizations' key leaders and stakeholders define and agree upon the business's core purpose and profitability, determine the positive impact the company can have on the community, and truly commit to the comingling of these goals. Guided by this greater purpose and the servant leadership, these organizations move from a single bottom-line impact (economic benefit) to a triple bottom-line impact (social, environmental, and economic benefit).

**Social standpoint/Strategic corporate social responsibility**—In the industrial world, business's role was defined in terms of economic viability. Church and state took on responsibility for commu-

nity viability. During the 1970s, states started to sag under the burden of their social responsibilities, which led to the formation of corporate foundations that funded large non-profit organizations. Still, individual businesses stayed focused on their corporate profits.

Now, the twenty-first century general public has rising levels of expectations regarding business responsibility. One strong voice in this vision is Peter Senge. In *The Necessary Revolution* [2], Senge calls upon businesses to dedicate themselves not just to renewable environments but also to what he refers to as restorative economics. Agile organizations embracing these externalities recognize the powerful internal value. Such forward-looking companies attack waste in how they work and deliver value, and they go beyond the company walls to attack all forms of waste in the community at large. In this way, agile organizations apply their practice of continuous improvement not just to their corporate contribution but also to their community involvement.

## 6. Collaborative and Smart

Talking with CxOs about adopting a collaborative culture can bring out great resistance. Some of it is very direct; often it is passive-aggressive. The problem lies in deep assumptions and old wounds:

- Collaboration means dumbed down.
- Collaboration means group think.
- Collaboration means bringing decision making to a halt.
- Command and control is the only way to avoid all these productivity killers.

When Ryan and I talk about “collaboration” in an organization, we see collaboration as a means to make better, more-informed decisions through greater insights. Collaboration and smart not only should go together, they must.

Great agile organizations hire smart people. If we support their intelligence and help them get smarter, we advance the overall intelligence of the organization. Collaboration helps spread the wealth of intellect. As Catherine May has pointed out, when we cling to command and control for making decisions,

“Great agile organizations’ key leaders and stakeholders define and agree upon the business’s core purpose and profitability, determine the positive impact the company can have on the community, and truly commit to the comingling of these goals.”

our organizations collectively lower their cognitive skills and employees become more error prone [3].

Conversely, in environments where the practice of collaboration is applied, people’s cognitive skills improve. Empowerment raises people’s IQs. Which statement would you rather tell your family: “I lower people’s IQs for a living!” or “I actually make people smarter!”?

And, the positive impacts go beyond individual wisdom. James Surowiecki has noted that organizations make better decisions when they empower teams [4]. This aligns with a fundamental lean principle: Gather as many insights as possible before making a decision. For agile organizations, facilitated collaborative dialogue is the key.

Organizations that equate collaboration with group-think (or dumbing down) suggest old wounds from dysfunctional environments: Trust and safety were low, meetings didn’t invite goal-oriented dialogue, and quick decisions were highly valued. These organizations never experienced the power of “storming” [5]. Agile organizations invite powerful, open divergence of ideas in order to create informed decisions through a convergence of ideas.

And, collaboration doesn’t just happen on its own. Agile organizations rely on effective facilitation. (*Collaboration Explained* by Jean Tabaka is a good guide!) They apply facilitated processes to gather insights, seek conflict, bring dialogue, and guide teams to insightful decisions. Facilitation eliminates energy-draining rat holes, compromise, or capitulation.

It ensures discussion around useful dialogue versus interesting debate. And, it prevents what we call “loudest-voice-driven decisions.”

## 5. Bottom-up and Top-down Decision Making

Nonaka [6] and Takeuchi [7] claim that knowledge-creating companies embrace information flow in which the leaders are informed by the knowledge workers (their term for team members). The knowledge workers in turn are informed by the leaders. This is “bottom-up” and “top-down” decision making.

Agile organizations use this circular flow when codifying knowledge into “standards” or “best practices.” That is, teams collect what Nonaka and Takeuchi refer to as “tacit knowledge,” the teams’ tribal sense of what works well. Tacit knowledge informs “explicit knowledge,” those useful standards to help guide other groups. So informed, the “top-down” decisions by leaders deliver well-informed guidance to the knowledge workers, which then sets the cycle in motion again. Top-down vision from leaders guides bottom-up knowledge from teams.

At Rally, we use Five Levels of Planning to guide this whole organization practice. The highest level of planning—the vision—feeds and is fed by subsequent levels, down to the lowest level of planning—the detailed, daily work. Bottom-up and top-down are structured in a virtuous circle through the five levels.

The basic flow of the Five Levels of Planning turns *product vision* into a *product roadmap*. In turn, this roadmap

guides each *release plan* that then guides the *iteration plans* in the release. Finally, the iteration goals help set the *daily plans* in motion.

And, the reverse is also true. The daily plan does more than guide the current iteration plan it also helps adjust expectations and intentions for the next iteration. Iterations then guide the release plan about the current state of the release, for instance, what features can be projected? The empirical data from each release helps the organization evaluate and update the product roadmap. Finally, the product-roadmap updates guide continuous reflection on the product vision.

The agile organization’s vision relies on and supports the vision for each product. And so, the bottom-up and top-down decision model guides the organization and its products.

## 4. Personal Flexibility and Rhythm

Rhythm is an important discipline when engaging in flexibility, both in our personal and our corporate lives. We contend that agile organizations rely on both. In Jim Collins’s book *How the Mighty Fail*, he describes his disciplined daily rhythm as a set of activities: 53 percent creative, 28 percent teaching, 19 percent other. This level of discipline may seem extreme, but it is an example of one of our fundamental agile organizational characteristics: personal flexibility and rhythm.

Agile organizations use rhythm to their advantage. They create a regular cycle of delivery and value, and they



create dedicated time toward innovation. Disciplined rhythm relies on slack, not only to allow for the non-deterministic nature of our work but also to ensure a natural sense of personal flexibility. The combination of regular rhythm of delivery, slack, and personal flexibility makes for a great agile company.

Rhythm and flexibility guide agile organizations in how they create a higher standard of living both for the organization and for the individuals. Once the sustainable rhythm is reflected in team members' calendars, everything else works around that. The high visibility into organizational and individual rhythms invites flexibility without surprises. This includes accommodating not only vacations and meetings but also creating flexibility for how and when to address issues in the regularly scheduled meetings. All of this leads to much more flow time and allows knowledge workers to stay on the path without undue stress.

Just as the Five Levels of Planning has a structure of different layers of planning attributes, rhythm also occurs at different levels, each with its particular attributes.

At a company level, the agile organization uses a quarterly rhythm for collaborative corporate planning. Based on the sustainable rhythm of previous quarters, the corporation sets high-level goals for the current quarter. It sets in motion the rhythm that then will guide the monthly, weekly, and daily rhythms of the company.

At Rally, our engineering group uses an eight-week release cycle, six of which are dedicated to actual development and testing. We have half a week focused on specific technical debt and one week dedicated to developer innovation (our "hackathon"). The remaining half a week is for planning and creative work associated with the next release cycle. This regular cycle puts all of our product lines into the same planning rhythm. That rhythm coordinates the team members in a continuous cycle of delivery. The intent of the cycle is not to constrict; rather, it encourages team members to manage and allocate their time both at work and personally.

For us, rhythm with appropriate

flexibility also serves another purpose. Rhythm invites high throughput, less noise, and less task switching, which can equate to more value delivery. Using Jim Collins's terms, that equates to 63 percent in value-added development, 19 percent in creative or innovative time, and 18 percent in the other category. The rhythm ensures flexibility and growth, and flexibility and growth help build a greater flow of value.

### 3. Quality and Faster

Ryan and I have for a while been talking about quality *and* faster as necessary components of an agile organization. Often, pushing more and more features into a release and to the market necessitates a loss—or at least great variance—in quality. In an agile organization, our experience is quite the opposite.

For a non-agile case study, let's look at the RCA example Ralph Aguayo describes in his book, *Dr. Deming: The American Who Taught the Japanese About Quality*. At its height, RCA's business model focused on getting as many televisions into the hands of consumers as possible. Quality was secondary to the savings RCA could make from cheap components and speed of product to market. Cheap and fast meant greater profits.

Unfortunately for RCA, their model began to crumble when television sets with a lot of defects started being returned within the warranty period. The cost of repairing these faulty sets ended up being 25 percent more than the cost of manufacturing the set.

Do you have a similar situation in your software product delivery? Are you trying to increase profits by pushing features out faster with "defective parts" because they are "cheaper"? In other words, is your emphasis on speed at the expense of quality, coupled with very large feature-rich releases? This is exactly the wrong way to create greater value and profitability.

In an agile organization, the entire organization concentrates on value delivery and quick feedback regarding that value. That is, "Are we delivering the right features to the market?" In turn, a real agile organization understands "there ain't no

such thing as a free lunch." Teams increase throughput by holding defects in check. They depend on a "stop the line" mentality. Fewer defects free up the organization to concentrate on faster value feedback. In truth, in an agile organization, quality is everyone's job.

And so, the notion of "quality and faster" is not as counterintuitive as we once may have thought. We need fast feedback loops in order to improve continuously how we deliver high-quality (defect-free and valuable) features into the market. Agile organizations rely on high-quality features in order to respond ever faster and more innovatively to the market.

## 2. Creating Your Own Reality and Corporate Vision

For an agile organization, we find another odd pairing for success—creating your own reality coupled with maintaining a corporate vision. In a traditional organization, you might think that, in order to implement the corporate vision, you must have clearly defined, rigid roles. In hiring, you recruit and interview specifically for these roles. Additionally, movement through the organization can be very difficult, not just up the hierarchy but across the organization, as well.

In agile organizations, the opposite is true. Implementing a corporate vision requires seeking and hiring self-motivated people who are prepared to create their own reality. Rather than relying on rigid role descriptions for success, the agile organization relies on great individuals. These people are prepared to collaborate, innovate, and seek to learn what their best contribution to the organization can be.

Jim Collins in his book *Good to Great* talks about "getting the right people on the bus." That means finding good people and trusting them to do the right thing. As the organization provides the corporate vision, the individuals create their role in that vision.

"Create your own reality" assumes a fundamental truth: People do their best work when they are passionate about what they are doing. And, when people are doing their best work, they create

the most sustainable commitment and dedication to the corporate vision. Agile organizations take advantage of this unlikely coupling by using roles more as a guide than a dictate. Role details emerge according to the work, and individuals self-assign according to their best work and their passion. In this virtuous circle, both the organization and the individual win.

## 1. Strong Corporate Culture and Metrics

Our number one characteristic of an agile organization goes back to a very strong element in agile culture. When I am teaching introductory agile or Certified ScrumMaster classes, I guide participants always to begin with the values. Agile values guide the agile principles that help teams inspect and adapt their agile practices. In so doing, you build a strong values-based culture. For the agile organization, this means openly committing beyond the team to a strong corporate culture based on values.

With regard to metrics, the ultimate measure for an agile organization is “Are we a great company?” Metrics must show how the organization is great or how it can create a path to greatness. We formulate metrics that show growth, commitment, vision, and rhythm all around being great. And, for a great agile organization, we measure not just the company but also its contribution to the community, its greater purpose. We check in on work/life balance, bottom-up and top-down decision making, servant leadership practices, and how we maintain rhythm and flexibility. We look at the percent of time we spend on innovation and technical debt reduction.

These are all metrics worth collecting in an agile organization. While we still track revenue and profits, we now use these additional non-traditional metrics to create a strong, sustainable corporate culture that in turn makes a great agile organization. {end}

### Sticky Notes

For more on the following topics go to [www.StickyMinds.com/bettersoftware](http://www.StickyMinds.com/bettersoftware).

- References
- Further reading



## Wind River Test Management

### *A Collaborative Automation Solution*

Software quality assurance has never been more important—or more difficult.

Wind River Test Management brings teams together with a collaborative automation solution, specifically designed for embedded device test and diagnostics. Its unique run-time analytics help you focus your time and resources on what really needs testing, so you can delivery higher-quality products on time and on budget, and with greater confidence.

Trust the device software leader.  
Wind River.

Learn more:  
[www.windriver.com/products/test\\_management](http://www.windriver.com/products/test_management)

Be sure to visit the Wind River booth at the STARWEST Expo, October 5–9, 2009, at the Disneyland Hotel, Anaheim, Calif.  
[www.sqe.com/starwest](http://www.sqe.com/starwest)

## WIND RIVER

© 2009 Wind River Systems, Inc. The Wind River logo is a trademark, and Wind River is a registered trademark of Wind River Systems, Inc. Other marks are the property of their respective owners.

# Telling Better User Stories Mapping the Path to Success



by Jeff Patton





# The User Story

has emerged as a standard practice in agile development processes. While the idea of user stories is simple on the surface, there are challenges to working with them—particularly when working with the multiple stories required to create a successful product. User story mapping is a useful way to organize, decompose, and prioritize user stories. A good story map keeps conversation about your product alive and productive throughout its construction. But, to understand user story mapping, we first need to understand user stories.

## User Stories

### USER STORIES ARE FOR CONVERSATION

When I ask people for a definition of user story, the most common response I receive is “a token for a conversation.” Extreme Programming (XP) popularized user stories and that particular definition, which came from Alistair Cockburn during some of the earliest discussions about XP. The name “user story” comes from the idea that we should tell the “story” of what’s needed and why it’s valuable and not just document requirements. In telling the story, we’ll arrive at a better idea of what to build. Having that conversation is the most important aspect of the user story.

A user story is a boundary object—a point of agreement that different groups with different concerns use to collaborate. While every group uses the same name for the story, each tends to consider only its own concerns while discussing stories. Users think of the stories as descriptions of their needs. Business people think of them as product features that will generate return on investment. Developers think of them as the specifications for software to build. Testers think of them as things they need to validate. Project managers think of them as work to schedule into a release. All of these views are correct, but all of them are also incomplete.

A user story isn’t an ideal user problem description, software specification, test script, or project management task. It’s the token that allows these communities to think about their own needs while they collaborate over their

mutual concern—the software they’d all like to see developed.

It’s difficult to engage in conversations without emphasizing our own concerns. A story that works well as a boundary object has meaning to everyone involved with the project but isn’t so specific that it loses meaning for any one person. Extra-specific details cloud the conversation and erode the value of the story.

Of course, individuals need all those details, and the details do need to be discussed. The trick is to keep very specific details to yourself. Understand that they are only relevant to some conversations you need to have, and keep them handy for those conversations.

### ORGANIZING THE PRIORITIZED STORY BACKLOG

User stories that describe only a small chunk of software are quicker to code and test and, when completed, show progress. Showing progress in the form of working software is an impor-

allows us to prioritize and schedule both the detailed conversations about the stories and their construction. But, when the backlog consists of hundreds of stories, it can be virtually impossible to create a mental picture of the product as a whole.

People looking at a prioritized backlog can have good conversations about the schedule, but other conversations about the product can be difficult. It’s not enough to write stories and prioritize your backlog for implementation. A backlog that really supports conversation has three important qualities:

**Descriptive**—It helps us understand the users and their needs and how the software addresses those needs. It helps us visualize the entire product.

**Right sized**—High-level conversations have high-level stories; detailed conversations have more detailed stories. Conversations must be enabled at the executive level and at the detailed development level.

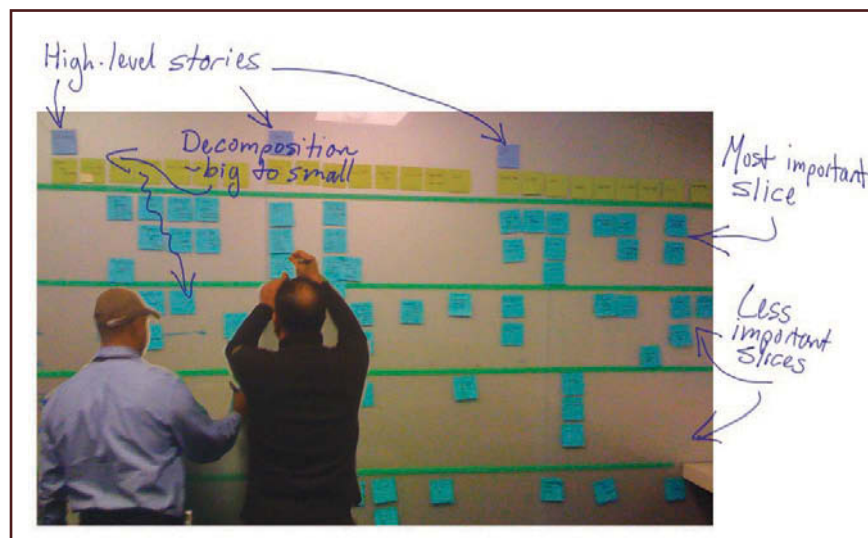


Figure 1: User Story Mapping

tant characteristic of agile development. If I’m scheduling work into a short timebox or sprint, or figuring out the details of software to build and test, small is good. But small is a relative term. The ideal-size story from a user’s perspective may be far too large from a developer’s perspective.

A common goal in agile—and specifically in the Scrum community—is maintaining a “prioritized backlog.” Organizing user stories into a backlog

**Prioritized**—The most valuable items, the ones we need to focus on first, can easily be identified.

### USER STORY MAPPING

A user story map places high-level stories in a meaningful order, typically left to right and top to bottom as shown in figure 1. Often, this reflects the user’s workflow order or business process and allows us to tell a meaningful story about a significant part of the entire system

without getting lost in the details. These top-level stories form the backbone of our application.

Below each of the top stories is a similar left-to-right “decomposition” of that story into mid-level stories that break it down. The mid-level stories could be the steps in a business process or details of an activity in which a person engages. During a conversation, we can drop down to this level to dig into the details of any one top-level story. Below each of these mid-level stories are more stories that decompose into more detailed stories. To save space, we stack the detailed stories in a nice vertical column.

So far, the shape of the map has helped us get to the descriptive and right-sized qualities we need now to layer in prioritization. The map is “sliced” into product releases. We’ll only slice the detailed stories since they’re the smallest bits—the parts we’re likely to build in a short iteration or sprint. Slicing the map lets us see what’s in each release over time and how the top stories grow in capability with each release.

With a story map, it’s easy to spot holes in our conversations—places where the story map is lacking and where more mid-level or detailed stories should be identified. In some cases, we’ll identify planned omissions—places where we’ve purposely deferred functionality for later release.

## Working with Story Maps

Working with a story map follows a simple process:

### 1. CREATE A STORY MAP

Start by identifying user stories and arranging them into the shape of a story map. Focus on breadth, not depth. The fastest way to generate a lot of stories is to think through a user’s experience. Let’s consider “booking an airline flight,” since most of us have had this experience.

The basic tasks to book a flight are:

1. Choose an origin and destination.
2. Choose the departure and return dates.
3. Search for flights.
4. Review the flight options.

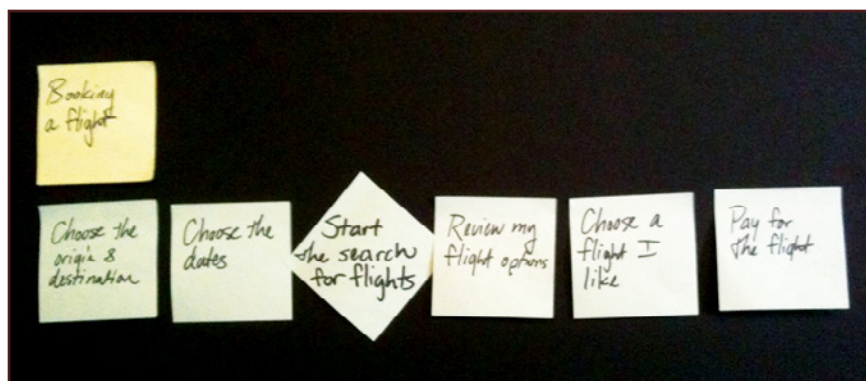


Figure 2

5. Choose the flights.
6. Pay for the flight.

I’ll write these user tasks down on stickies and lay them out left to right. On a different color sticky, I write “Booking a flight” and place the sticky on top of my neat row as shown in figure 2.

I used the phrase “user task” to describe things a user would like to accomplish. User tasks usually start with a verb to help us understand what they’re doing. We’ll call these “task-centric user stories.”

To help us remember that user stories should be about users, this popular template has emerged:

As a [type of user],  
I want to [do something]  
so that [I get some value].

I like using this template to check my stories: “As a *traveler*, I want to *choose the flights I like* so that *I can fly at the times that suit my travel plans*.” However, I don’t write all those words because I don’t need them. The “working words” are the short verb phrases such as “choose the flights.” That’s the least I can write and still be clear enough about what a user of a booking system would want to do. Writing too many words makes the map hard to read and diminishes its value as a conversation piece.

At the outset, we’ll want to identify the full breadth of a potential system. Think of the other big activities that people might engage in with the reservation system. Based on my experience, my initial list is:

1. Change my flight.
2. Check in.
3. Book a flight I often take.

I could go on, and, if I were building such a system, I would.

These big activities bundle a number of smaller user tasks performed to achieve a larger goal. These smaller tasks often can be performed in varying order. For instance, I might stop halfway through booking a flight and go back to the beginning. I might always do some smaller tasks, like choosing flight dates, and rarely do others, like selecting special meal options. But, all those smaller tasks support a common goal—the user activity.

Because I know that when I’m engaged in an activity like “booking a flight” I might skip steps or back up, I don’t get too concerned about the order in which I put the stickies. They’re in the order that helps me best discuss it with others.

In Mike Cohn’s book *User Stories Applied*, he coins the term “epic” to refer to a big user story, one that should be broken down further before placing the story into a next sprint or iteration. These user-centric activities that we’ve placed as top-level stories in our map are likely “epics.” But, the task-centric stories we’ve placed as mid-level stories could be as well. At this point, they’re all too big and too poorly understood to be built, so don’t get caught up in the epic versus story distinction.

Thinking through the user experience as activities and tasks will give you a beginning story map a mile wide and an inch deep. Building the story map is often eye opening. For developers and testers who may be used to focusing on the details, this may be the first time they’ve seen the big picture.



## 2. FILL IN AND VALIDATE

Once the map looks right, start diving a bit deeper. Discussing the map with others helps you identify missing stories or alternative ways of doing things.

Let's go back to "booking a flight" and start with the task "choose an origin and destination." Thinking about the details, we can identify smaller sub-tasks inside that task: "enter airport code," "enter city name," or "look up airport code." Since I travel to New York City frequently, and three major airports serve that area, I often "choose to include nearby airports" in my search. All these are tasks I write on stickies and hang below the "choose an origin and destination" sticky. Doing this for all the mid-level, task-centric stories in the user-centric story at the top results in a more complete map.

Each kind of user is coming to our booking system with varying amounts of experience but with a goal in mind. These are user roles, and they overlap a bit. Think of roles like hats you wear and can change as your goal changes. For example, I'm often a frequent business traveler, but sometimes I'm a family vacation booker. And, because I'm a cheapskate, I always have a bit of price shopper in me.

If I put on my frequent business traveler hat and walk the story map, I remember my goals a bit better and add to my tasks. I often "check my current mileage balance" and "place my name on the first-class upgrade list." I hate spending too much time on planes so, while reviewing my flight options, I always "look at the number of stops" and "look for shortest flight duration."



Figure 3

But, I'm a frequent flyer and probably not the best example of all the types of users who might interact with this system.

I took a moment to brainstorm different kinds of users who could use our system and add them to my map, as shown in figure 3.

- Frequent business traveler
- Family vacation booker
- Price shopper
- Infrequent, perplexed traveler

When I put my family vacation booker hat on, I remember other tasks like "entering family member names and ages" and "saving flight itineraries to discuss later" with my wife, who needs to make this decision with me.

Understanding different types of users and walking the map from their perspective expand the map with things you might have forgotten otherwise.

It's also a good idea to begin walking the map with others on the team including engineers, testers, and other

stakeholders. They begin to understand the user's experience and why it is the way it is. They can point out areas where writing software to support such a user task may be difficult. Or, better yet, when armed with a good understanding of the user experience, they can identify product solution ideas that would really help the users.

As you fill in and validate the map, you'll add, split, reorganize, and annotate stories with details and cool solution ideas. As you walk and talk through the story map with a variety of people, you'll gain confidence that it's complete.

When you're comfortable that you haven't left out anything big, it's time to move on to make choices about what to build now, what to build later, and, possibly, what never to build at all.

## 3. PLAN RELEASES

Armed with a strong understanding, we can begin to prioritize what's most important to implement in our software and what we might be able to do without. We then can slice our map into coherent releases of detailed stories.

When trying to identify what to build first, I focus on two levels of planning. First, I think about what I should release first to create a minimally viable product and what should come next in my product roadmap. This is called release planning. Knowing the goals of a first release allows me to focus on what to build in that first release. I'll target functionality that helps me learn fast about the product. I'll try to create a fully functional "walking skeleton" [1] of the product as soon as possible. Keep these two levels of planning in mind as you start to prioritize stories.

If I had a nickel for every time I heard someone say, "Prioritization is my biggest challenge when working with stories," I'd have a lot of nickels. The trick to prioritization is not to work with the stories first but to work outside in the product's context. By context, I mean the customers and users of our product and the business strategy that motivates building the product.

We have previously identified a few user types such as frequent business traveler and family vacation booker. Start by prioritizing those.

Now, whether business travelers or vacation bookers are more important to our organization depends on what we're trying to do with the product. Are we trying to build the best business travel site on the Web? Or, do we think that the vacation planner is an underserved market and we'd do best to focus on them? These sorts of decisions are strategic business decisions. Don't try to prioritize details about what to build until you're clear on the business strategy.

Business stakeholders ideally will make strategy decisions informed by market research, revenue- or cost-saving goals, or other considerations. Once those decisions are made, we can see more clearly which users are most critical to realizing our business strategy. Given that, we can begin to prioritize what those users need to do with the system.

The left-to-right organization of the story map supports good discussion. The top-to-bottom organization shows decomposition. When it comes time to build specific bits of software, it's the smallest detail stories that we'll want to build, so they're the ones we'll prioritize into releases.

I like to do prioritizing as a group so we're all on the same page. First, post and discuss the business strategy and organize the user types into priority order. Then, create slices in our story map by adding tapelines left to right to create horizontal slices, each representing a release.

Together, we'll begin moving the smallest stories up into their respective release slices. The first release slice on the top should contain all the stories needed to deliver our minimal viable product. Each slice below that will add a layer of capability to build up the product one release at a time.

Don't get tangled up in prioritizing one high-level or mid-level story against another. It doesn't do any good to debate "What's higher priority: choosing origin and destination or entering travel dates?" It's an irrelevant question since we know the product needs to support the activity of booking a flight and both tasks are critical for that. Focus on the detail-level stories and what slice they belong in. Focus on identifying the

smallest first slice that will support your target user and business strategy.

While planning, we may split stories, add more stories, reword stories to make them clearer, or reorganize the map to better support discussion.

At the end of this exercise, we have a release roadmap ordered top to bottom where each slice should be in a coherent product release. We can test a slice's coherence by walking it left to right and talking through a user's experience. If we start with our highest priority user and describe his experience, pointing out stories as we go, we do so only by discussing stories inside the release slice. If we can't tell the big story of our user's experience with the first release without dipping down into second-release functionality, then we know we have not included sufficient functionality.

When you're comfortable with a release plan, you can summarize each slice or release by giving it a name, writing a couple of short sentences expressing the business benefit for building it and what the user's benefit is when using it. This list of names and benefits—along with a few bulleted, high-level stories—is our product roadmap.

#### 4. CONSTRUCT SOFTWARE

It's time to get to work building something. Working with the detailed stories and moving story by story, we'll choose what we'd like to implement in the upcoming development cycle. Over time, we see the stories in our release being completed.

During product construction, keep the story map visible so it can help give context to further detailed discussions. Ignore the slices for releases other than the one on which you're working. I remove them from the story map completely, setting them aside to discuss as we get ready for the next release. Break the current release again into three thinner slices. Using a chess game metaphor, I call the slices "opening game," "mid game," and "end game."

The opening game slice contains the simplest possible functional version of the product. Remember, we're not releasing yet, so it doesn't need to be release ready. But, it should help us validate our functional scope, our architecture, and,

at the end, be able to support end-to-end performance and load testing. It ideally contains a little bit from every major activity in the system. I refer to these activities as the "backbone" and this very thin, not quite releasable product, as our "walking skeleton."

Mid game stories focus on getting the system to full functional completeness. It's important in mid game to ramp up the amount of validation we do with users and with the system architecture.

End game stories focus on polish, fit, and finish. We'll need to make sure we have time during end game to accommodate all the "predictably unpredictable" work that comes in as a result of the user validation and other testing we've been doing.

### Keep Your Eye on the Prize

Keeping your eye on a coherent, growing system and constantly evaluating it against the benefits you targeted in your release plan are critical. And, it's not something one person does alone. The entire team needs to help keep an eye on these things.

Keep visible your business strategy, target user types, release roadmap, and user story map. Mark off stories that are done. Discuss how ready your product is to release at a high level, user activity by user activity. Use the map to pull your attention away from the minutia of day-to-day work and back to the big picture of the product you're growing. {end}





Prepare  
to

# Succeed

A Guide to

Effective

Code Review

by Jason Cohen



Most people despise doing code reviews. Reviews take time away from the things programmers enjoy, such as crafting code. As it is, no one has enough time to keep up with demanding release schedules. Worse, code reviews are actual invitations for criticism and provide an ideal opportunity to showcase to the world all of the dumb mistakes you've made.

At the same time, most programmers will admit that if someone else were to look over their code, bugs would be found. And, even though most programmers hate reviews, the concept makes sense if you think it through. Two brains are better than one. You wouldn't write a book or even an article without having someone else—usually several people—review it. Extra eyeballs catch things even the most diligent author overlooks. Whether code or prose, I sometimes review my own writing five times and still miss errors that another person catches immediately—not because I'm a moron, but because I wrote it and I'm too close to it to see the problems.

Typos and clumsy sentences in an article won't crash a system or cause data loss, but if I'm writing code, the stakes are much higher. An error in my code could cause a catastrophe, so that makes another pair of eyes on the code I write even more important.

"OK," you say. "Maybe we *should* be doing code reviews. But how do we do them efficiently, so they're worthwhile? How do we make them not painful? How do we position our code reviews for success and actually get people to do them?"

I've spent a lot of time researching and doing code reviews and working with teams that do them. And, while code review processes and metrics are fine in theory, successfully implementing code review for a team is an art. To get started and position the endeavor for success, I've found the following approach works best.

## Start Small

The best way to start doing code reviews is at a small scale, on only a handful of files. Don't plan to review all of your code at the outset. Instead, use the following suggestions for limiting review scope to help you realize the most

useful and persuasive results. Choose as many of them as make sense for your team:

- Review modifications to the core module that is referenced by all other code.
- Only review changes to the stable branch.
- Don't review entire files; just review the changes (code review tools or version control systems can highlight them for you).
- Let the developers choose the top ten most risky files to review. Also let them decide when else a review is necessary, such as when they know a domain expert can lend assistance or when they're feeling "iffy" about a particular code change.
- Review only unit tests (after all, they're source code files and can be reviewed like everything else). If these are complete and well written, you can safely fix bugs or refactor later.

## Motivate Through Your Own Real Data

Now let's address the most difficult question: How do you motivate people to do code reviews? Everyone wants to improve code quality, but do they really believe code review is an efficient way to do that? A simple way to get buy-in (and subsequent participation) is to demonstrate the value of code reviews to your team on its specific projects.

One easy way to start is by sharing this factoid I've found in my research, gleaned from stats I collected working with thousands of programmers: When code reviews are conducted efficiently, the average team finds and fixes a bug every ten to fifteen minutes.

That's pretty powerful. In fact, there's no other process in software development that identifies and fixes bugs that quickly.

"But will it really save us that much?" Good question. The easiest way to find out is to try it. First, get your team members to agree to review code for one week, spending a total of two to two-and-a-half hours each. Then, meet with team members to establish the process to use (described in the Ways to Review

## WAYS TO REVIEW CODE

*You have a choice of multiple methods to review code. Pick the one that works best for your team.*

**Formal meetings**—Teams print out code and review it together in meetings. The traditional method of code review, formal meetings are eschewed by many because they are usually excruciating and too time consuming to be practical.

**Pair programming**—This is sometimes the code review method of choice for agile developers. Programmers write all code as a two-person team and can catch each other's mistakes.

**Walkthrough/over the shoulder**—This informal approach works well when developers are in the same office. The author steps through his or her code with the reviewer.

**Email pass-around**—This informal method works pretty well for geographically distributed teams. They can email each other code as it's developed and send comments back and forth; however, they still have to manually attach files and reference line numbers.

**Tool-assisted**—The most efficient method, tool-assisted code reviews use dedicated software to automate uploads, facilitate online discussions (sometimes with comments directly on the code itself, so programmers don't have to search for line numbers), and keep the process moving along. Tool-assisted reviews take about one-fifth of the time of formal reviews because the tools handle the tedious parts. They work well for both local and distributed teams.

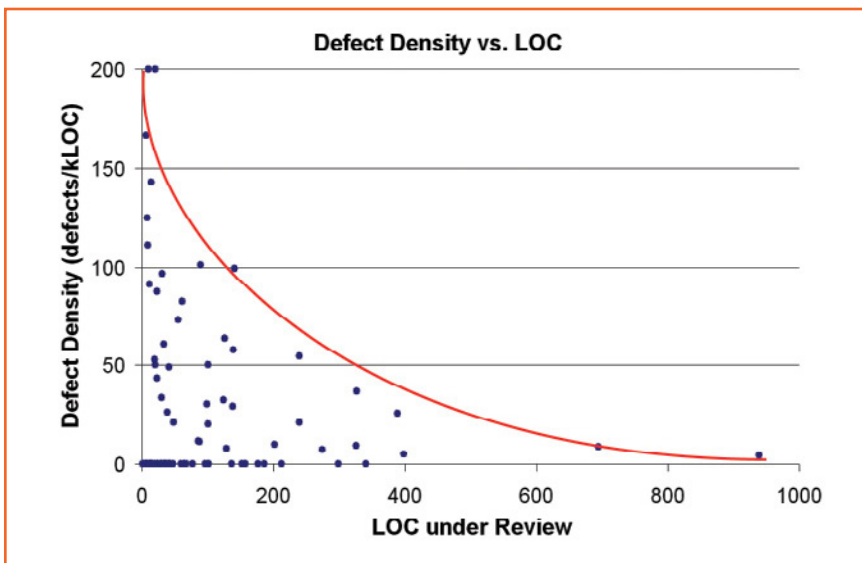


Figure 1: Defect density dramatically decreases when the number of lines of inspection goes above 200 and is almost zero after 400.

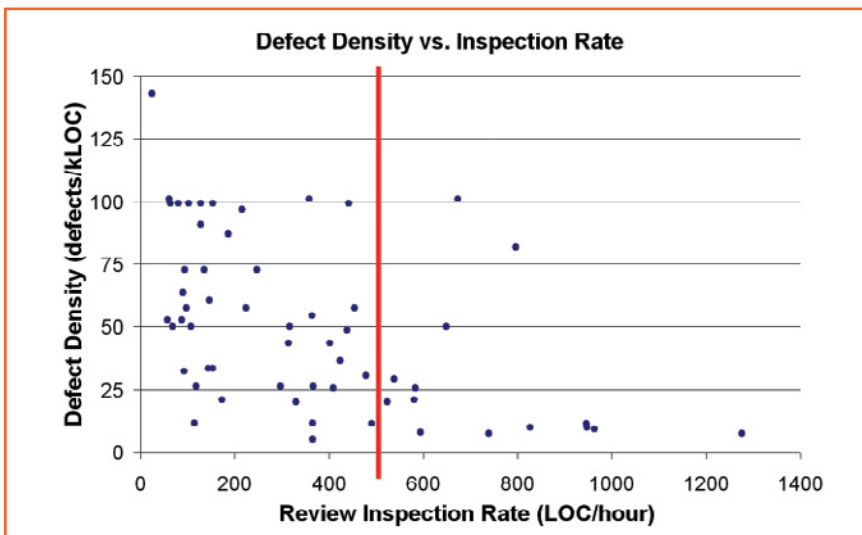


Figure 2: Inspection effectiveness falls off when greater than 500 lines of code are under review.

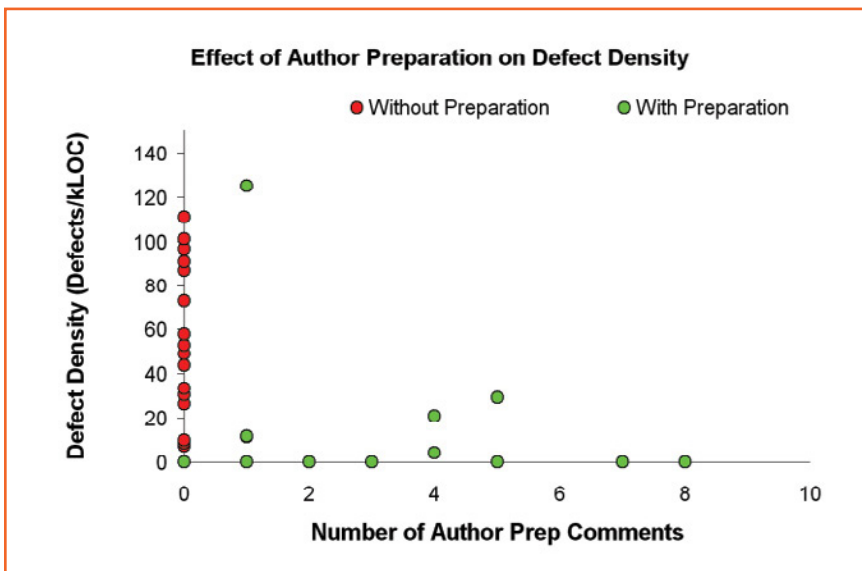


Figure 3: The striking effect of author preparation on defect density.

Code sidebar). For this evaluation, I suggest tool-assisted, over-the-shoulder, or email pass-around. These methods are easiest to conduct and track.

Have everyone do reviews for twenty to thirty minutes per day for that one week and capture two simple metrics: number of bugs found and time spent. Calculate the rate at which your team finds bugs by dividing the total time spent on reviews by the number of defects found. For example, twenty hours spent divided by eighty defects found yields fifteen minutes spent to find one defect.

Now get the team together, discuss the results, and have the group decide if it was worth it.

Even the most crotchety of programmers can suffer through this process for a week, and then you'll have hard data that shows whether code reviews are worthwhile in your organization. If they don't buy in to the value of code reviews at this point, it's hard to force them. If they do agree that code reviews are worthwhile for improving software quality, you'll almost certainly have successful reviews going forward. Now slowly expand the scope of the reviews.

Note that during this week you'll need to take the time to collect metrics like defects found and time spent. But after that first week, if you don't care about tracking this information, then don't. In that case, future code review may be even less effort than it was when you tried it out.

## Get Maximal Results, Spend Minimal Time

"All right," you agree. "Code review is clearly worthwhile, so we'll try it. But how do we get the best results in the least time?"

First, choose a style of code review that is lightweight and easy but tracks things like number of defects found and time spent in review, so you can evaluate your results. Any of the methods listed in the sidebar will work—except formal code reviews, which are not lightweight and take too much time—though some require more manual work to collect metrics. For a good compromise, use an evaluation copy of a commercial peer code review tool or an open source tool. Then,

you can enjoy the benefit of structured guidance without affecting your budget.

Once you've decided on the process you're going to use and which files to review, follow the guidelines below to make the most of your code reviews. They're based on a code review study at Cisco Systems that encompassed fifty programmers, 2,500 code reviews, and 3.2 million lines of code (see the StickyNotes for more information).

- Review no more than 200 to 400 lines of code (LOC) at one sitting. Outside of that range, your ability to find defects plummets, as shown in figure 1.
- Review fewer than 300 to 500 LOC per hour. It stands to reason that if you don't spend much time on the review, you won't find many of the bugs that lurk there. But if you try to review too much code at once, you won't be able to devote enough detailed attention to the code you're reviewing, as shown in figure 2.
- Limit reviews to sixty to ninety minutes. After that point, you get tired, lose focus, and your performance starts to drop off. Note that this number is about the same for other focused activities.
- Have the author annotate the code before the review starts. This practice provides two huge benefits. First (and most obvious), the annotations help the reviewer know which files to look at and explain the reasoning behind methods chosen. The second benefit is even more powerful. Often, as authors are rethinking and explaining their code, they discover their own defects even before the review begins, as shown in figure 3. This kind of self-review is a good idea even if your team doesn't do code reviews!
- Use checklists. They greatly improve results for reviewers and authors, alike. Checklists remind programmers to pay attention to easy-to-overlook tasks, such as proper error checking, unit tests, invalid value checking, and others. Personal checklists also can be a big help. Figure out what

mistakes you typically make and keep a checklist of the five to ten most common ones. Once you stop making those mistakes, update the list to cover whatever mistakes you now perpetrate. See the StickyNotes for more detailed information on how to build checklists.

- Make sure that bugs are fixed. It sounds simple, but since bugs found during review aren't automatically tracked, you need to put a process in place to ensure that they're dealt with.
- Encourage a positive culture around reviews. Make sure everyone understands that finding bugs is a good thing. Each defect found is a bug that gets fixed at the optimal time in the cycle and never reaches the customer. The results of code review (how many bugs each programmer introduces) should never be used as the basis for a programmer's performance evaluation (except in cases where it shows that the person is a strong team player who conducts a lot of code reviews). I'll give specific tips on how to set the right tone for reviews in the next section.
- Always review some code. Even if you don't have time to review all of it, you'll find that programmers become more careful if they know that someone else might be checking their work. They'll check it themselves first, because they want to avoid public embarrassment and want potential reviewers to see the brilliance that is their work. We call this the Ego Effect, and it kicks in if a team reviews at least 20 to 33 percent of a project's code.

## Position Code Reviews for Success: Suggestions for Managers

The most important thing a manager can do to ensure successful code reviews is to set the right tone and help programmers adopt the right attitude toward reviews.

### WHY CODE REVIEW?

- **Improve software quality**—You won't just ship fewer bugs, you'll improve design, structure, documentation, maintainability, testing, and even code comments.
- **Help your team communicate better**—Code reviews get people talking, collaborating, and building trust in addition to better code. Once the "ice is broken," many teams report vast improvement in how they work together.
- **Mentor and teach new team members**—It's a good way to let them work on "real" projects while making sure they don't do any serious damage, and it's a fantastic way for them to learn.
- **Disseminate knowledge throughout the team**—Code reviews deepen understanding about how your code works and spread best practices. It's a great way to make sure that more than one or two people truly understand your code.

Before you kick off the review process, allow team members to air their concerns up front so that the process addresses these issues from the start. Open lines of communication are crucial. It helps if team members already get along, but if they don't, and you do code reviews with the right attitude, you might find they'll start to get along. It also helps if team members have the ability—and feel safe enough—to laugh at themselves and know that others support and respect them even when they do something shockingly bone-headed (which we all do at least once in a while).

You can use these points to set the stage and define the right "code review culture" at the outset:

- **Finding defects is good**—You want to find defects in code reviews. Every bug found now is one that is not found later in test or by customers. It's a bug that's cheaper and easier to fix. Finding bugs in code reviews should be applauded.
- **Code review is about the code, not the author of the code**—It's about giving suggestions to improve the code; it should not be a personal evaluation or an affront to the coder. Don't make it personal.



## FIND BUGS EARLY, SAVE EXPONENTIALLY

When folks catch bugs early, the error can be easily rectified without major rework. Consider that fixing bugs found during test costs about eight to twelve times more to fix than bugs found and fixed when the software is still in development. The tester has to discover an error, document it, and send the code back to development. Then the programmers search until they find the bug, and they start to fix it—which might be many months after the code was written, when the error has affected many other things. To compound the problem, the code is no longer fresh in the programmers' minds, or no one even remembers who wrote that section of code.

Once the software is released into customers' hands, the cost to find and fix bugs rises to thirty to one hundred times what it costs in the development stage (and that's not counting damage to your company's reputation). So, if code reviews truly help you find a bug every ten to fifteen minutes, that's a huge cost savings and trouble avoidance for everyone.

Share this information with your team, and you'll likely get a lot more buy in. Even though most teams barely have time to meet their tight development schedules, code reviews (done right) save time in the long run.

- **Don't tolerate bullying**—Occasionally, a team member uses code review as an opportunity to destroy the self-confidence of another programmer or to try to establish intellectual superiority over all. It's the manager's job to ensure that reviews are conducted in a safe and encouraging environment. If a team member starts to "abuse her power," it's time to have a chat to remind her about the right spirit for code reviews.
- **Work with everyone on the team, not just your buddies**—Team members will learn more if they make the rounds and review everyone's code. This way they can get to know each other and pick up new ideas from more people.
- **Code quality is a team effort**—Code reviews strongly support this goal, and they help build teams. But, if someone on your team isn't following the rules (such as by bullying), it's best not to single her out in front of the team; that approach likely will cause even more trouble. Instead, remind the team as a whole that finding defects is a good thing, and that the number of bugs in someone's code does not correlate to her innate intelligence. After all, harder code and newer code have more bugs, regardless of the author.
- **Inject humor and have fun at every opportunity**—Code review can be the perfect opportunity for team members to joke around and to make light of themselves and their mistakes. If you encourage humor, then everyone becomes more comfortable laughing together—and laughing at themselves. And, they probably won't be offended by review comments or despise the time they spend reviewing. With regular humor and fun, overall morale is likely to improve substantially.

## Position Code Reviews for Success: Tips for Programmers

While it's easy to look down on errors and disdain those who make them, remember that everyone—even you—

makes mistakes. Code review is about recognizing and supporting a phenomenon that is inherently human. To keep the spirits up and attitudes positive, programmers can use these simple guidelines:

- **Always find something good to say**—Better yet, lead with that. In general, offer as much praise as you can.
- **Keep in mind that you're working to improve the code, not to criticize the author**—This mindset helps you frame comments in a more positive, less personal way and helps keep the author focused on the goal of improving code quality, as well.
- **Be patient and respectful with everyone**—Although it can be difficult, this tenet applies especially to colleagues who are less experienced, knowledgeable, or skilled than you. Your encouragement and equanimity will pay off in the end—for you, for them, and for the whole team.
- **Embrace the opportunity to learn from others**—It's not often you get the chance for one-on-one, targeted instruction. You can learn something from everyone, even the greenest member of your team.
- **When disagreements happen, win and lose with grace**—Especially win. You'll foster future allies rather than enemies. Added bonus: You won't look like a jerk.
- **Ask questions rather than making statements**—Asking someone what her logic was when he did something is a lot less offensive than saying, "This is wrong." Especially because it might not be wrong, and her answer might teach you a thing or two. Asking a question sets the stage for an evenhanded discussion rather than a one-sided criticism.
- **Give specific, timely feedback**—It's helpful and time-saving for all if you tell the author exactly where a problem lies (exact line number) and give her enough details to be able to fix the problem in a satisfactory way. Obviously,

once the author has submitted her code to you for review, she's waiting to get it back. Be considerate and don't make her wait too long.

- **Be thankful**—Someone is taking her valuable time to help you produce a better product, share her knowledge, and help your team in general. Appreciation is in order.

## Get Started!

You already know that code review will improve all aspects of your code—from design to structure, maintainability to documentation, testing to general quality—or you wouldn't have read this far. If you're still skeptical about the team dynamics, I'll proffer a few final insights. I've worked with hundreds of teams that implemented code review, and they repeatedly report social benefits that surprised them.

First, everyone's happy and excited about producing a better product and about learning new things. The goodness continues to grow as code quality improves and as team members pick up more and more new tricks from their peers.

And although code reviews may be conducted across oceans, teams find that when they talk to their teammates around a mutual goal, they start to build their relationships. Developers come out of their caves and start to work together, rather than in parallel. Now that they have something to talk about, they have the opportunity to bond with others in person, by email, in meetings, or online. Teams with the right code review culture report that everyone is having more fun and getting along better.

So, what are you waiting for? Try your hand at code review for a week. Social benefits aside, if you do the math, you know you can't afford not to. **{end}**

## Sticky Notes

For more on the following topics go to [www.StickyMinds.com/bettersoftware](http://www.StickyMinds.com/bettersoftware).

- Code review study
- How to build checklists

# The **Seven** Habits of Highly Effective Testing Organizations

A New White Paper by

Testing Expert Lee Copeland

How are you tackling quality issues?

Is your testing organization regarded as trusted advisor – or a perceived bottleneck to getting releases out the door?

**From this white paper you will learn:**

- . How Agile is changing the way we test
- . Why the testing organization no longer needs to be a distinct group
- . The importance of risk-based testing and its main elements
- . Change management's importance for producing quality software

Lee Copeland brings us a look at the habits that we should adopt to take testing to a new level and reap greater value from QA efforts.

Get the free white paper, compliments of MKS:

[www.mks.com/seven\\_habits](http://www.mks.com/seven_habits)

Contact us: 1 800 613 7535 | [info@mks.com](mailto:info@mks.com)

**MKS**



# Virtual Taskboards

## for Agile Teams

- Real-time collaboration
- Instant updates

Sign up free at  
**SeeNowDo.com**





## TestDrive-Assist

WESTMONT, IL—Original Software has announced a strategic partnership with AppLabs. AppLabs has entered into an agreement with Original Software to take to market its innovative quality management suite, TestDrive-Assist. The agreement likewise will allow Original Software to position AppLabs' services into its own customer base.

Some comments from a recent Ovum report include: "TestDrive-Assist provides you with a rich environment for manual testing, but the bigger payback is how it facilitates adoption of full automation. Uniquely for any of the guided manual test execution products that we know of, Original can automatically convert a manual TestDrive-Assist test into a fully automated TestDrive test that can be repeated."

"Overall this is a low risk, low cost way of progressively adopting test automation."

"It fits in with classic waterfall processes, and it can also be used in agile processes more easily than other functional test automation suites."

Visit [www.applabs.com](http://www.applabs.com) for additional information.

## SCM System 2009.1

ALAMEDA, CA—Perforce Software has announced the availability of release 2009.1 of its software configuration management system. The latest version features support for previewing Web pages and multimedia files as well as the ability to customize the cross-platform interface to individual requirements.

With this release, Web developers and artists can immediately preview changes to Web, video, and audio content from within Perforce's cross-platform graphical interface, the Perforce Visual Client. A preview tab in the Perforce Visual Client automatically displays the version of the content selected by the user.

Visit [www.perforce.com](http://www.perforce.com) for information.

## TotalView 8.7 and ReplayEngine 1.5

NATICK, MA—TotalView Technologies has announced the release of TotalView 8.7 and ReplayEngine 1.5. In addition to its support of a wide variety of UNIX, Linux, and Mac OS X environments,

TotalView now supports selected cross and heterogeneous debugging configurations that incorporate the Cell processor and 32-bit PowerPC processors.

Both products are interoperable with MemoryScape, TotalView Technologies' memory analysis tool. TotalView users can activate MemoryScape from within TotalView to learn more about the memory status of variables and to perform leak detection and overrun checking. ReplayEngine, when used in conjunction with MemoryScape and TotalView, allows users to examine the conditions that lead up to any memory fault.

TotalView 8.7 introduces support for heterogeneous debugging across several Linux platforms, including the Cell and 64-bit Power processors, while also adding cross-debugging capabilities to 32-bit PowerPC devices and improved support for network configurations that eschew the use of the DNS name resolution system.

ReplayEngine, a TotalView add-on, records program execution history and replays it for diagnosis with TotalView. With the release of ReplayEngine 1.5, users can record and replay just the last few minutes of long-running programs. Users control the amount of memory resources they are willing to devote to storing recorded history. ReplayEngine manages memory to allow replay of the most recent—and likely most relevant—part of the execution history of long-running applications. MPI support within ReplayEngine has been enhanced with support for MPICH 1 added to MPICH 2, Open MPI and others.

Visit [www.totalviewtech.com](http://www.totalviewtech.com) to learn more.

## TeamSupport.com and Beanstalk Partnership

DALLAS, TX—TeamSupport.com has announced integration between TeamSupport and Beanstalk. Beanstalk is used by software developers to manage versions and source code changes.

Easily configured and customized, TeamSupport is offered in several reasonably priced versions and is scalable from a simple help desk system to an enterprise-wide issue, bug, feature, and customer management system.

Visit [TeamSupport.com](http://TeamSupport.com) to learn more.

## Adaptive ALM Connector

CHICAGO, IL—ThoughtWorks Studios and Tasktop Technologies has released the ThoughtWorks Adaptive ALM Connector. The connector will help streamline software development by providing direct access to ThoughtWorks Studios' products from within the Eclipse IDE.

The ThoughtWorks Adaptive ALM Connector integrates Mylyn's task-focused productivity technology to provide development teams with the most relevant source code for a given task. It will help reduce information overload for large-scale development projects and facilitate more efficient and collaborative multitasking. With the connector, ThoughtWorks Studios' customers will improve productivity by eliminating the need to toggle between the browser, email, and IDE to gather information necessary to complete application development tasks.

Visit [www.tasktop.com](http://www.tasktop.com) for additional information.

## LISA Virtualize

DALLAS, TX—ITKO has signed an agreement for HP to resell iTKO's LISA Virtualize product. With this agreement, customers can combine iTKO's innovative virtualization software product with HP's market share-leading quality management, functional, and performance testing solutions to accelerate testing efforts and reduce costs associated with the delivery of modern applications.

The addition of iTKO's LISA Virtualize product to HP's quality and performance management solutions can help accelerate the application lifecycle by eliminating costly and common system and infrastructure dependencies during application testing. The ability to reduce testing constraints caused by dependent IT resources that are unavailable or inaccessible for testing will have a measurable impact on reducing the cost and risk of modern quality assurance.

System dependencies and constraints can be problematic for testing, especially in highly distributed, interdependent environments. Downstream application components, packaged applications, integration infrastructure, or data sources may be unavailable or poorly performing and unusable. ITKO's LISA Virtualize

# More time... Less stress...

*eLearning is the perfect solution for training at your own pace.*



**Special  
eLearning Offer!**  
**Save \$150 off  
either eLearning  
course.**  
Register by 12/31  
using promo  
code: **BSEL**

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

Travel from your desktop with SQE Training's eLearning courses, eSoftware Tester Certification and eMastering Test Design. Experience classroom value with the convenience of self-paced instruction on the Web. Because people learn in different ways and everyone learns best with multiple learning options, we've combined audio, video, text, graphics, examples, questions, exercises, and additional learning resources for the best Web-based delivery possible.

### eSoftware Tester Certification— Foundation Level

Train to be an ISTQB Certified Software Tester from your desktop! Through the eSoftware Tester Certification—Foundation Level training course, learn the basics needed to become a software test professional and understand how testing fits into software development. Find out what it takes to be a successful software test engineer and how testing can add significant value to software development.

### eMastering Test Design: The Art and Science of Creating Test Cases

eMastering Test Design teaches you to select an optimal set of what to test and develops your practical skills to become a better test engineer. This course begins where many software testing courses end. Once the test plans are written, test teams are formed, and test tools are selected, it is time to create test cases. Since testing everything is impossible, the first step in test design is to choose a subset of all possible tests of program paths and data combinations to find important defects quickly.



[www.sqetraining.com/eLearning](http://www.sqetraining.com/eLearning)

## What are the benefits of eLearning?

- Same valuable information as the classroom courses
- Superior lesson content developed and delivered by testing experts
- 90 Days access to all course materials: Learners have unlimited access to online content
- Access to expert test consultants and administrators: Email questions and comments to testing experts
- Powerful multi-media format: Students experience professionally narrated audio and video clips
- Interactive exercises: To reinforce new skills and concepts



eliminates system dependency constraints by “virtualizing” or simulating the dynamic behavior and performance conditions of downstream system dependencies so that they react and respond the same as the live system. With virtualization, functional and performance testing can continue without the delay of acquiring and accessing these frequently unavailable or cost-prohibitive dependent resources.

Visit [www.itko.com/virtualize](http://www.itko.com/virtualize) for more information.

### **TWIST 1.1 AND CRUISE 1.3.2**

ThoughtWorks Studios has released updates to Twist and Cruise, with new features and a host of performance and usability enhancements.

With Release 1.1, Twist is your team’s single, central space for all your manual and automated testing. You can now just as easily use Twist to create, execute, and maintain all your manual test scenarios as you do with your automated tests. In addition, Twist 1.1 allows you to debug failing automated scenarios using manual execution and

record manual test results.

Highlights:

- A central place to manage all your automated and manual tests. Additionally, Twist’s feature-rich IDE enables you to evolve your manual tests as your application changes.
- Execute and record results for manual tests. Twist also supports batch execution of all manual tests in a test suite.
- Record assertions from the Web browser. Twist Selenium Recorder now also supports asserts for text, window title, URL, etc.

Cruise 1.3.2 simplifies the release management experience with usability and performance improvements. It provides the ability to more easily correlate builds with the version in source control from which they are derived. Cruise 1.3.2 also eases maintenance efforts by monitoring and warning teams of low disk space for agents or pipelines.

Visit [www.thoughtworks.com/studios](http://www.thoughtworks.com/studios) for more information.

### **Test Case 1.1**

NEW YORK, NY—Elementool Inc. has announced the release of Test Case version 1.1, the newest tool in its suite of software-as-a-service (SaaS) project management applications. Test Case 1.1 allows testing managers to customize the tool in order best to organize, control, and oversee the testing process.

New features include:

- File attachments: The ability to attach files to test cases allows for further clarification of instructions, functions, and steps in the test case and individual tests.
- Remarks: A new remarks message board lets team members share information quickly and access results or questions from previous tests.
- Customizable test form: Enables the administrator or test case manager to change and add fields to the test form, altering the form for different projects or teams. Account administrators can now clear the history in any field.
- Report customizations: Users can

# Why does software collapse but bridges don't?

Today, software is your bridge to customers and partners. You can’t afford to let it collapse. At Infosys, we have an engineering approach to IT. We build software as if we are building bridges, with the same rigorous approach to process, quality, and testing; from requirement gathering to final deployment. To know more, visit [www.infosys.com/testing-services](http://www.infosys.com/testing-services)

**Infosys®**

POWERED BY INTELLECT  
DRIVEN BY VALUES

• Consulting • IT Services • BPO • Product Engineering

[www.infosys.com/testing-services](http://www.infosys.com/testing-services)

Infosys® is a registered trademark of Infosys Technologies Ltd. Infosys acknowledges the proprietary right in the trademarks and product names of other companies mentioned in this document. The trademarks are being used without permission and the publication of the trademarks is not authorized by, associated with or sponsored by the owners of the relevant trademarks.



now customize the test report and choose filters and report columns to obtain only the specific data needed at that moment.

- Sub groups: The option to create sub groups within individual tests adds further clarification to the process.

Using Test Case 1.1, as with the original, project managers can assign test cases to team members, track test progress, activate email notifications to individual users regarding assignments, and generate test case reports.

As with all Elementool SaaS tools, Test Case 1.1 can be accessed from any computer via Web browser without downloading or installing any software. Visit [www.Elementool.com](http://www.Elementool.com) for more information.

### uTest 2.6

BOSTON, MA—uTest has launched version 2.6 of its platform, which includes a dramatically improved tester profile. This new version will enable uTest to more precisely match customers' Web,

desktop, and mobile app testing needs with the right members of its global QA community. The uTester community also benefits from this update, as testers can more accurately define for which types of projects they are best suited.

After capturing basic demographic information (location, age, gender, languages, experience), the new profile enables testers to define their hardware and software, experience, and skills.

Visit [www.utest.com](http://www.utest.com) for more information.

### PureCM 2009-2

PureCM has announced the latest version of its software configuration management solution, PureCM 2009-2. This major release offers much tighter integration into the development environment of software development teams, boosting productivity and facilitating collaboration.

Developers can now use the enhanced Visual Studio integration to perform task-based development within their IDE. Features originally available in PureCM only are now available di-

rectly within Visual Studio, providing developers with intuitive control about their changes and complete transparency about project progress.

The enhanced Visual Studio integration dramatically improves ease of use and productivity along the whole developer workflow. Developers can switch to project branches without rebinding solutions and group their current changes within their IDE to reflect different tasks. Collaboration is facilitated as features like shelving and unshelving changes for code review or integrating changes from other branches are now directly available within Visual Studio.

The 2009-2 release marks another substantial improvement with up to 400 percent faster submit performance when compared to its predecessor. In combination with highly efficient delta transfer and data compression, PureCM 2009-2 has become one of the fastest SCM solutions on the market.

Visit [www.purecm.com](http://www.purecm.com) for more information.

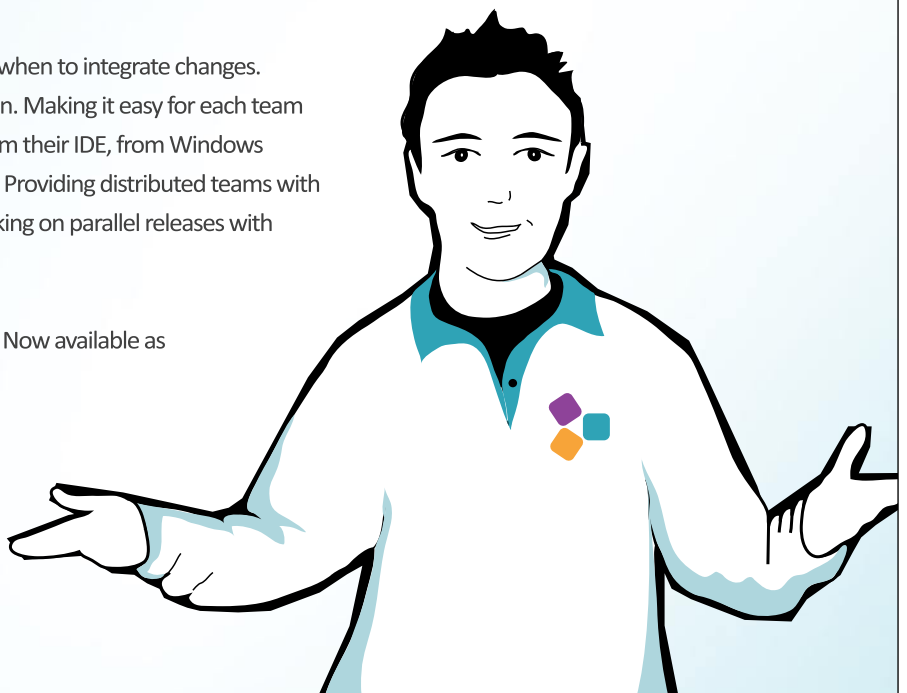
## Source control can be much more than just checkout and checkin.

Get your **FREE** trial now at [purecm.com](http://purecm.com)

Being able to respond to change by deciding when to integrate changes. Improving quality with Continuous Integration. Making it easy for each team member to access the same project data: from their IDE, from Windows Explorer or from their Mac or Linux machine. Providing distributed teams with secure but fast access and having teams working on parallel releases with minimum overhead.

**PureCM – SCM that grows with your needs.** Now available as Standard and Professional edition.

 **PureCM** [purecm.com](http://purecm.com)





## Bug Tracking

Track bugs and other issues to create and maintain high-quality software

## Requirements

Collect and define features of your software and track changes from various sources

## Writing Test Cases

Document your testing coverage to maintain consistency between releases

## Task Tracking

Track individual or project-wide tasks with private and shared "To Do" lists

## Scheduling

Stay on schedule and watch progress with simple-to-use project management features

Hosted Bug Tracking, Requirements Documentation and Project Management System

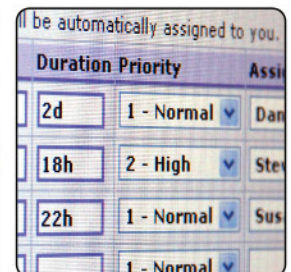
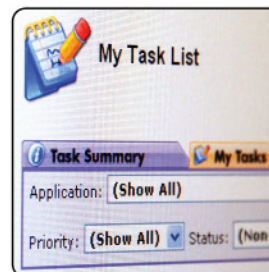
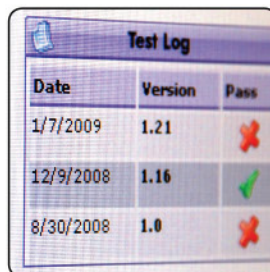
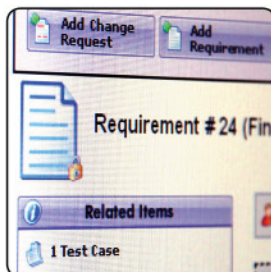
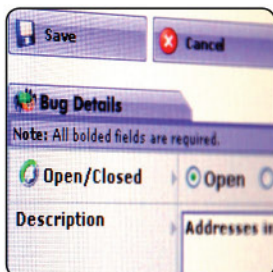
# management

Bug Tracking  
Requirements  
Writing Test Cases  
Task Tracking  
Scheduling



just got **easier.**

[www.BugHost.com](http://www.BugHost.com)



# Things You Might Not Know About

## Continuous Integration

by Jeffery Payne

- 1 **THE NAME SAYS IT ALL.** Continuous integration (CI) is the process of continually integrating your software to assure that any software issues are eliminated as early as possible during software development. Effective CI heavily leverages automation.
- 2 **CI BUILDS QUALITY IN.** CI not only automates your software build process but it also is the building block for your testing process. Done right, CI moves software testing earlier in the software development process and dramatically increases your testing capability.
- 3 **AGILE WON'T WORK WITHOUT CI.** Agile development is all about reducing the cost of change for your requirements, design, and code. Rapid, iterative development only works when integration and quality issues are dealt with as soon as they arise. CI assures that happens.
- 4 **CI ISN'T JUST FOR AGILE.** Successful integration of software is critical to any software development project. Too often, integration of code across team members or separate development teams is done late in the lifecycle, when the cost to correct issues is significant. Even if you are not doing two-week development cycles, use CI to more frequently build and test what has been developed to date.
- 5 **OPEN SOURCE TOOLS ARE MATURE.** The difference between open source and commercial tools is often substantial. Not so for continuous integration. There are many very mature open source solutions that can significantly reduce your costs of leveraging CI.
- 6 **CI HOLDS TEAMS ACCOUNTABLE FOR QUALITY.** CI systems can be configured to notify the development and testing teams when a build or set of tests fail. This makes visible to everyone on the team when there is a problem and who is responsible for correcting it. Team members will hold each other (and themselves) accountable when it's clear where software issues are coming from.
- 7 **CI HELPS SECURE APPLICATIONS.** Integrating automated secure code analysis and security testing into CI provides an easy mechanism to start getting development and testing teams to understand and address security issues. Make sure you provide some security expertise to these teams to help them review security results and figure out how to remediate any vulnerabilities.
- 8 **CI INCREASES PROJECT VISIBILITY.** CI done right generates a lot of information on the progress being made toward a successful release. Savvy project managers use this information to keep their business sponsors informed on project progress. You will find that this significantly reduces the effort of managing your business customer.
- 9 **CI HELPS INTEGRATE DEVELOPMENT AND TEST TEAMS.** CI is the perfect integration point for development and testing teams during development. Have your test lead involved in the setting of pass/fail criteria not only for his testing activities but also for the unit tests developed by software engineers. You will also find that your test teams can help developers significantly improve their unit testing skills.
- 10 **CI WORKS ACROSS PROJECTS.** Large-scale programs comprising many interconnected projects greatly benefit from CI. Develop an approach that assures your entire program goes through a frequent integration process, and you will see a significant reduction in defects.



# QA Is Not Evil

by Chris McMahon

Many people in this profession prefer to be called “software testers” rather than “software quality assurance” (QA) workers. They point out that quality can only be assured by the entire development team, and that software testing is rightly defined as quality control (QC). QA work is process work, but software development has historically lumped QC and QA together. On a lot of projects this means that the traditional QA role is set up for failure, being the ones on the project whose responsibility it is to criticize the work only after the work is done. As Antony Marcano once said, “Unfortunately, in the software industry, all too many teams don’t realize their *process* doesn’t work until the testers find all the ways in which the *product* doesn’t work ... maybe that’s why software testing has come to be known as QA.”

There are important differences between the work of testing software and the work of QA. But, there are important reasons why testers are in a good position to exercise excellent quality assurance.

## Definitions and Confusions

Software testing is the act of investigating the behavior of software products in order to supply information to the stakeholders about the fitness of the software for use in particular context. Software testing is not a phase in the development lifecycle.

QA is analysis of the software development process for the purpose of increasing or maintaining the quality of the software. QA is all about process, of which testing is only a small part.

In the world of manufacturing, for a product to roll off an assembly line, measurements must be taken (testing or QC), and if parts do not conform to spec, the reason must be found (process analysis or QA).

But we are discovering that analogies from manufacturing often map poorly to

software development practice. Ten years ago, implementing ISO9000 processes and Six Sigma processes for software development was very popular. Today, many software development organizations see such approaches as expensive and wasteful. Software quality assurance for modern development teams demands a more subtle approach, and testers can be a big part of that work.

## Key Aspects of Testing

Good software testing shares some focus with software development. Both require close reading of the software requirements. Both are closely involved with the code as it comes to implement the requirements. Both are closely concerned with avoiding software failure in the production environment.

But good software testing deviates from development in a few important ways. Developers typically work in a very limited area of the code base for long stretches of time, while testers are very concerned with the behavior of the system as a whole. Developers are deeply concerned with the technology of the software, while testers tend to be more concerned with the business implications of the software. The tester typically is a proxy for the eventual consumer of the software product and works accordingly.

This means that testers often are the only role on the project team to have deep and close exposure to every aspect of the software—from conception to development to deployment to ultimate use.

## Learning QA

But good testing does not automati-

“Just because testers are familiar with the entire process does not mean they are in a position to improve it—or even influence it at all.”

cally mean good QA. Just because testers are familiar with the entire process does not mean they are in a position to improve it—or even influence it at all. That work takes an entirely different set of skills and a whole new area of knowledge.

For one thing, it helps to have a solid grasp of the history of software development processes, what problems each process was intended to address, and what problems each process introduced. A

brief list of well-known software development processes includes: staged (waterfall) processes, with gated handoff requirements; V-models, with work going on simultaneously by all participants; spiral models; and iterative models. It is also helpful to know about general subjects like requirements management, code design, validation, verification, deployment strategies, and other aspects of delivering software.

There is nothing to stop a tester from learning these things. And a tester, in the course of his work and in the course of his career, will be able to learn and practice bits of what he reads about these processes.

## Gentle QA

Changing process is hard. Changing process quickly can be devastating. A good QA person will look for “inflection points,” that is, opportunities where a series of small changes can bring about higher-quality work, higher-quality products, and higher-quality process.

Some processes have such inflection points built in. For example, the agile Scrum process demands a retrospective at the end of each iteration, where the

whole team analyzes the work of the previous iteration, identifies issues that caused problems for the team, and commits itself to fixing those problems. The cumulative effect of such frequent process improvements becomes dramatic over time.

A QA person is in a position to take advantage of practices like retrospectives to guide the team's work along lines that are known to be effective.

## The Quality Police

Bret Pettichord's excellent and widely quoted essay "Don't Be the Quality Police" is worth reading. Pettichord argues that attempting to punish poor performance in the name of quality is bound to backfire.

QA practitioners—whether or not they are also testers—do not manage the project, nor do they have power to hire or fire employees, nor do they have control over budgets. QA is not a position of power.

But QA can be a position of great influence. The QA worker can help the team instead of holding it back. The QA worker can streamline the process instead of being a drag on it. The QA worker can recommend great new information instead of hoarding secrets. Done well, QA can be a valuable member of any kind of software development team.

## Use Your Power for Good

QA in software has a bad reputation because of its poorly conceived conflation with QC and because of its misguided attempts at policing projects in the name of quality. But slowly, advocates of good process, good communication, and good practice are turning that reputation around. Many of these advocates are testers or have a background in testing. Seek them out, and join in the conversation yourself. **{end}**

## 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 your query or submission to

[editors@bettersoftware.com](mailto:editors@bettersoftware.com).

## Index to Advertisers

ADP West 2010	<a href="http://www.sqe.com/adpwest">www.sqe.com/adpwest</a>	10
Avnet	<a href="http://www.avnet.com">www.avnet.com</a>	9
Better Software Conference	<a href="http://www.sqe.com/BSC">www.sqe.com/BSC</a>	10
BigVisible	<a href="http://www.bigvisible.com">www.bigvisible.com</a>	36
BugHost	<a href="http://www.BugHost.com">www.BugHost.com</a>	41
Cognizant	<a href="http://www.cognizant.com">www.cognizant.com</a>	2
Hewlett-Packard	<a href="http://www.hp.com/go/software">www.hp.com/go/software</a>	Back Cover
Infosys	<a href="http://www.infosys.com/testing-services">www.infosys.com/testing-services</a>	39
MKS	<a href="http://www.mks.com">www.mks.com</a>	35
PureCM	<a href="http://www.purecm.com">www.purecm.com</a>	40
Rally Software	<a href="http://www.rallydev.com/bsm">www.rallydev.com/bsm</a>	Inside Front Cover
Seapine	<a href="http://www.seapine.com">www.seapine.com</a>	1
Serena	<a href="http://www.serena.com">www.serena.com</a>	Inside Back Cover
SQE Training—eLearning	<a href="http://www.sqetraining.com/eLearning/Default.aspx">www.sqetraining.com/eLearning/Default.aspx</a>	38
STAREAST 2010	<a href="http://www.sqe.com/STAREAST">www.sqe.com/STAREAST</a>	6
TechExcel	<a href="http://www.techexcel.com">www.techexcel.com</a>	5
Wind River	<a href="http://www.windriver.com">www.windriver.com</a>	23

### Display Advertising

[advertisingsales@sqe.com](mailto:advertisingsales@sqe.com)

### All Other Inquiries

[info@bettersoftware.com](mailto: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 \$40.00 per year. A US \$35 shipping charge is incurred for all non-US addresses. Payments to Software Quality Engineering must be made in US funds drawn from a US bank. For more information, contact [info@bettersoftware.com](mailto: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](mailto:info@bettersoftware.com).

# Agile Development

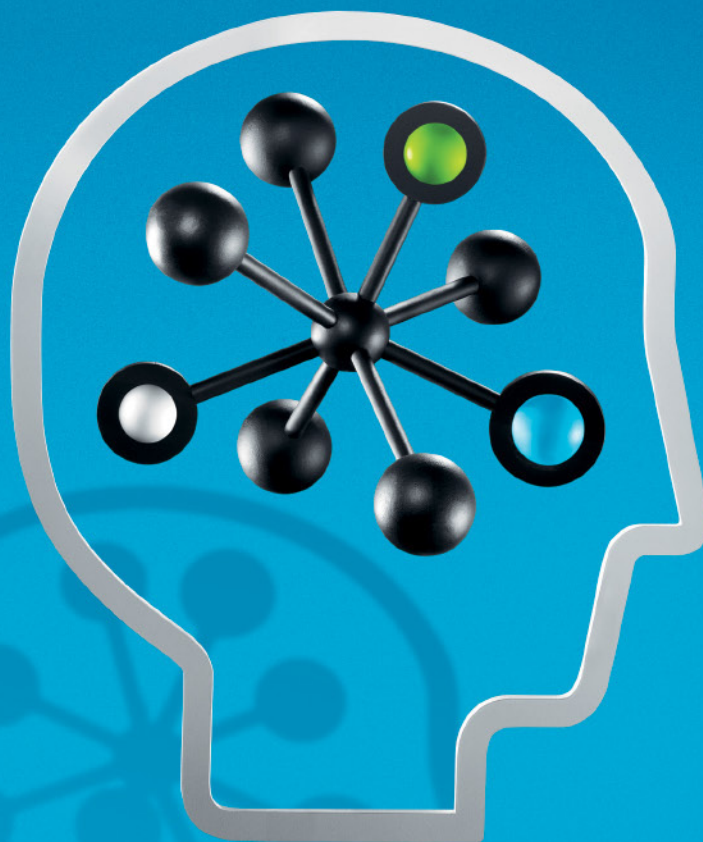


## Agile. We are! Are you?

Agile is here to stay. But how do you make the transition, and adopt it alongside your existing methodologies? Look no further. Our tools make adopting Agile practices easier, faster, and lower cost regardless of size and complexity of your projects. How can we be so sure? We understand, better than anyone else, how to build application development tools. We've been doing it for over 28 years. What's more, our developers co-invented Scrum. Who better to work with in your move to Agile?

[www.serena.com/agile](http://www.serena.com/agile)





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](http://hp.com/go/alm)



Better Software magazine  
StickyMinds.com

2009

# SALARY SURVEY

BY HEATHER SHANHOLTZER

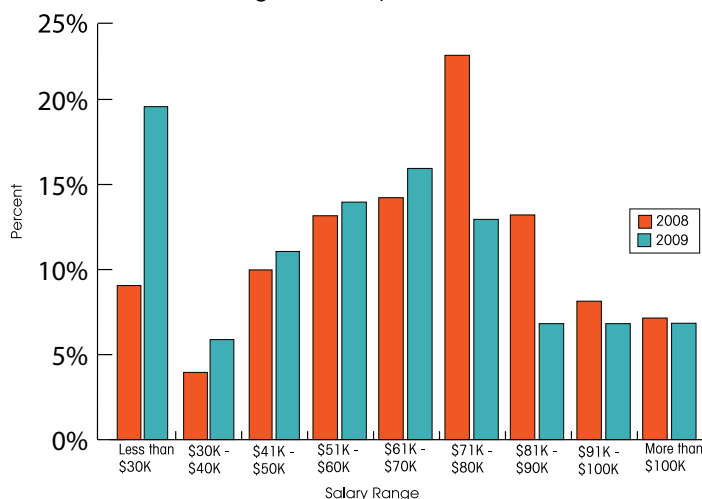
The results of our annual look at industry employment trends are in. This year's salary survey reports salary figures for director-, management-, and staff-level professionals across a wide variety of fields. Our survey included questions not only about salary but also demographics, education, and experience.



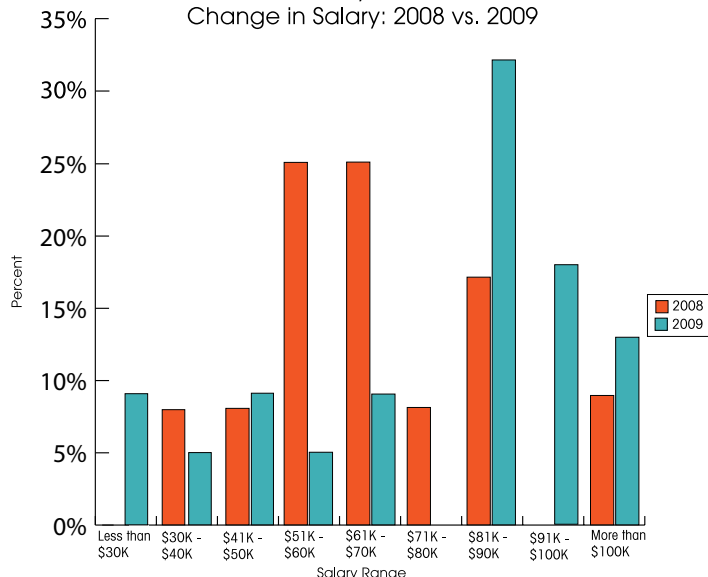
ISTOCKPHOTO

# STAFF LEVEL

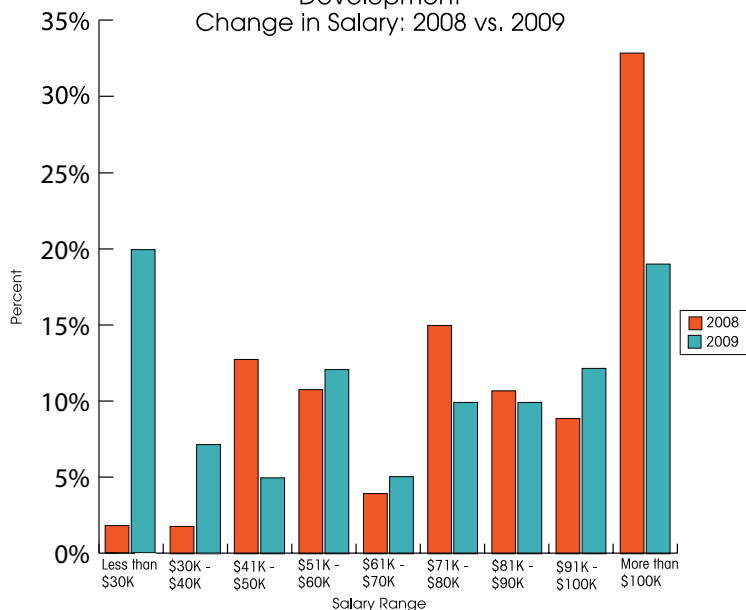
Test/QA  
Change in Salary: 2008 vs. 2009



IS/IT  
Change in Salary: 2008 vs. 2009



Development  
Change in Salary: 2008 vs. 2009



## GENDER

Male	57%
Female	43%

## AGE

40 or under	56%
Older than 40	44%

## EDUCATION

Some college	22%
Bachelors	55%
Masters or higher	23%

## DEGREE

CS/IS	36%
Engineering	20%
Business	14%
Liberal arts	8%
Other	22%

## GEOGRAPHIC DISTRIBUTION

Northeast	17%
Southeast	10%
Midwest	19%
Northwest	5%
Southwest	9%
Canada	7%
India	6%
UK/Ireland	3%
Other	24%

## YEARS IN SOFTWARE INDUSTRY

5 years or fewer	32%
6 to 10 years	21%
More than 10 years	47%

## YEARS AT PRESENT COMPANY

5 years or fewer	65%
6 to 10 years	16%
More than 10 years	19%

## YEARS IN CURRENT POSITION

5 years or fewer	80%
6 to 10 years	13%
More than 10 years	7%

## JOB FUNCTION

Test/QA	80%
Development	12%
IS/IT	6%
Other	2%

## CERTIFICATION STATUS

Yes	41%
No	59%



## GENDER

Male	58%
Female	42%

## AGE

40 or under	43%
Older than 40	57%

## EDUCATION

Some college	20%
Bachelors	47%
Masters or higher	33%

## DEGREE

CS/IS	27%
Engineering	14%
Business	21%
Liberal arts	10%
Other	28%

## GEOGRAPHIC DISTRIBUTION

Northeast	17%
Southeast	9%
Midwest	20%
Northwest	5%
Southwest	14%
Canada	7%
India	7%
UK/Ireland	3%
Other	18%

## YEARS IN SOFTWARE INDUSTRY

5 years or fewer	6%
6 to 10 years	21%
More than 10 years	73%

## YEARS AT PRESENT COMPANY

5 years or fewer	61%
6 to 10 years	19%
More than 10 years	20%

## YEARS IN CURRENT POSITION

5 years or fewer	84%
6 to 10 years	12%
More than 10 years	4%

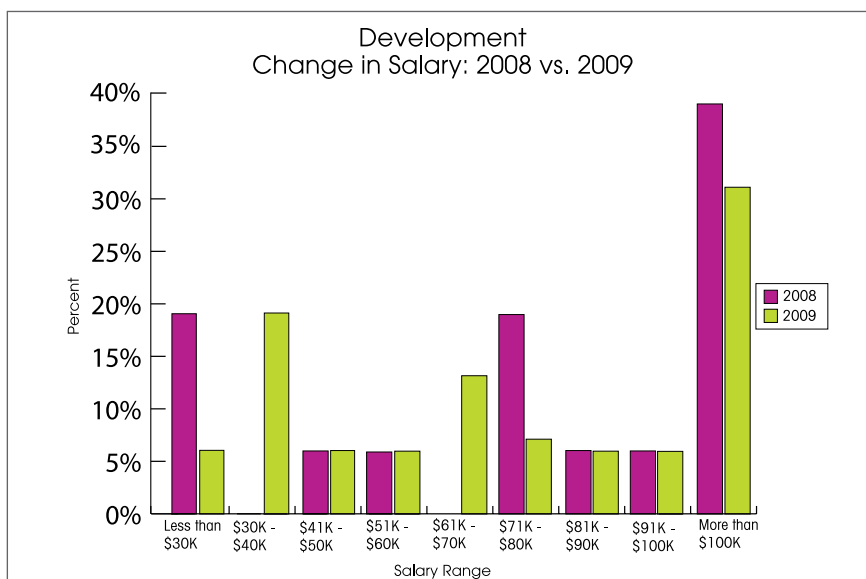
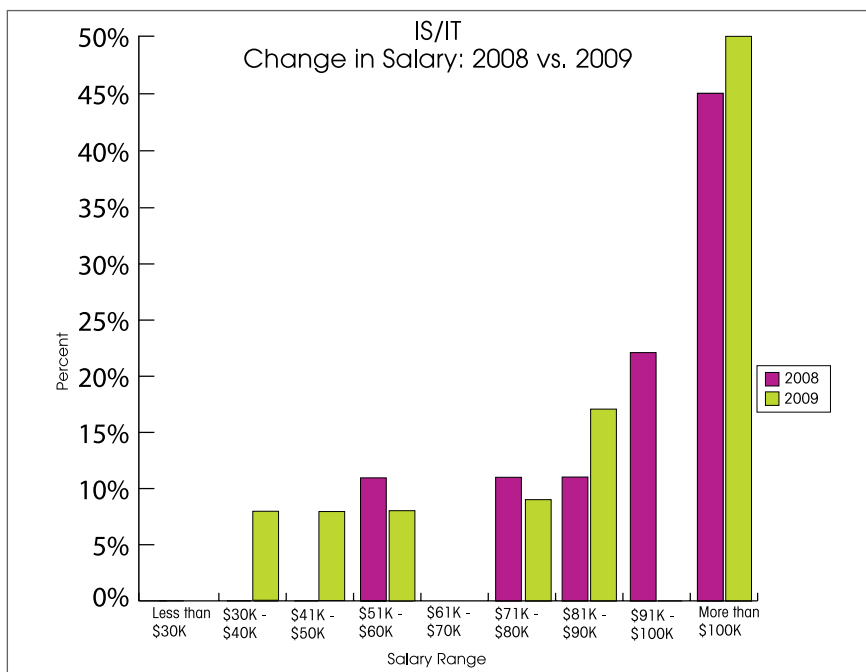
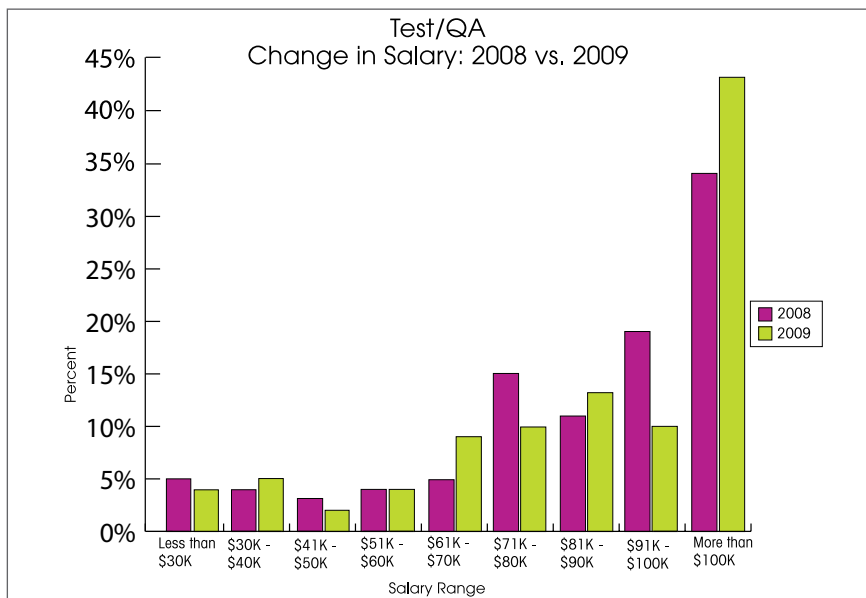
## JOB FUNCTION

Test/QA	76%
Development	10%
IS/IT	7%
Other	7%

## CERTIFICATION STATUS

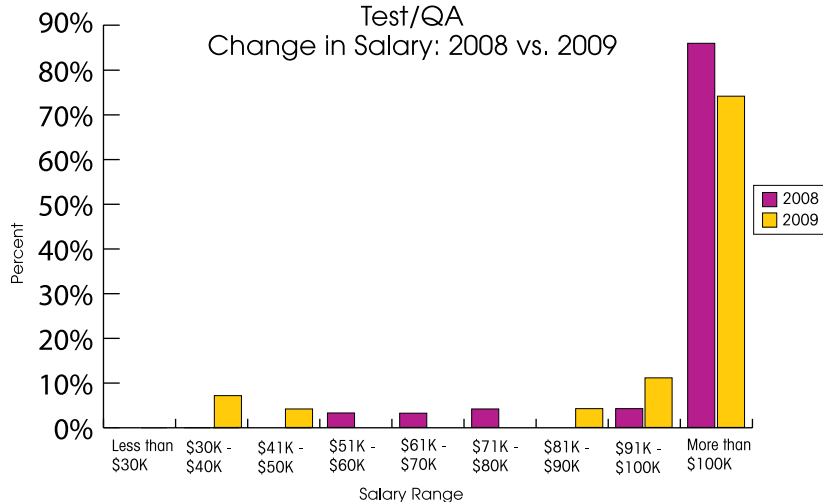
Yes	48%
No	52%

# MANAGEMENT LEVEL

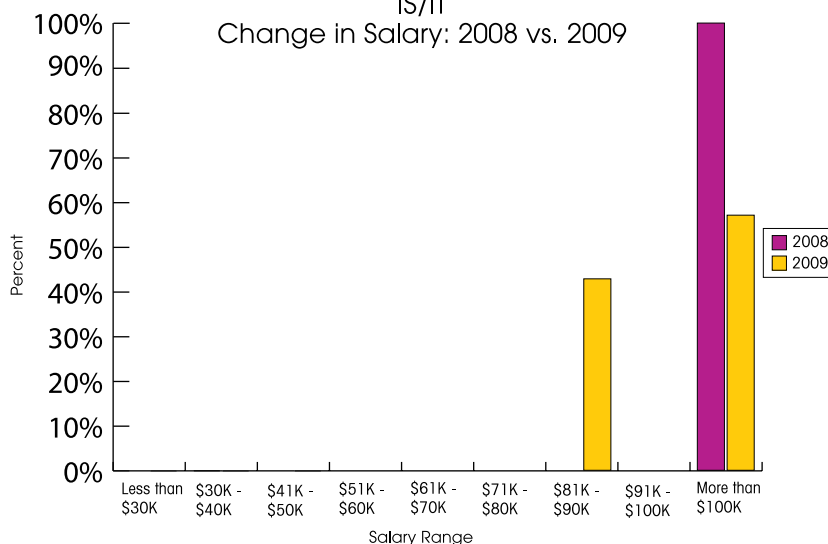


# DIRECTOR LEVEL

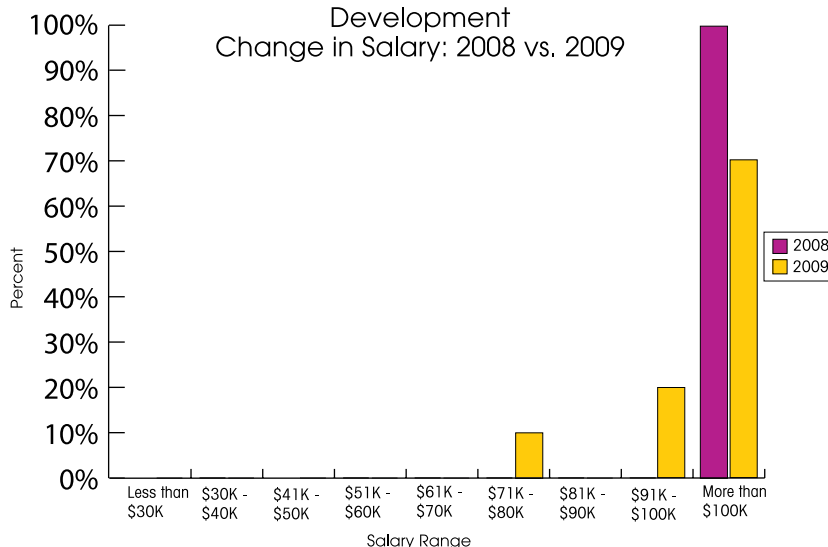
Test/QA  
Change in Salary: 2008 vs. 2009



IS/IT  
Change in Salary: 2008 vs. 2009



Development  
Change in Salary: 2008 vs. 2009



## GENDER

Male	65%
Female	35%

## AGE

Under 40	21%
Older than 40	71%

## EDUCATION

Some college	21%
Bachelors	48%
Masters or higher	31%

## DEGREE

CS/IS	37%
Engineering	32%
Business	15%
Liberal arts	5%
Other	11%

## GEOGRAPHIC DISTRIBUTION

Northeast	33%
Southeast	4%
Midwest	19%
Northwest	13%
Southwest	13%
Canada	6%
India	2%
UK/Ireland	2%
Other	8%

## YEARS IN SOFTWARE INDUSTRY

5 years or fewer	6%
6 to 10 years	0%
More than 10 years	94%

## YEARS AT PRESENT COMPANY

5 years or fewer	43%
6 to 10 years	21%
More than 10 years	36%

## YEARS IN CURRENT POSITION

5 years or fewer	77%
6 to 10 years	13%
More than 10 years	10%

## JOB FUNCTION

Test/QA	54%
Development	20%
IS/IT	14%
Other	12%

## CERTIFICATION STATUS

Yes	36%
No	64%