July/August 2011 $9.95 www.StickyMinds.com

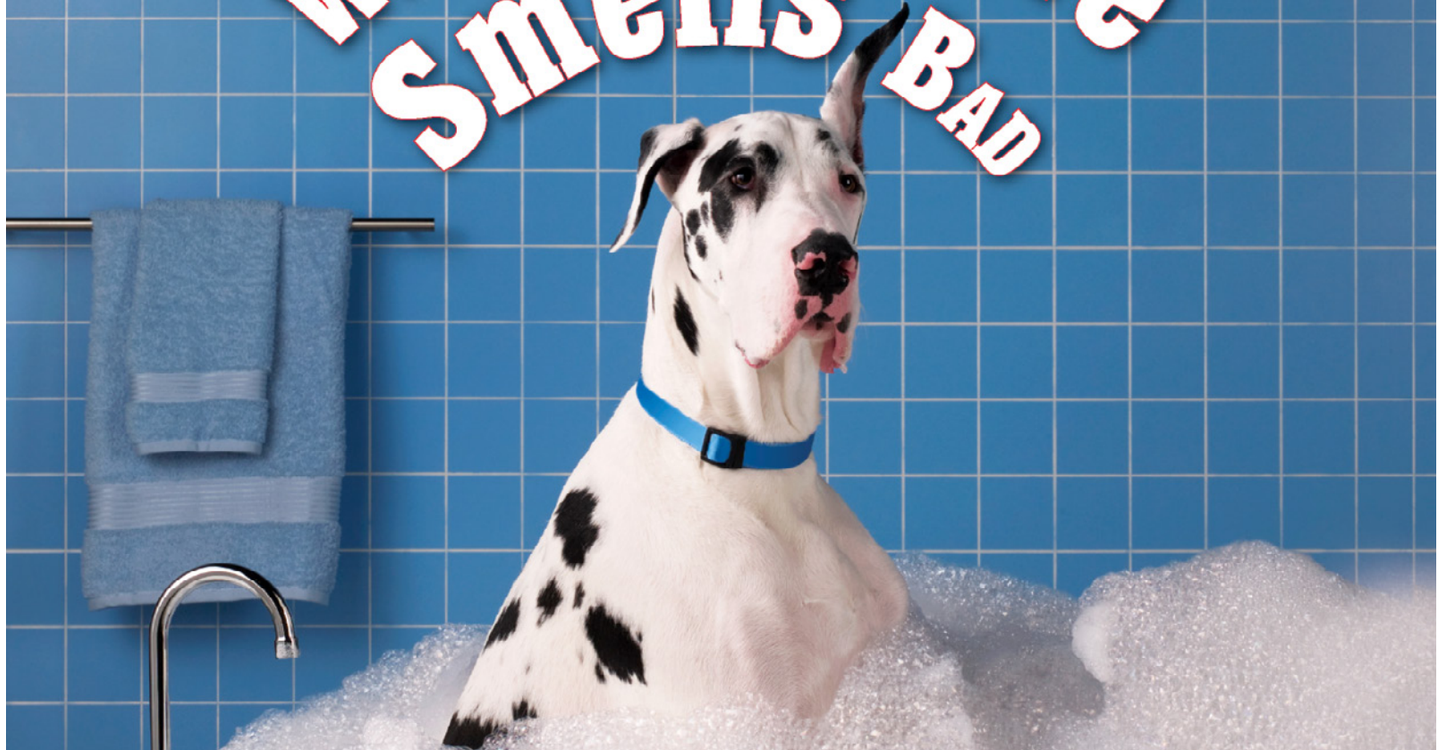
BETTER SOFTWARE™

TECHWELL™ PUBLICATION

IT'S ALL IN YOUR HEAD
Creating software with personas

BOUNTIFUL HARVEST
Reaping an organized backlog

When Software Smells Bad



BETTER SOFTWARE™
A TECHWELL PUBLICATION
Volume 13, Issue 4 • July/August 2011

CONTENTS

features

BETTER SOFTWARE™
A TECHWELL PUBLICATION

Publisher
Software Quality Engineering, Inc.

President/CEO
Drew Thoeni

Vice President of Publishing
Holly N. Bourquin

Editor in Chief
Heather Shanholtzer

Editorial

Managing Technical Editor
Lee Copeland

Online Editor
Joseph McAllister

Production Coordinator
Cheryl M. Burke

Design

Creative Director
Catherine J. Clinger

Advertising

Director of Sales
Sonia Lavin

Sales Consultants
Daryll Paiva
Kim Trott

Customer Success Manager
April Evans

Circulation and Marketing

Product Marketing Manager
Diara A. Sullivan

columns

in every issue

MARK YOUR CALENDAR

SQE TRAINING
www.sqetraining.com

software tester certification

www.sqetraining.com/certification

August 23–25, 2011
Boston, MA
San Jose, CA

August 30–September 1, 2011
Charlotte, NC

September 13–15, 2011
Irvine/Los Angeles Area, CA
Minneapolis, MN

September 19–21, 2011
Washington, DC

September 20–22, 2011
Atlanta, GA
Toronto, ON

September 27–29, 2011
St. Louis, MO
Pittsburgh, PA

training weeks

www.sqetraining.com/testing

Software Testing Training Weeks

September 19–23, 2011
Washington, DC

October 17–21, 2011
San Francisco, CA

November 14–18, 2011
Tampa, FL

Agile Software Development Training
November 6–8, 2011
Orlando, FL

conferences

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

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

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

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

BETTER SOFTWARE
A TECHWELL PUBLICATION

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

Visit www.BetterSoftware.com or call 800.450.7854.

SOFTWARE QUALITY ENGINEERING · ASBPE American Society of Business Publication Editors · fma

CONTACT US
Editors: editors@bettersoftware.com

Subscriber Services:
info@bettersoftware.com

Phone: 904.278.0524, 888.268.8770

Fax: 904.278.4380

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

PRINTED WITH SOY INK

Microsoft

Unreal Speed Made Real

Deliver solutions with incredible velocity.

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

VISUAL STUDIO 2010 SPEEDS DEVELOPMENT WITH TOOLS FOR:

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

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

EXPLORE AND EXPERIENCE THE POWER.

Microsoft® Visual Studio® 2010

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

Contributors

Bob Aiello is a consultant, editor in chief for CM Crossroads, and the author of *Configuration Management Best Practices: Practical Methods that Work in the Real World*, Addison-Wesley Professional (http://cmbestpractices.com). Bob has served as the vice chair of the IEEE 828 Standards working group (CM Planning) and is a member of the IEEE Software and Systems Engineering Standards Committee (S2ESC) management board.

Arlen Bankston is the executive vice president and managing partner at LitheSpeed, LLC. Arlen is an established leader in the application and evolution of process management methodologies and is a Lean Six Sigma Master Black Belt and Certified ScrumMaster Trainer. A frequent presenter and trainer at both industry conferences and to Fortune 100 clients, his recent work has centered on combining Lean Six Sigma process improvement methods with agile execution to dramatically improve both the speed and quality of business results.

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

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

Shmuel Gershon is a testing engineer at Intel Corporation. He coaches testers and teams and speaks at public software testing conferences. He is especially interested in the philosophical questions of software testing and is trying to learn enough to be able to understand (but not always answer) them. Read more about Shmuel and his exploratory testing reporting tool at http://testing.gershon.info.

Contributors

Mary Gorman, CBAPTM, 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 and ebgconsulting.com.

EBG Consulting principal consultant and founder Ellen Gottesdiener helps business and technical teams collaborate to define and deliver products customers value and need. Author of two acclaimed books, Ellen works with global clients and speaks at industry conferences. Learn more from her articles, tweets, blog, and free eNewsletter and find resources on EBG's website, ebgconsulting.com. Contact Ellen at ellen@ebgconsulting.com.

Tim Lister is a principal of The Atlantic Systems Guild (www.systemsguild.com). Tim is a well-known expert and lecturer on software management techniques. He co-authored the popular book *Peopleware: Productive Projects and Teams*.

With twenty-five years of project management, software development, engineering, and business experience, Bob Payne is the VP of coaching services at LitheSpeed, LLC and a leading proponent of agile methodologies and agile engineering practices. As host of the AgileToolkit podcast, Bob has produced more than 120 podcasts, recording a variety of industry leaders and agile practitioners.

Dr. Kevin Rutherford is a UK-based agile/XP coach, developer, and project leader with more than twenty-five years' experience in software development. He is the founder of AgileNorth and XP-Manchester, co-author of *Refactoring in Ruby*, and developed the Reek tool for detecting smells in Ruby source code. Contact Kevin at www.kevinrutherford.co.uk.

Bill Wake is a software coach, author, and a senior consultant with Industrial Logic, Inc. (industriallogic.com). His interests include design, serious games, and music. Bill's personal web site is www.xp123.com and his email address is William.Wake@acm.org.

TechWell.com Now in Beta

I'm excited to announce that TechWell.com—our new portal to the interactive communities TechWell Manage, TechWell Agile, and StickyMinds on TechWell—is currently in beta, and we'd love your input. Become a registered member of TechWell, and join a group, comment on blogs, post a question for one of our experts, or start a conversation with your peers.

Once you've had a chance to look around, send your feedback to our community manager, David DeWald, at ddewald@sqe.com.

TechWell builds on the great information and resources that you find in *Better Software* magazine to include enhanced content, streamlined organization, and improved networking features. In fact, you'll find several familiar faces from this issue throughout the TechWell communities.

This issue's Inside Analysis column, "Harvesting Stakeholder Perspectives to Organize Your Backlog," was one of the first columns to appear on TechWell. The authors, Ellen Gottesdiener and Mary Gorman, are long-time contributors to all of our publications and continue to share their years of experience and expert knowledge on TechWell.

Lisa Crispin, regular *Better Software* contributor and interviewer for this installment of From One Expert to Another, is one of our exclusive TechWell bloggers, so make sure to check in regularly to see what new insight she offers.

Lee Copeland, our *Better Software* technical editor, has published many columns on StickyMinds throughout the years, and you can find all these words of wisdom dating back to 2000 on TechWell.

Almost without exception, every author in this issue can be found on TechWell as an author, conference speaker, podcast host, or cited expert. So, when you've finished reading the magazine, learn more by searching our communities.

As always, I hope you enjoy this issue of *Better Software* magazine. And I doubly hope you enjoy the new TechWell communities. Get in there and beta test and let us know what you discover.

Happy reading,

hshanholtzer@sqe.com

Ask to See His …

The things that make management "unscriptable" are also practiced by testers. So, why have testers fallen into the "script trap"?

by Lee Copeland | lcopeland@sqe.com

Recently, I had the privilege of attending the Software Quality Association of Denver (SQuAD) biennial software testing conference. While I'm the program chair of STAR*EAST* and STAR*WEST*—big testing conferences that attract delegates from all over the world—I'm also a fan of smaller, regional conferences, such as SQuAD, and think they deserve our support.

For me, the highlight of the conference was Michael Bolton's keynote presentation, "Testers: Get Out of the Quality Assurance Business," noting that since we have no control over schedule, budget, staff, product scope, development model, customer relationships, contractual obligations, and other vital items, there's no way we can assure anything, especially quality.

Michael explained the difference between checking (verifying that which we already believe is true) and testing (exploration, discovery, investigation, and learning to find new information). He suggested we consider giving these responses in the following common situations:

- When your manager demands to see your test cases, demand to see her management cases.
- When your manager demands to see your test scripts, demand to see his management scripts.
- When your manager demands that every test have a single, precise, and expected result, ask her if every management action has a single, precise, and expected result.

"When your manager demands to see your test scripts, demand to see his management scripts."

The manager would, of course, be incredulous at such requests, responding that "management" can't possibly be reduced to a set of unvarying sequential steps. Management deals with complex, interlocking, nonlinear, ever-changing interactions among goals, people, resources, and priorities. It's far too complicated to script.

The manager would carefully explain (perhaps using small words for our benefit) the five key components of management—planning, staffing, organizing, directing, and controlling. *Planning* consists of establishing goals, objectives, and tasks for our work. *Staffing* is the process of gaining access to people with the right skills, competencies, time, inclination, and interest to participate in our work. *Organizing* consists of allocating resources, both human and physical, to accomplish our work within the constraints imposed on us. *Directing* is the process of coordinating all the ongoing activities to produce the best possible results using our resources. Finally, *controlling* is the process of measuring and evaluating our work and making necessary adjustments to our goals, objectives, and tasks when required. Does that sound like something that could be pre-scripted by an expert and later executed by someone of lesser skills?

As testers, we are quite familiar with these five components—we practice them in our work every day. Testers establish goals, objectives, and tasks to fulfill their testing mission. And, as tests are performed and software defects emerge, we often revise our objectives and tasks to focus our testing efforts on higher-risk areas. Testers are always seeking the right people to perform testing, whether they are part of the test team or external, such as developers, users, customers, conscripts, or volunteers. Selecting and motivating the right mix of people from such a pool is a highly dynamic activity. Testers organize and balance their work using different types of testing, at different periods of time, and over changing circumstances and priorities. Testers direct their own investigations of software products and coordinate the work of others doing the same. Testers measure their work in terms of coverage achieved, defects found, time spent, defects escaped, and testing and repair costs, and then adjust their processes to become more effective and efficient. When interruptions occur and knowledge is gained, new problems are identified and new opportunities emerge. Does this sound like something that could be pre-scripted by an expert and then later executed by a person of lesser skills?

So, why have testers fallen into the "script trap"? Because, as Bolton explains, we have confused checking (which can be scripted and automated) with testing (which cannot). We've forgotten that real testing is, according to Bolton, "an investigation of code, systems, and people and the relationships between them." Such relationships are inherently complex and dynamic. The investigation of these relationships will be complex and dynamic, also. **{end}**

Harvesting Stakeholder Perspective to Organize Your Backlog

This game will help teach your team that the key to organizing the stories in your backlog is to explore their value from multiple perspectives.

by Ellen Gottesdiener and Mary Gorman | ellen@ebgconsulting.com mary@ebgconsulting.com

Every agile or Scrum team knows the dilemma of organizing the backlog of project requirements. It's like cleaning out the garage. You know you'll have to handle everything sooner or later, but where do you start? How do you organize the work to get the most out of it?

It's easiest to follow the squeaky wheel methodology—fulfill the desires of whoever is complaining loudest this week—but that's not the path to delivering the highest value for your customer. The key to organizing the stories in your backlog is to explore their value from a number of perspectives.

This lesson was the goal of a game Mary co-created at the New England Agile Bazaar's Deep Agile Games weekend event. The game is called "The Backlog Is in the Eye of the Beholder." Recently, at the Boston chapter of the International Institute of Business Analysis (IIBA), we facilitated this game for a group of more than sixty people. The gamers learned a lot, and we learned even more.

Sowing the Seeds

The game consists of four rounds, each focused on a different persona related to a farm: the producer/farmer, the customer/buyer, the land owner, and the Farm Bureau inspector. In small teams, the participants analyze a group of tasks (e.g., fertilize crops, spray insecticide, rotate crops, and assure organic). The players organize the tasks according to the perspective of that round's persona.

This game has two purposes. The obvious one is to explore the views of multiple stakeholders and organize the tasks according to who wants what. The less-obvious (but related) one is to illuminate the value of organizing the work *without* prioritizing. If that concept leaves you puzzled, read on. As we said, IIBA participants weren't the only ones who learned valuable lessons.

> "When team members are open to discovery, the very act of collaborating to organize the backlog helps build the project community."

What the Gamers Gleaned

The game worked quickly to meet our first purpose. Exploring different stakeholders' perspectives (in this case, using personas) indeed allowed for progressively deeper understanding of the product needs.

In one example of that deeper understanding, the players realized that a number of stakeholder needs were shared among personas, whereas other needs did not overlap at all—a key discovery in organizing a backlog. How do you apply the lesson? You use a gradient of "don't care" to "really care" for each persona for each product need—a practice that quickly illuminates shared value versus conflicting value. Many of the gamers were surprised to find that understanding the "don't care" perspective was just as important as comprehending the "really care."

When it comes to understanding product needs, the players learned a needed lesson in setting priorities (the second purpose of the game)—namely that it's important to resist the urge to prioritize requirements too soon. Instead, you should learn more about the stakeholders and their needs before making delivery decisions. You can organize the backlog in different ways, and each has benefits and drawbacks that you need to identify and analyze. When you delay prioritizing, it lets you "walk around" the product needs, looking at them from different perspectives.

It also helps you remain open to discovering dependencies among the requirements. Interestingly, we overheard only one of the fourteen teams discuss this topic during each game round. But, we know that requirements interdependencies are a virtual certainty in most projects, and the longer they go unrecognized, the more headaches you'll have.

Speaking of remaining open, the gamers also learned that the persona you analyze *first* can mislead you, prematurely narrowing your thinking about the requirements. You can also cramp your process by reducing the number of personas you invite to the party. As facilitators, we noticed that the participants stuck only with the personas we provided, although during the retrospective several people spoke about other roles or personas.

When team members are open to discovery, the very act of collaborating to organize the backlog—discussing, sorting, and evaluating various stakeholders' needs—helps build the project community.

What We Reaped

After the event, we conducted our facilitators' retrospective and learned some key lessons.

First, when you're exploring product needs, evaluate time as a factor. For each backlog item, factor how time:

- Erodes (or increases) the value of the delivered requirement
- Increases (or reduces) the cost to deliver
- Alters implementation options (because of emerging technologies)
- Heightens the need to retire older technologies

Second, always take into account how the backlog is affected by regulations and policies. There is a cost for noncompliance, such as fines and negative market perception. Consider the probability that changes in regulations will devalue the implementation of a given set of requirements. And, if your business has volatile business rules, consider the value of separating rules from requirements to enable business capability and agility.

A Welcome Windfall

Moments before the groups debriefed their last round of play, we realized there was another learning opportunity: cross-persona theme analysis.

Here's a bit of background. During each of the four persona rounds, the players had used uniquely colored index cards to label themes (or categories) of the backlog items for each persona. For example, the farmer's category cards were blue, the land owner's category cards were yellow, and so on.

At the end of the four rounds, we noticed the colored cards strewn on the tables, and we wondered what would happen if we asked the teams to cluster the entire set and order them according to importance. We decided to take a chance and go for it.

The activity was fascinating to watch. Most teams ordered categories by persona; they clustered all the yellow cards together, all the blue cards, and so on. Only a few ordered them across personas. Perhaps, if we had had more time, it would have allowed them to explore this further.

This exercise taught us two things:

- Build in time to understand the "big-view" across all perspectives—a critical dimension when teams are planning product delivery.
- Adapting a game on the fly is exciting and may yield interesting findings—but it may also introduce risks. When you haven't prepared, your instructions to the players may not be clear. Ours weren't. Had our instructions been clearer, the players would have better understood the objective. Planning also would have allowed us to factor in sufficient time for the work.

High Yield

The Backlog Is in the Eye of the Beholder learning game aligns with our real-world experiences in facilitating and coaching teams. Eliciting, analyzing, and prioritizing needs—prerequisites for planning releases to deliver value—rely on a rich understanding of all stakeholders. You must learn about your stakeholders—their goals, motivations, preferences, and unique definitions of value—before you can organize your backlog for greatest value. And, because you often serve multiple stakeholders, it's crucial to learn how their needs both intersect and depart. **{end}**

This article was originally published on TechWell.com.

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

- Resources

Visit **StickyMinds.com** to comment on this article

Virtual **Resource** Shelf

Author recommended books, blogs, gadgets, websites, and other tools for building better software

Q: What nontechnical book has most influenced your work?

I've always loved Scott Kim's Inversions. They're just the right blend of technical and artistic thinking, and that's what I aspire to.

–Bill Wake

One of my favorite books is *How to Lie with Statistics* by Darrell Huff. In my younger days, as a naïve would-be mathematician, I was shocked to find that people might use numbers to deceive me. One of Huff's favorite "lies" is the Gee-Whiz graph, which uses a number of tricks to manipulate the effect on the reader. By manipulating the x- and y-axis scales, graphs can be made to communicate any idea the creator desires, no matter what the data shows.

–Lee Copeland

Many nontech books stand out, but *The Goal* by Eliyahu M. Goldratt stands head and shoulders above all the others. It's a novel about an operations manager who has three months to save a factory from closure, which he does with the help of Jonah, an expert in the Theory of Constraints. I ask every manager I coach to read it, too. It's the clearest introduction to the importance of a holistic view of organizations and it changed the way I see companies and teams. Everyone to whom I give a copy is blown away and changed forever.

–Kevin Rutherford

I read *Alice's Adventures in Wonderland*, *Through the Looking Glass*, and *Sylvie and Bruno* during my university years, after randomly buying *The Complete Works of Lewis Carroll* from a discount bin. And then I read them again. And again a few years later. The text makes you stop every few paragraphs to ponder a word, a phrase, a joke, or a critique. Not only did it motivate me to try to be creative and clever but, to this day, I use some of Carroll's practical ideas on note taking.

–Shmuel Gershon

I read *Chaos* by James Gleick in early high school and used it as a primary input to a science project on fractals, a new and unfamiliar subject to most at the time. It centers around the science of complexity and dynamical systems, and Gleick is a wonderful writer who humanizes the history, people, and theories behind this potentially dry and mathematically stultifying subject.

It influenced me by providing my first significant public programming excursion (using an Amiga 1000 to display interactive, animated fractals at a time when computers were a rare sight at a science fair at all), by helping me to blend my interests in visual design and development and by kickstarting my knowledge of the complexity theory that underlies most agile approaches today. The fact that I won the fair that year also helped convince me that this stuff could pay.

–Arlen Bankston

Simon Baker

Years in Industry: **19**
Company Name: **Energized Work**
Email: **simon@energizedwork.com**

Interviewed by: **Lisa Crispin**

Email: **lisa.crispin@gmail.com**

We challenge assumptions, provide options, and try to improve it in ways the customer hasn't envisaged. We encourage the customer to launch a minimum viable product at the earliest opportunity and we track the charter measurements to see how it performs in the market.

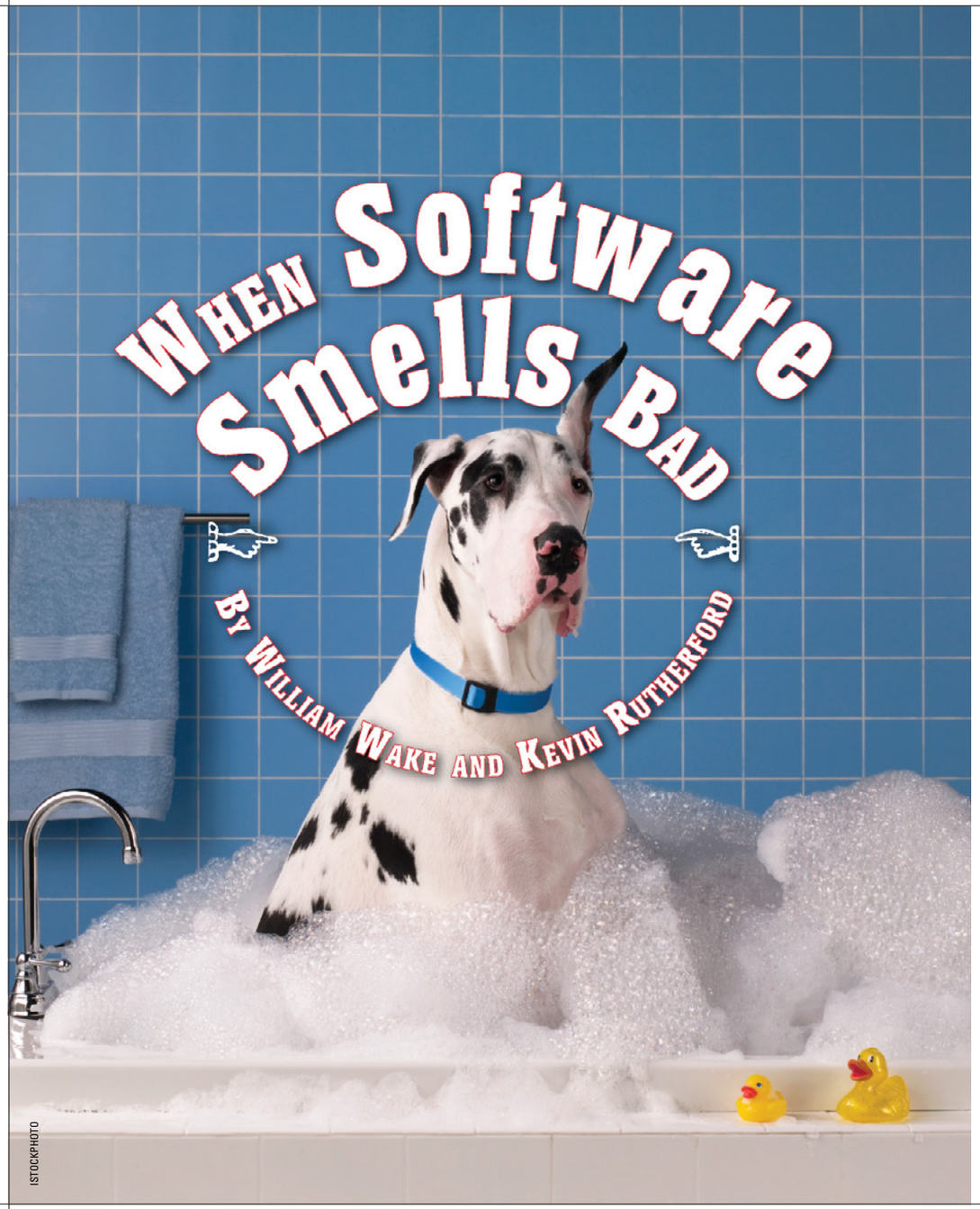
It's easy to enjoy work when humor is used to share ideas and insights and you can put energy into the things you believe in.

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

> **Visibility** has always been a big thing for me. Teams need to see what's important so they can make informed choices and use time effectively.

Leading by example encourages people to try things. I'm constantly experimenting. It's important to make mistakes publicly and show vulnerability.

In a way, a tester is one half of a double act with the customer. The tester transforms the customer's ideas into actionable data and acceptance criteria and is usually the first to challenge priorities.



Imagine a piece of software that works correctly and yet doesn't "feel" well written. How could you describe the code's quality in concrete terms? How do you convert the vague notion—"this is poor code"—into a plan for fixing it? Let us introduce a catalog of software design anti-patterns called code smells and show how to use them both to describe software quality in concrete terms and to plan remedial actions for improving code.

What Is Code Quality?

In recent years, agile methods in general and test-driven development (TDD) in particular have become significantly more popular. The micro-process of TDD tells us to work to the rhythm of red-green-refactor. Specifically, we write a test and see it fail; then, we write code in the most straightforward way possible to make it pass; then, we refactor to clean up any mess we made. But, in that third step, just what should we refactor and why? And if we need to refactor a pile of code that wasn't developed in a test-driven manner, how do we get a handle on where to begin?

Refactoring is defined as "improving software without changing what it does." The intent is to replace bad code with something better. The next questions must then be: What is bad code and what is good code? How can we recognize which is which? And, when we find bad code, what should we do to eliminate it?

The first step is to understand why code quality matters. The answer to that lies in our need to change the code in the future. If this code will never be read again and if it will never be changed, then it only has to be functionally correct. Such code can be messy and disorganized as long as it functions correctly. But, if in the future someone needs to fix a bug, add a new feature, or make a small tweak to the code's behavior, then we need good code. The moment we discover that the code is tortuous to navigate, difficult to understand, or hard to change is the moment we wish it were better written. Over and above functional correctness (which we take as a given in this article), we can define software quality as the ability to change the code easily: Good code supports change; bad code hampers change.

Exercise, Part 1

Before we go any further, look at the program in figure 1. This is a small script written as a command line to-do list. The script is written in Ruby but should be fairly easy to understand even if you are not familiar with scripting languages. If you haven't seen much Ruby, these hints will get you through:

- The "case/when" block is similar to a "switch/case" statement in other languages (but with no fall-through from one case to the next).
- Arrays (the variables initialized to "[]") have a "<<" operator that appends an item to the end of the array.
- Regular expressions are bracketed by "//" and form a kind of pattern object.

The application is written as a big `while` loop, reading input lines and interpreting them as commands. To use this tool, you add new tasks to the list via the "todo" command. See figure 2 for a sample session. Working the list in order, you can either mark the current task "done" or skip it. If you skip a task three times, it's deleted; you'll have to manually re-add it if you intend to do it. If you move past the end of the list, you go back to the first not-done task and start over. There are two other commands: "list" prints a list of all incomplete tasks, and "quit" exits the system (losing all the data—this code hasn't addressed persistence; it's just a simple example).

You can download this code, along with some tests that demonstrate its behavior, from the StickyNotes.

Take five minutes to look over the code and become familiar

```
t = []          # tasks
skp = []        # skip count for each task
c = 0           # index of current task

print "TODO> "
STDOUT.flush
line = gets.chomp

while (!(line == "quit"))
 rest = line.sub(/[^ ]+ /, '')    # remove command from front of line

 case line
 when /^todo/
  t << rest
  skp << 0

 when /^done/
  if (t.length > 0)
   t.delete_at(c)
   skp.delete_at(c)
   c = 0 if c >= t.length
  end

 when /^skip/
  if (t.length > 0)
   skp[c] = skp[c] + 1

   if (skp[c] == 3)
     puts "Too many skips; deleting - " + t[c]
     t.delete_at(c)
     skp.delete_at(c)
     c = 0 if c >= t.length
   else
     c = (c + 1).modulo t.length
   end

  end

 when /^list/
  puts t
  puts

 when /^$/
  # do nothing

 else
  puts "Unknown command. Commands: todo, done, skip, list, or quit"
 end

 if (t.length > 0)
   puts "Current: " + t[c]
 end

 print "TODO> "
 STDOUT.flush
 line = gets.chomp
end
```

Figure 1: todo.rb

Figure 1 (continued): todo.rb

with it. Now, imagine you have been given responsibility for the future maintenance of this script. Review the code's changeability in response to the following potential change requests:

- Add persistence.
- Add a more sophisticated user interface (web or GUI).
- Add the notion of a recurring task, one that is automatically added to the end of the list when it is marked "done." (Think of a task that never goes away, such as "check email.")
- Change the rule for choosing the next task. For example, try an "elevator" rule: At the end of the list, reverse direction rather than starting back at the beginning.
- Keep completed tasks showing on the list until they're specifically cleaned out (but never make them current tasks).
- Manage a separate list that holds deleted tasks.
- Gather statistics about tasks (e.g., number of tasks completed vs. skipped, average time from task creation to completion).

Describe what qualities of the existing script might make those changes difficult or easy. We're not asking you to design the changes—just assess which might be easy and which could be difficult and why. Better yet, do this exercise with a partner or workgroup and discuss the changes you would make. Go ahead, mark up the code. We'll wait.

All done? How easy was that? What vocabulary did you use? If you're anything like us, you'll now have notes about the code being "tangled" or even "monolithic"—still a little vague and not a great contribution toward an improvement plan.

The Language of Smells

To help with this difficulty, the agile community has developed an informal catalog of the most common ways in which software can render change difficult. These problems are often called "code smells," after Kent Beck's analogy between code and babies—"If it stinks, change it" [1]. Most smells are either problems of poor communication (because it's hard to change code you don't understand) or duplication (which more than doubles the risk of making the changes).

Each code smell has an indicative name, a set of tell-tale symptoms so you can spot it easily, an indication of what refactorings you can do to remove the smell, and an indication of what other smells you might want to look for next. We've outlined a few smells in the sidebar.

"Each code smell has an indicative name, a set of tell-tale symptoms so you can spot it easily, an indication of what refactorings you can do to remove the smell, and an indication of what other smells you might want to look for next."

Exercise, Part 2

Using the smell descriptions in the sidebar as a guide, revisit the to-do list script in figure 1 and identify any smells you can. Again, work in a discussion group if you are able. Come back when you're done.

How was the exercise this time? At this point, the smells catalog generally provides a number of benefits. First, our search for problems was more focused because we knew what symptoms to look for. Second, we were able to be more precise about the code's problems and where they occurred. And finally, we had the beginnings of an action plan—a list of things to do that might help this code become more adaptable to future change.

Here are some problems we found with the to-do list script:

Uncommunicative Names—`t`, `skp`, and `c` are too short to communicate their roles.
Comments—The code isn't self-explanatory enough.
Long Method—The whole thing is written as one big script; some code is nested four levels deep (`while / case / if / if`).
Magic Number—"3" is the number of times a task is skipped before it's deleted. English text appears in strings and patterns (limiting internationalization).

A Few Important Smells

Uncommunicative Name

What to Look For:

- One- or two-character names, vowels omitted, or misleading names

What to Do:

- Rename the variable, class, etc.

What to Look for Next:

- Duplication: Look for places where the same thing has different names.

Feature Envy

What to Look For:

- Code references another object more than it references itself, or several clients manipulate a particular type of object in similar ways.

What to Do:

- Extract Method to isolate the similar code.
- Move Method to put it with the referenced object.

What to Look for Next:

- Duplication: Look for further duplication around the clients.
- Communication: Review names for the receiving class for consistency.

Long Method

What to Look For:

- A method with a large number of lines (five lines is large in Ruby)

What to Do:

- Extract Method to break up the long code.

(*CONTINUED ON PAGE 17*)

(*CONTINUED FROM PAGE 16*)

What to Look for Next:

- Duplication: Are the extracted pieces similar? Can they be consolidated?
- Communication: Review names to make sure they communicate well.
- Abstraction: Is there a missing class?
- Flexibility: Is there Feature Envy, where code seems more concerned with another class than with its own class?

DUPLICATED CODE

What to Look For:

- Code that is nearly identical in its text or in its effects

What to Do:

- For code with similar effects, Substitute Algorithm to make the code have similar text.
- For duplication within a class, Extract Method to isolate common parts.
- For duplication among sibling classes, Pull Up Method and Pull Up Instance Variable to bring common parts together. Consider whether there should be a Template Method.
- For duplication among unrelated classes, extract the common part into a separate class or module, or consolidate code onto one class.

What to Look for Next:

- Abstraction: Look for related responsibilities and missing abstractions.

GREEDY MODULE

What to Look For:

- A method with more than one responsibility (especially decisions that will change at different frequencies)

(*CONTINUED ON PAGE 18*)

1. ***Rename Variable***—changed `t` to `task_list`
2. ***Rename Variable***—changed `skp` to `skip_counts`
3. ***Rename Variable***—changed `c` to `current_task_index`
4. ***Extract Method***—created a `next_line` method that prompts the user and returns a String of input text; used in two places in the script
5. ***Extract Class***—created a `TodoList` class; assigned a singleton instance to a constant
6. ***Move Field***—moved `task_list` to be a field `@task_list` on `TodoList`
7. ***Move Field***—moved `skip_counts` to be a field `@skip_counts` on `TodoList`
8. ***Move Code***—moved the code that parses the arguments into the branch for the "todo" command
9. ***Extract Method***—created `append_task`, called from the "todo" branch of the case
10. ***Move Method***—made `append_task` a method on `TodoList`
11. ***Extract Method***—created `delete_task`, called from the "done" and "skip" branches of the case
12. ***Move Method***—made `delete_task` a method on `TodoList`
13. ***Extract Method***—created `length`, called from the "done" and "skip" branches of the case
14. ***Move Method***—made `length` a method on `TodoList`
15. ***Move Code***—moved the wrap-around check on `current_task_index` outside of the case statement
16. ***Move Field***—moved `current_task_index` to be a field `@current_task_index` on `TodoList`
17. ***Remove Parameter***—`TodoList.delete_task` can use `@current_task_index` instead of taking a parameter
18. ***Extract Method***—created `current_task_description`, called from the "skip" and "list" branches of the case
19. ***Move Method***—made `current_task_description` a method on `TodoList`
20. ***Extract Method***—created `skip_current` to hold the body of the "skip" branch of the case
21. ***Move Method***—made `skip_current` a method on `TodoList`
22. ***Replace Conditional with Guard Clause***—inverted the check at the top of `skip_current` in order to reduce the nesting levels in that method
23. ***Extract Method***—`created check_for_current_overflow`, called from `TodoList.delete_task` and `TodoList.skip_current`
24. ***Extract Surrounding Method***—created private method `TodoList.edit_list`, which yields to a supplied block only if `@task_list` is non-empty, and then wraps `@current_task_index` around to the top afterwards if necessary; called from `TodoList.skip_current` and `TodoList.delete_current_task`
25. ***Inline Method***—inlined `TodoList.check_for_current_overflow` into its only caller, `TodoList.edit_list`
26. ***Remove Method***—deleted the public getters and setters for `@skip_counts` and `@current_task_index` as these fields are no longer used outside of the `TodoList` class
27. ***Extract Method***—created a `to_s` method to return a String listing the tasks
28. ***Move Method***—moved `to_s` into `TodoList`
29. ***Remove Method***—deleted the public getter and setter for `@task_list` as this field is no longer used outside of the `TodoList` class
30. ***Extract Method***—created a `empty?` method to indicate whether the list has any tasks; called from two places in the main loop
31. ***Move Method***—made `empty?` a method on `TodoList`
32. ***Remove Method***—`TodoList.length` is no longer needed
33. ***Extract Method***—created an `execute(command, arg)` method containing the whole of the case statement; returns true to continue, or false if the user wants to quit

Figure 3: Refactoring change log

> **"Methods are fundamental units of code reuse, which means that the remedies for most code smells involve creating or manipulating methods."**

Duplicated Code—Sometimes we have literal duplication (like the prompt and line fetching code or the way tasks are deleted). Other times, it's more subtle, such as the two ways used to wrap around when we hit the end of the list (explicitly checking and setting to zero, or using the modulus operation).

Greedy Module—The user interface (such as it is) is mixed up with the business logic, even though these two aspects of the code will change in different directions at different times in response to different forces.

Open Secret—A task is maintained as a string, and the list is maintained as a pair of arrays and a counter. Note how the two arrays are kept parallel.

Feature Envy—Chunks of the code want to "live with" the tasks list.

Note that by naming the smells, we have made them more tangible and more precisely located the code's problems. Now, by collecting the various "What to Do" notes from the smells we found, we can begin to select a few first refactoring actions. In order to address the smells we found, we'd likely make the names longer and more descriptive of the program's design, pull out a `TodoList` class, and perhaps pull out a `Task` class.

This plan derives directly from knowledge of the code smells. The smells catalog has helped us review the code, communicate clearly about its problems, and form an action plan for refactoring it. See figure 3 on page 17, for a log of the detailed changes made during our most recent run at tidying up the basic smells in this code.

After these steps, the code is simple and clean. The mechanics of dealing with the list are hidden inside the `TodoList` class, which has no public fields, and those mechanics are nicely separated from the tool's user interface. We never found a need to move to the third step of the plan and extract a Task class; you could do that yourself as an exercise.

As a footnote to the refactoring log: Ten of the thirty-three steps used Extract Method, and twenty-two of the thirty-three were operations on whole methods. Methods are fundamental units of code reuse, which means that the remedies for most code smells involve creating or manipulating methods. We performed thirty refactorings on methods in order to extract and encapsulate one class. While this seems like a lot, many were quite small steps, and all followed the basic recipe set out in *Refactoring in Ruby* for fixing the Greedy Module smell.

Conclusion

Code smells form a vocabulary for discussing code quality, and, hence, for describing how well suited code might be to change. The smells also provide good indications as to what to refactor and how. Learn the language of code smells to get started on refactoring—the road to faster development. **{end}**

william.wake@acm.org
kevin@rutherford-software.com

Sticky Notes

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

- `Todo.rb` and demonstration lists
- References

(*CONTINUED FROM PAGE 17*)

- Clumsy test fixture setup

What to Do:

- Extract Class or Extract Module to split up the module.

What to Look for Next:

- Communication: Review names to make sure they communicate well.
- Simplicity: Check for Feature Envy.
- Testability: Simplify the unit tests.

OPEN SECRET

What to Look For:

- Several classes or modules know how to interpret a simple value.
- Several classes or modules know what data is held in each slot of an array or hash.

What to Do:

- Extract Class or Introduce Parameter Object to consolidate the interpreting code.
- For an array or hash, Replace Array (or Hash) with Object.

What to Look for Next:

- Duplication: Look for Feature Envy around the new class.
- Communication: Review related names to make sure they communicate well.
- Flexibility: Consider whether the new object can be used earlier in time.

(SUMMARIZED FROM *REFACTORING IN RUBY* [2])

ISTOCKPHOTO

IMAGINARY FRIENDS

CREATING SOFTWARE WITH PERSONAS

BY SHMUEL GERSHON

> "It might seem counterintuitive, but designing for a single user is the most effective way to satisfy a broad population." – Alan Cooper, *The Inmates are Running the Asylum*

We commonly define a computer program as "a sequence of instructions written to perform a specified task." Software *is* what it *does*. With that mentality, it is easy for us—not only testers, but also programmers, architects, and customers—to think that the solution to our dissatisfaction with software lies in a different set of features. A common development process starts with requirements, conveyed either as a document or in verbal or implicit team communication. These requirements enumerate a set of functions that is reflected in the design, programming, and testing. The entire organization is preparing to make the product meet the required functionalities.

Let's look at a website devoted to booking flights online. A simplified requirements document for a product like this might include the following:

- The user shall see all options for flights between two chosen locations.
- The software shall present the user with discounted locations and last-minute deals.
- The software should accommodate booking for one flier or many fliers.
- The user shall be able to order either round-trip or one-way flights.

Many ticket-booking websites are likely to share a significant part of the full requirements list, but the differences between sites show that there is more to software than the functions it implements. That's because written or verbal requirements lists—even when built with users in mind—are poor channels for describing usability or any other expression of experience with the software. Often, the best requirements we get are along the lines of "The program should be user friendly" or "The application should have good usability."

But, what if we could visualize experiences and usability just like we do with concrete features? One way to do this is by building a *persona*—an archetype of a user segment that we provide with substantial details about life and work to help

ensure a consistent behavior.

Who might be typical users of our ticket-booking site? Here are some simplified examples:

- ***Jerome, 42***, from Tennessee, is a frequent flier who spends half the year traveling. He travels the world as a senior representative of a machinery company, but most often goes to repeated destinations in the US and China.
- ***Jessica, 30***, of San Francisco, travels rarely—only when her husband's family in London has a celebration. She is very comfortable using computers and will visit a variety of websites when shopping for tickets.
- ***Peter, 35***, is married with two kids. His wife often travels to Europe for business, and, when possible, he joins her on the trip with or without the kids.
- ***Irene, 38***, is a travel agent from Boston. She uses the site as an alternative to her official ticketing system to check different options and combinations to reach a destination. Irene is very skilled with spreadsheet software and the ticketing system she uses for work, but she lacks proficiency with other applications.

Independent of how applicable this specific list may be to a real product, reading the requirements again with the personas in mind gives us new insights.

When staging visits to the website by these primary personas, we may notice that most—if not all—of the time, they start with a determined destination, making the banners and ad space for discount and last-minute destinations useless. Frequent flier "Jerome" ignores the sale ads. Should we hide the discounts to avoid disturbances? Alternatively, should we emphasize the discounts even more to compensate for the lack of relevance?

In a similar manner, travel agent "Irene" is very happy with the option to query flights for multiple passengers at one time. As a parent, "Peter" notes the need to display clearly how to mark some passengers as "child age."

"When staging visits to the website by these primary personas, we may notice that most—if not all—of the time, they start with a determined destination, making the banners and ad space for discount and last-minute destinations useless."

Different companies and contexts will generate different choices when building the personas, which create distinctive experiences. The method gives uniqueness and adaptability to software projects.

Personas have been in use as a marketing tool for many years. They became popular as a software tool after Alan Cooper presented them in his 1999 book *The Inmates are Running the Asylum*, where he argues that we, as software people, rarely manage to provide software that fits the users' needs due to our inability to detach ourselves from our own knowledge and needs.

When we characterize users in a software project, we usually do so by dividing the users into generic groups like "advanced users," "single-parent mothers," "students," or "beginners." However, the personas method symbolizes user types by using precise details to define very specific (though fabricated) users. These personas must represent the most important users of our system, and they ought to be believable and familiar to as many teams working on the project as possible. They change the focus of the work from "average users" to specific people with needs and habits.

The "preciseness" we practice with personas is not a mere expression of language. In order to create a persona, we breathe into him a very complex identity. We know this character intimately. He has a name, photograph, age, family status, level of computer literacy, level of acquaintance with our product (and those of our competitors), domain knowledge, level of job satisfaction, and a job routine. What are his work hours, and does he work overtime? If he commutes by car, what model and color? If he commutes by public transport, how many buses does he take? Where does he look for help when stuck, and what does he do when there's an error? Of particular value are his goals when using our software, as we want to make sure those goals have as few obstacles as possible.

Imagining your users' routines may help you identify needs. For example, if your product is interpersonal communication software, the private relationships of a persona may trigger interesting questions about privacy and user control on data retention.

The personas method grants additional benefits. First, knowing a persona as closely as a real person provides answers to questions like "Is this command in the correct menu?"; "Are these details known before performing the operation?"; and "Is the nomenclature clear?"

Secondly, calling a user by his name creates a personal commitment to him, solving three common problems:

- ***The Elastic User Problem***—assigning vaguely defined users with opposite traits or conflicting preferences
- ***The Design Justification Problem***—justifying the rationale behind a less-than-perfect design after it is complete
- ***The Self-referential Design Problem***—projecting onto users our own knowledge and preferences

Although personas are fictitious users, building them requires an idea of who our real users are. Begin with a base of real data from usage statistics, surveys, and user observation if

you have them. Lacking these, personas invented from the team's imagination are still better than working with elastic, generic users. As personas are unique for the users of each product, it is not worth copying them between projects. But, you may use existing personas as a base for new ones if the market segments are similar.

Some teams may be skeptical at the start and want to disregard the name and picture of a persona, but these two characteristics are essential to making personas "real." Conversely, we should not use a real person instead of an artificial one, because we don't want to be limited by real people's quirks and caprices.

End-users are not always equipped to ideate a solution for their problems. Likewise, personas may be poor software designers. Thus, instead of asking how a feature should be designed, you'll get better results by asking design questions like "Which of the steps are hindering your progress on this specific design?" or "With which of these two design options is it easier to attain the objective?"

Another advantage of personas is that they expose pain points clearly and effortlessly. Additionally, these pain points are easier for the rest of the team (including programmers and managers) to understand when the team faces the persona.

Some workplaces hang posters depicting the personas in meeting rooms to make them visible and present during conversations about quality and bugs. Others place the posters on chairs around the table to let "Jerome," "Jessica," "Peter," and "Irene" have their say on decisions. When the characters are visible and known to all, chances increase for more focused discussions and collective understanding of bugs and challenges.

This technique is suitable not only to graphical interfaces (GUIs) but also to any system that has an external interface or interaction. For instance, for an automated attendant phone interface, personas can help us ask questions like "Which are the most common operations for 'Ariel'?" or "Which personal data is readily available for 'Aaron' (72 years old) during the call?" For a programming library (like an API or SDK), we could ask, "Over which of the parameters does 'Leonard' want control?" or "Which of the operations does he prefer the library to perform for him without interaction?"

We've discussed how personas can aid or direct the design process to make software better fit the problem it's trying to solve and focus test efforts on important risks to the user experience. Personas help us understand not only the "what" (functions) of requirements but also the "who" and "how" (users) and the "why" (goals).

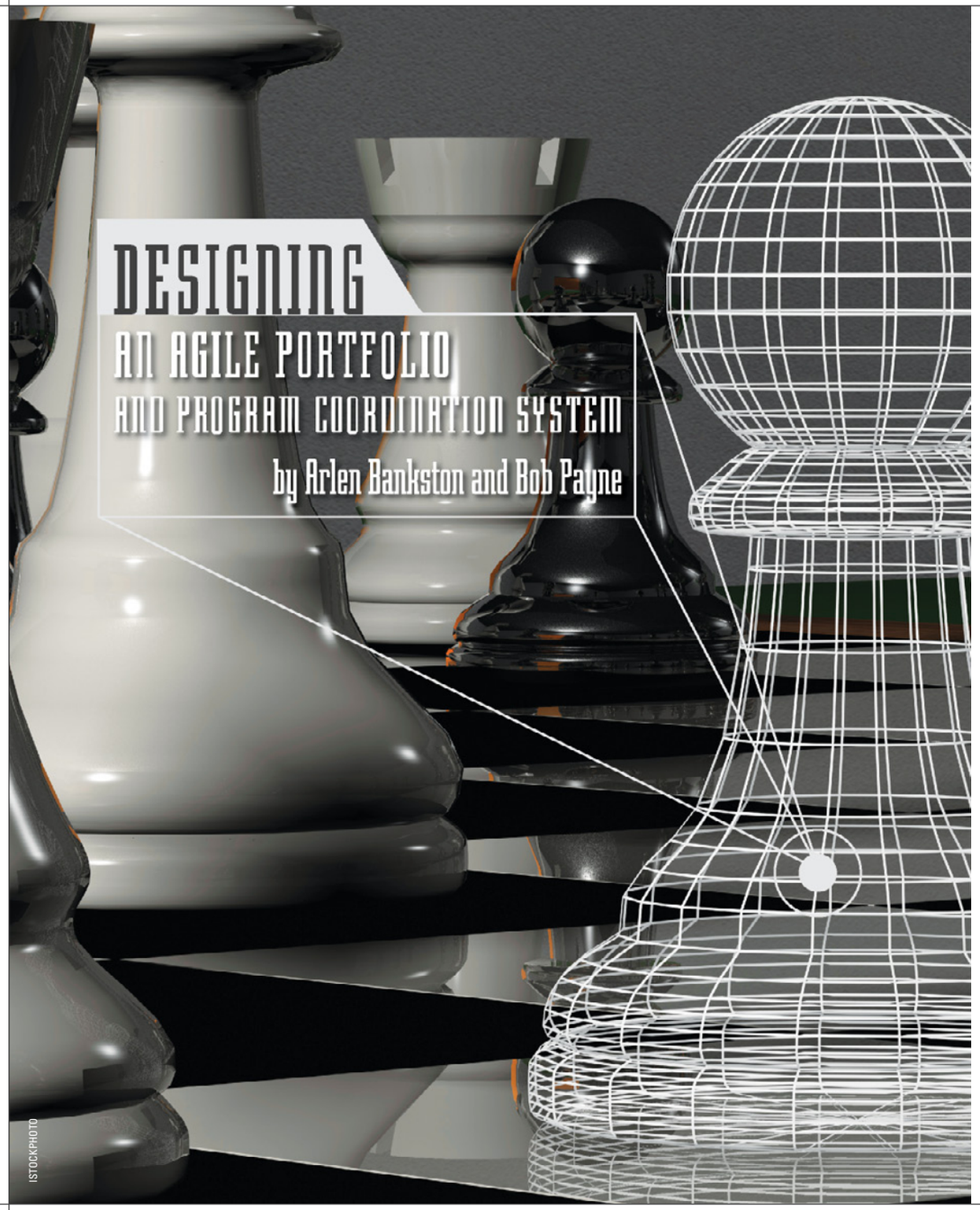
The next step is for you to try personas in your own project. It takes just a few hours to develop a list of ficticious users using this method in its most basic form. An impersonal approach to software development can hinder the creation of a positive and useful experience to users, but an intimate and personal relationship with imaginary users has the power to simplify this task. **{end}**

shmuel@gershon.info
testing.gershon.info

Sticky Notes

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

- Automated test tools
- References



ISTOCKPHOTO

Many otherwise astute agilists have struggled when faced with the challenge of scaling methods like Scrum to large, multiteam projects and programs. Luckily, this shared hardship has resulted in some creative solutions to the problems such efforts can pose. We will explore key aspects of creating a simple but effective agile-ready coordination system for managing such initiatives, based upon the authors' observations and experiences across widely differing companies.

Guiding Principles

The goals of an agile-appropriate portfolio coordination system might be stated as follows:

Build trust and engagement through transparency by making priorities, dependencies, and capacity visible and discussing them regularly.

Facilitate informed strategic guidance via clear communication to key stakeholders about current and potential delivery challenges.

Bridge silos and functional organizations with regular, coordinated collaborative touch points.

Provide rapid, targeted portfolio management with minimal time or capital investment.

We will explore how these objectives might be achieved in a process with minimal overhead and setup time.

The Scrum of Scrums Is Not Enough

Much of the early agile literature spoke from the perspective of a single team. Scrum, the most popular agile framework today, espoused a simple mechanism called the Scrum of Scrums for coordination between multiple teams. However, this mechanism is rather loosely defined and often is inadequate for complex dependency management across multiple releases and a portfolio of projects.

Originally, the Scrum of Scrums was described as a daily meeting between ScrumMasters across teams, which followed a similar structure to the daily scrum but focused on progress, plans, and blockers at the team rather than individual level. One key problem with this system was that it implicitly relied on ScrumMasters to know the details of every issue. Also, the fact that external stakeholders are often needed for negotiation, communication, or blocker removal can make their absence a progress-killing issue.

Many teams have moved beyond the Scrum of Scrums or modified it in ways that they find better suited to the complexities of their situations.

The Ubiquitous Visual Management System

Large, visible "information radiators" have been a common theme in agile implementations, and they figure perhaps even more prominently in the lean manufacturing implementations that inspired many agile values and practices.

For example, Japanese companies such as Toyota long shunned complex material resource planning systems in favor of simple paper charts, bins sized to fit just so many parts, and other similarly low-tech tools. The persistent presence, ease of creation and change, and ability of such systems to facilitate

group interactivity was hard to beat.

In today's agile teams, one continues to encounter physical Scrum taskboards being used alongside full-fledged software suites, proof of the efficacy and persistent appeal of physicality.

Creating a Portfolio Management System and Team

Let us tell a story by way of example: Nine teams are working together on the delivery of several major projects on an enterprise business intelligence platform. The teams are mostly focused on delivery of independent feature groups, but there are two exceptions. One team helps to create and manage shared data services and enterprise service bus integration; this team's backlog is populated by the other teams. Another team is a vendor that is implementing its software to provide some visualization pathways for mobile and tablet devices. It's operating on a slightly modified waterfall development lifecycle.

The rapid pace, continuous integration, and high information flow across a set of agile teams require an equally nimble coordinating structure. The teams also want to engage frequently with high-level stakeholders, so meetings have to be short and focused, which will be a welcome change in any case. The system as a whole will consist of a team, a process, and a visual management board.

The Portfolio Management Team

The first order of business is defining the portfolio management team's roles and responsibilities:

One chief ScrumMaster will lead the creation, maintenance, and facilitation of the portfolio management system.

Team ScrumMasters will provide insight into dependencies and blockers, as well as highlight process improvement and collaborative opportunities.

The project manager from the vendor waterfall team will provide updates on his team's status, focusing on frequent short-term deliverables, actionable information, and stable interfaces.

Team dependency representatives will be nominated by each team, changing as necessary each sprint, to provide insight into specific technical and process issues.

Stakeholder dependency representatives must attend when called upon by the teams. They will generally represent key customers and affected departments, such as sales, marketing, and operations.

Product owners will collaborate between themselves to optimize realization of their value propositions, adjusting scope and workloads as appropriate.

One strategic product owner will be responsible for facilitating high-level tradeoffs, ensuring smart release strategies across the teams and outbound communication to interested stakeholders.

The objective of this team is to ensure that the organization receives valuable incremental releases of the reporting system on a continuous basis, allowing beta users throughout the organization to test new functions and processes and then to see them refined as the program proceeds.

The Portfolio Management Process

Everyone agrees that meeting in person is impor-

tant, for reasons of transparency, effective group collaboration, and time. Team members agree to meet four times per two-week sprint, a cadence that will be reviewed and refined as necessary to keep up with the needs at hand. They craft the following agenda: Review action items from previous meetings; update visual management system, ensuring that each team's sprints are updated to reflect completed and planned items; review status and roadblocks by major feature set; and review new scope or action items that have been identified or suggested.

Team members recall the criticality of a short, focused meeting that doesn't waste anyone's time, so they agree on the following rules of conduct: All attendees will be present in person, barring prior notice; attendees will come prepared with detailed information as needed according to their roles; the core meeting will be timeboxed to half an hour, with small group focus sessions occurring afterward as necessary; and problem resolution will be conducted offline by the necessary parties in these focus sessions.

The Visual Management Board

The team has seen the importance and ease of physical tools for large group activities and works to create a board that will quickly and accurately represent current plans and activities and, more critically still, will facilitate actionable conversations. This board (illustrated in figure 1) will require careful design and continual refinement to ensure that it accomplishes its intended purpose into the future.

The team quickly sequesters a large magnetic whiteboard in a reasonably central conference room; team members will use this in concert with small magnets and index cards to visualize the flow of work and value. This setup has the advantages of being tactile and easy to change, facilitating impromptu sketches as a bonus.

A few design principles, philosophies, and assumptions guide the team's board design: Layout and lines should be neat and clean, it should be visible from a distance of ten feet, wall design

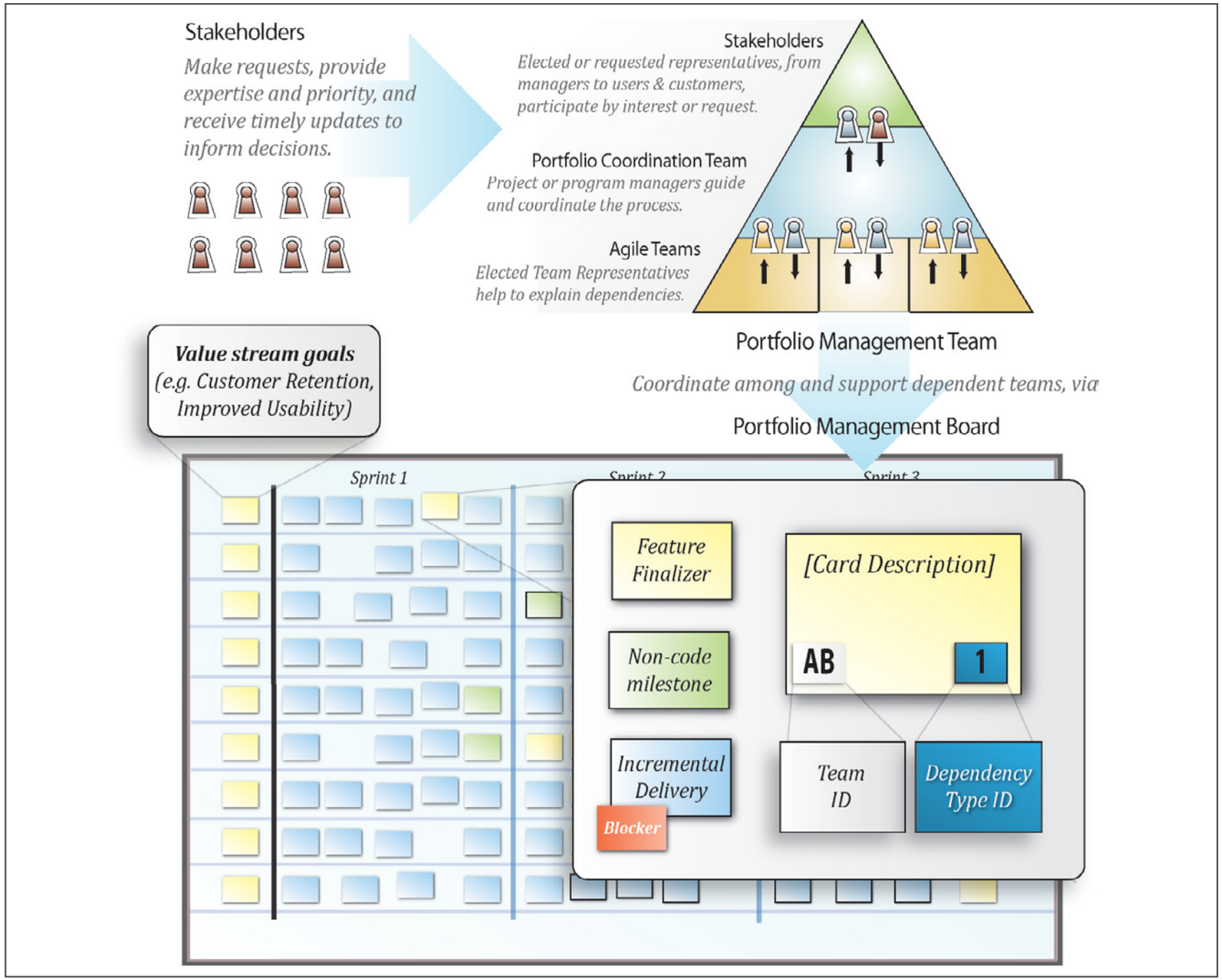


Figure 1: Portfolio management process and visual management board

should implicitly limit the amount of work in process, and five to ten portfolio cards will be the median output for most teams during a sprint.

The team defines the core elements of the visual management board:

Rows represent large dependent units of value, mostly a combination of department and report type in this case.
Columns represent timeboxes with decreasing granularity for the future, e.g., two-week sprints for the next three months, monthly for the following six months, and then quarterly after that.
Releases are represented by green cards at the top of each column, listing the major features of each release.
Value Cards represent units of value (e.g., features) that will contribute to a feature stream release or are required by dependent teams. These are not story cards but represent the externally consumable feature delivery by each team. Teams must work to make these concrete with some representing "soft" deliverables such as design checkpoints.
Dependencies are represented by tags.
Teams are denoted by stickers on each card.
Card Color denotes the type of delivery from the team, e.g., green for a card finalizing a particular release feature, blue for a technical dependency on another team, and yellow for non code milestones.
Blockers are represented by big red "X" magnets.

A Subtly Winding Road

After a few months of use, stakeholders are happy with the lack of surprises and increased flexibility that this lightweight system has offered them. The face-to-face nature and reliance on concrete deliverables has increased accountability and greatly improved cross-team coordination and collaboration. The board itself has seen a healthy batch of tweaks and adjustments, keeping up with the rate of learning and change. The team is now looking at digital solutions for some of their distributed projects, hopefully replicating the success of this program.

Dealing with Distance

While most team members would prefer to be collocated with their fellows and dependent teams, this is, of course, often not the reality of the matter. However, all is not necessarily lost; a few options have been tried and worked, and others with promise lie on the horizon.

The Digital Option

Many agile lifecycle management products exist at this point, such as VersionOne Ultimate, the GreenHopper Jira framework by GreenPepper software, Thoughtworks' Mingle platform, Rally, and many others. Some of these offer robust portfolio and program management capabilities.

A key advantage to such systems, aside from the obvious facilitation of distance collaboration, is that they allow deeper and more rapid reporting and analysis. One significant problem is that they don't allow for the multiperson parallel interaction often seen at such sessions. Also, it can be difficult to "see the big picture" easily across multiple teams given current display and view options.

Many teams have chosen to take the best of both worlds by using the board as the basis for the meeting then having folks update the tools as appropriate after each meeting. Given the focused nature of changes at this level and tight meeting duration, the volume of updates is not very high, so the costs of double entry are low.

The Mirror Option

When there are only two to three dependent locales, some teams have chosen to keep synchronized boards, or at least portions of them. This is generally done when there are significant changes at each locus but not a great deal between them. This can justify the high bandwidth interactivity of the physical medium while minimizing the amount of board synchronization necessary at any point in time.

Two Real-world Examples

Large Enterprise

One of the world's largest insurance companies managed delivery of eleven major feature streams across seventeen agile and waterfall teams. Coordination between the teams was tracked on a visual management system they called the portfolio alignment wall. On-time delivery was achieved with higher than average stakeholder satisfaction.

Agile Startup

A leading DC-area startup serving utility companies balanced workloads and managed dependencies between five teams working on widely differing systems. Meetings were weekly and generally lasted less than an hour, and scary surprises were rare.

Your Next Steps

Any company running multiple Scrum teams can use this basic pattern to their advantage. The physical board in particular has offered a surprising degree of transparency and focus in numerous organizations.

If you'd like to give the method a try, it's easy enough to get started: Find a group of work streams that could benefit from tight, lightweight coordination and form a coordination team as described above. Agree on your own goals, objectives, and design philosophies; put together a visual management system that supports them; and get started. Remember, no system is perfect on launch, so inspect and adapt! **{end}**

arlen.bankston@lithespeed.com
bob.payne@lithespeed.com

ALMComplete 9.6

BEVERLY, MA—SmartBear Software has announced ALMComplete 9.6, the latest release of its application lifecycle management tool. ALMComplete 9.6 enhancements (which also include DevComplete and QAComplete) add additional features for distributed development teams and project managers, improving visibility and efficiency through online task boards, and customer collaboration through customizable support management micro-sites.

ALMComplete 9.6 includes capabilities for cross-project task management, customer support incident tracking, and test automation with the following new features and enhancements:

- Cross-project task management—new task board. The new task board gives developers the ability to organize all of their assigned tasks and identify critical or delayed items. The simple-to-use, drag-and-drop interface enables developers to reprioritize tasks just like sticky notes on a whiteboard to stay on track whether in or out of the office.
- Support management—enhanced customizable customer support ticket micro-sites. Improved self-service management and customization of support ticket micro-sites enable clients to create, manage, and customize the branding and layout of the web pages they use to track customer service issues and defects. Tracking what issues must move to development versus which can be easily managed by support also improves customer satisfaction and reduces support calls.
- Automated testing--New Automation Scheduler. A new automation scheduler allows development teams to schedule TestComplete or HP QuickTest Professional (QTP) tests to automatically run during off hours or between builds then view results in the test cases dashboard. Supports automated testing with scripts residing locally or on network shares.

smartbear.com/almcomplete/whats-new/

Orchestrated ALM Dashboard

REDWOOD CITY, CA—Serena Software has announced new technology and customer milestones in its Orchestrated Application Lifecycle Management (ALM) strategy.

Serena's orchestrated approach applies process automation to the application delivery supply chain, allowing software developers to work more effectively and reduce the challenges of siloed project teams working with multiple development tools and methodologies.

The new Orchestrated ALM Dashboard supports interoperability and standards. An effective dashboard must be able to flow work and insight to developers, analysts, executives, operations managers, and other stakeholders in the application lifecycle. While other dashboards provide a view into key performance indicators and correlated metrics, most are built to report on a single vendor's technology. This prevents companies from gaining an intelligent, comprehensive, and timely look at how their critical processes are functioning.

www.serena.com

Seapine ALM 2011.1

CINCINNATI—Seapine Software has announced details about its release of Seapine ALM 2011.1. The new version of this popular application lifecycle management suite of tools will include many new customer-requested features, including enhancements to help software developers and testers better meet internal and industry standards and regulations.

Seapine ALM 2011.1 includes new features and enhancements for TestTrack, Surround SCM, and QA Wizard Pro. The changes include extensive enhancements for enabling better creation and re-use of test cases, tracking test execution progress, and ensuring adherence to best practices and compliance requirements during testing.

Seapine ALM 2011.1 also adds new Surround SCM features, including support for shelving in-progress changes, enhanced trigger capabilities, and more.

Seapine added workflow history diagrams and improved task board and traceability reports to TestTrack, as well as extensive TestTrack TCM enhancements. One such enhancement is a detail grid view that provides a new format for analyzing test run results. Powerful compliance options for test runs are also new for TestTrack TCM in 2011.1.

New QA Wizard Pro features include a new Keyword View, status tool enhancements, and general report improvements. In addition to the Microsoft Silverlight support added in 2011.0.1, Seapine also added support for the regression testing of Adobe Flash applications with QA Wizard Pro.

www.seapine.com/alm20111

Web Performance Load Tester version 3.6

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

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

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

webperformanceinc.com

Requirements to Object-Code Traceability

SAN BRUNO, CA—LDRA has developed another first—requirements-to-object-code traceability. In the safety-critical domain, devices required to meet the most critical levels of certification must verify software traceability from requirements through design to code at both source and object-code levels. By providing the most comprehensive traceability, LDRA ensures that verification problems found at the object-code level can be quickly and easily traced to the originating source code and requirements levels.

Evidence that all lines of software have been fully tested at the source- and object-code levels is becoming more important for a number of industries. DO-178C, the new avionics software standard, will soon mandate this for the most critical software, and medical and automotive industries are recognizing that this verification process is equally valuable in their environments. Discrepancies caused by compiler interpretation or program optimization can lead to code verification passing at the source level but failing at the assembler object-code level.

Tracing the object code—also referred to as assembler code—back to the originating high-level source code is a tedious, time-consuming challenge without requirements-to-object-code traceability.

In the past, many companies needing to meet stringent certification requirements verified their object code using in-house tools. However, with the adoption of more complex architectures, engineering teams no longer have in-house expertise on the modern architectures, nor can they afford to develop and maintain complex object-level verification tools for project-specific implementations. LDRA equips developers with the ability to review code instruction by instruction, while eliminating the cost of developing and maintaining tools in-house.

www.ldra.com/emf_rm-svt_2011.asp.

SmarteQM Version 3.0

AUSTIN, TX—SmarteSoft's SmarteQM provides complete end-to-end test lifecycle management. Manage project requirements, releases, test cases, issues, and tasks from one unified environment. SmarteQM gives each member of the team, from software development to QA testers, a high-level view of progress as well as depth of detail and ownership of each piece of the project. Up-to-the-minute status, critical for agile or other continuous improvement software development methods, is available via email alerts and an extensive library of printable reports and graphical charts. SmarteQM is web-based and its intuitive graphical user interface is easy to learn and quick to deploy.

Version 3.0 includes new features and functions that provide automated test execution and management including keyword driven testing. A new screenshot utility provides easy uploading of screen captures to artifacts in the system. An enhanced email notification function includes customizable events and new templates. You can track discussion threads for requirements, test cases, test sets, releases, and tasks. Performance of the application has been enhanced for improved usability including right-click shortcut menus. A subscription function lets users subscribe to specific artifacts for closer tracking of activity.

info@smartesoft.com.

FAQ

expert answers to frequently asked questions

by Tim Lister
tlister@systemsguild.

How detailed should a requirement spec be?

Let me start off by saying that if this were a question with a simple declarative answer, it would have been answered years ago. But it is now 2011, and I hear the question many places I go.

My answer to the appropriate level of detail has two parts:

Part 1—I think it was Alan Davis, author of *Just Enough Requirements Management*, who first stated that a requirement should stop at the level of indifference. You specify down to the point where you can say that no matter how the developer implements this requirement, as long as she obeys it to the level specified, you will find the implementation completely acceptable.

For example, I am going to visit a client, and when I land at the nearest airport to their offices, I'll rent a car. I go online to the car rental company of choice and login. I then specify the pick up and return location and the date and time of pick up and drop off. The site offers me categories of automobiles, and I select "Economy" for $39.95/day, plus applicable taxes. I get a confirmation number to close the transaction. Note that I never specified the specific make, model, mileage, or color of the car. This is acceptable to me. I am renting a car for three days, and then I'll hand back the keys. My level of indifference is high. Obviously, if my wife and I are in the market to buy a new car, our level of indifference is much, much lower. We will probably keep this car for eight to ten years, and we want to fit the car to our needs and wants for that time period, not three days.

I really like the level of indifference criterion because at its heart it is giving the developer every degree of freedom she deserves, no more and no less.

Part 2—The second part of my response pivots on when does the requirement shift from being a statement of the need or want of the business to a statement of how best to implement that want or need? As long as you are stating an outcome you need or want, you are probably safely in the land of business rules or business choices. When you get into specifying which design choices are acceptable, either you are now stating constraints to the designers, or you have crossed the border into design, itself. A constraint is a limitation on any acceptable solution for a business reason. For example, "This application shall work on all 4G mobile phones" may be a perfectly reasonable constraint, because the business is purposely aiming at that market segment. On the other hand, "All bills of lading shall be faxed to the shipping agent responsible for the off-load of cargo at that port" may well be stepping into design, since there may be no business rule that the only delivery method must be fax.

This is not to say that requirements cannot be very detailed and specific. If the business rules are very detailed and specific, then the requirement must reflect that. "The system shall calculate the compound interest for the loan" is way too high level for the developer to have a great chance at implementing something that the specifier will find completely acceptable. (See the indifference rule of Part 1.) At the minimum, the specifier needs to state exactly the formula for interest calculation and exactly the time period when the interest is calculated. This has nothing to do with systems and everything to do with business rules. Handing it over to the developer is demanding that she become a business analyst.

So, how detailed should spec be? Detailed enough so that the designers can deliver an acceptable process but not too detailed that it bleeds into a design document. Give developers every degree of freedom, no more no less, and give them a chance to build an acceptable solution!

Raising the Bar for Configuration Management

While CM has grown as a discipline, not all CM professionals have kept current with their skills. This is a call to raise our standards.

by Bob Aiello | bob.aiello@ieee.com

Configuration and release management has matured into a discipline that is a must-have for any software or systems development effort. Agile or not, development groups all desire best practices, including continuous integration and automated build, package, and deployment with dashboards to show the current state of the release management process. While configuration management (CM) has grown as a discipline, many CM gurus have not. I am sometimes shocked to find CM consultants who don't know the basics. Even worse, some very large CM consultancies fall far short in terms of the necessary knowledge and skills. I would like to outline what needs to be done to raise the bar for CM.

Regulatory Requirements Forgotten

Banks, government agencies, defense contractors, and many publicly traded companies are required to implement configuration and release management practices that meet very specific guidelines as defined by widely accepted industry standards and frameworks. In practical terms, these organizations need to have code baselines identified and independently built with a "separation of duties," including deployment to production by someone other than the programmer who wrote the code. Change management needs to include traceability to document exactly who authorized a particular change request with the essential procedures to conduct a functional and physical configuration audit. There's more, but these are just some of the basics that many technology professionals seem to have forgotten—that is, until internal or external auditors come by and then my phone rings.

> "Can you imagine paying a contractor to build your new home, only to discover that he was completely unaware of the building code

Standards and Frameworks Define CM

CM is defined in standards supported by many organizations, including the IEEE, EIA, ISO, and ANSI. I have had the pleasure of being a member of the IEEE 828 standards working group where we recently received approval for a new version of the CM standard that moves from CM planning to a document that provides CM-related guidance for every phase of the software or system delivery cycle. The new EIA/ANSI National Consensus Standard for Configuration Management also has recently been approved after years of effort from the working group. Frameworks include ISACA Cobit, itSMF ITIL, and the SEI's CMMI. Most of these frameworks (and standards) provide detailed guidance on how to implement every possible CM-related function.

Too Smart for Their Own Good

What has really impressed me is just how many CM "experts" are so smart that they can disregard the guidance in standards and frameworks without even reading them! The result can only be described as the CM equivalent of malpractice. Can you imagine paying a contractor to build your new home, only to discover that he was completely unaware of the building code requirements for that location? Imagine finding that your defense attorney never bothered to take the bar or getting treated by a doctor who had not passed his medical boards. In my opinion, that is precisely what I see happening with many CM consultants.

What Version of the Code Is in Production?

I have followed some very well-known consultancies who claim expertise in the CM arena only to observe that their customers did not know how to ascertain what version of their code was running in production. These clients spent thousands of dollars hiring CM experts who apparently never heard of the requirement for a configuration audit (to ascertain the version of the code running in an environment). Instead, they pointed back to their version control systems, including CVS, Subversion, and ClearCase, seemingly unaware that having a tagged baseline safely tucked away in a source code repository doesn't really help you definitively ascertain the exact version running in production. One "senior" CM consultant told me that he had stopped embedding version IDs in binary executables because he didn't want to take a chance on the ID being wrong! (So much for having a repeatable build process.) Much of what is missed by these consultants is due to ignorance because they

have never done their homework. It's time for this to change.

CM Certifications Based Upon Standards

I believe that the CM industry has reached a point of maturity where we must have standardized certifications that are based upon a combination of industry best practices and the guidance defined in industry standards and frameworks, along with confirmation of hands-on experience and specialized knowledge. I have seen a number of folks skeptical of the validity of this material, most notably the itSMF ITIL framework. Others vehemently disagree with what they see mandated in the CMMI. Many technology professionals are fiercely in favor of agile and lean practices, while others cite their own experience with trying to test code produced (actually half baked) from a time-boxed sprint. Regardless of the preferred process model, CM folks need to have studied and be able to verbalize their own recommended practices based upon the guidance described in industry standards and frameworks. I am fine with folks disagreeing with what's been written—but it's time for that view to be based upon actually reading and analyzing the material, while also keeping in mind that these practices are often a regulatory requirement.

SOX Anyone?

The ISACA Cobit framework has become the de-facto standard for compliance with section 404 of the Sarbanes-Oxley Act of 2002. Yet, I recall a colleague at a large bank telling me that they used to do Cobit. This colleague was apparently unaware that the bank had to implement these practices to comply with regulatory requirements. A short time later, I learned that the bank's audit findings described a lack of IT controls with not-so-surprising consequences.

Conclusion

It's time for us to raise the bar for CM in terms of establishing essential industry best practices in alignment with well-accepted standards and frameworks. It is also time to ensure that CM consultants and practitioners have the required expertise before representing themselves as CM experts. Standardized CM training is only the start. What we need now is certifications based, in part, upon the wisdom defined in industry standards and frameworks. There should also be an opportunity to specialize in specific functions (e.g., build engineering) and recognition of years of experience. Come to think of it, implementing CM best practices is very similar to building a new house, and we need to ensure that CM professionals adhere to the highest standards in all of their activities. {end}

index to advertisers

Display Advertising
advertisingsales@sqe.com

All Other Inquiries
info@bettersoftware.com

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

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