

QUALITY

Software and Testing

VOLUME 4, ISSUE 2
September 2006



Good and Practical Ideas for Testers

We asked some of the leading people in QA to give us some clever, innovative and practical ideas for Testers. Many of them responded.

Ideas by

James Bach
Cem Kaner
Rex Black
Scott Ambler
Duncan Card
Michael Bolton
Fiona Charles
Joe Larizza
Richard Borne

QUALITY Software and Testing

FEATURES

4 **Good, Innovative and Practical Ideas**

by Scott Ambler
James Bach
Rex Black
Michael Bolton
Duncan Card
Fiona Charles
Cem Kaner
Joe Larizza
Richard Bornet

21 **Quality Cost Analysis: Benefits and Risks**

by Cem Kaner

How to Submit Articles to Quality Software and Testing

Quality Software and Testing provides a platform for members and non-members to share thoughts and ideas with each other.

For more information on how to submit articles, please visit <http://www.tassq.org/quarterly>.

DEPARTMENTS

Editorial	Page 3
Humour: Cartoons	Page 22, 29
LITruisms	Page 5, 7
Events, Conferences, Education and Services	Page 30

TASSQ

Toronto Association of Systems and Software Quality

An organization of professionals dedicated to promoting Quality Assurance in Information Technology.

Phone: (416) 444-4168

Fax: (416) 444-1274

Email: tassquarterly@tassq.org

Web Site : www.tassq.org

QUALITY Software and Testing

Editorial

In our last magazine we had a topic of “What is Quality Software?” To answer this question, we had some of the leading lights in the QA community give their answers. The views spanned a wide range, from “It’s in the eye of the beholder” to “Rubbish, we can define quality and we all do.” That set of articles elicited much debate, and in some cases some strong emotions.

This pleased us to no end. We want this magazine to be a focus point for debate. We want to stimulate people and bring the subject matter alive. We want to make people think and stretch people’s minds. We also want this magazine to be one which takes a look at the future by providing new ideas and new visions.

Are we succeeding? I don’t know, but we are certainly trying.

In this issue, we wanted to write about good and practical ideas for testers. The easy answer to this question is to point testers to Cem Kaner’s book, "Testing Computer Software", and to Cem Kaner and James Bach’s book, "Lessons Learned in Software Testing". Whether you are new to testing, or have a lot of experience, both books are so full of useful ideas and solutions to problems that you can't help but get something you can use.

Then we decided to ask those authors if they had any more really good ideas, and they responded by giving us two very thoughtful and interesting articles.

In addition, we received some great pieces from authors who have a deep insight and commitment to the testing field. We want to thank previous contributors Rex Black and Michel Bolton, and new contributors Scott Ambler, Duncan Card and Fiona Charles. Joe and I have also added to the debate.

I hope you will agree with me that we have a series of articles which offer helpful advice to take with you when you test software.

We are also proud to be able to reprint Cem Kaner’s seminal article on “Quality Cost Analysis: Benefits and Risks”.

Our regular features include our humour section with our regular cartoons and LITruisms (Life and IT Truisms or LITruisms for short), which are some insightful thoughts for you to paste on your office wall. They are sprinkled around the magazine. If you have any to share with us, please send them along.

Finally, TASSQ is sponsoring some innovative and interesting presentations and courses. In October, Cem Kaner is speaking and at the end of November, Jim Hobart is putting on his excellent User Interface course. Rex Black is also coming in November. See the detailed descriptions at the back of the magazine.

We hope you enjoy the magazine. Please feel free to drop us a line; we would love to hear from you. And if you have an article where you share your ideas, send it in.

Richard Bornet
Editor-in-Chief

PUBLISHER

Joe Larizza

EDITOR-IN-CHIEF

Richard Bornet

ASK TASSQ EDITOR

Fiona Charles

**BOOK REVIEW
EDITOR**

Michael Bolton

Copyright © 2006

Toronto Association of
Systems & Software
Quality. All rights
reserved.

Good, Innovative and Practical Ideas

In the last magazine, we put out a call for articles. The question we asked was fairly vague. We did this on purpose so as not to steer anyone in a particular direction, but to get people talking about the topic.

Here is the text that we sent out:

“As stated above, we really want the magazine to be a must-read. The emphasis will be on the practical. The goal is to stimulate thought and discussion, and hopefully each issue will be a learning experience for the reader. We don't mind controversial opinions or heated debates; we want the subject matter to be alive.

We have selected our topic for the next issue: Creative, Clever and Practical Good Ideas for Testers.

The reason we chose this topic is two-fold. James Bach constantly reminds us that testers have to think and be clever. So what are some practical applications of this? Here is your chance to give us some really great examples.

The second reason we chose this topic is because of the feedback we receive about both the magazine and the meetings we put on. We hear that they are interesting, but too high level. One tester summed it for us when she said, "Give me something really useful I can use tomorrow when I am testing!" We have heard this comment on many occasions. So we want to give her and others an answer!

What we would really like from you, if you are willing, is some of those great creative and practical ideas for testers, and any other comments you would like to make about the subject matter. We think the readers would be very interested in any thoughts you have.

We thank Scott Ambler, James Bach, Rex Black, Duncan Card and Cem Kaner for their contributions.

We then supplemented their thoughts with our own. Joe Larizza, Michael Bolton, Fiona Charles and I entered into the debate by adding ideas.

Below are the responses we got. We have listed the authors in alphabetic order, just to make things simple, followed by Joe and myself.

Again, we are very grateful to the people who took the time and effort to write to us.

Richard Bornet

From Scott Ambler

Scott W. Ambler is Practice Leader Agile Development within IBM's Methods group. He is the author of several books, including the award winning Agile Database Techniques (Wiley 2003) and the recently released Refactoring Databases: Evolutionary Database Design (Addison Wesley 2006). He maintains the Agile Data site (www.agiledata.org) where he has shared a wealth of information about evolutionary and agile database development. His home page is www.ambysoft.com/scottAmbler.html

Test-Driven Database Design (TDDD)

Test-driven development (TDD) [1,2,3] is an evolutionary approach to development which combines test-first development (TFD) and refactoring [4]. With a test-first approach to development you write a test before you write just enough production code to fulfill that test. Refactoring is a disciplined way to restructure code where you make small changes to your code to improve your design, making the code easier to understand and to modify.

When an agile software developer goes to implement a new feature, the first question they ask themselves is "Is this the best design possible which enables me to add this feature?" If the answer is yes, then they do the work to add the feature. If the answer is no, they refactor the design to make it the best possible then they continue with a TFD approach.

There are four iterative steps of TFD:

1. **Quickly add a test.** You basically need just enough code to fail, typically a single test.
2. **Run your tests.** You will often need the complete test suite although for sake of speed you may decide

to run only a subset, to ensure that the new test does in fact fail.

3. **Update your production code.** Do just enough work to ensure that your production code passes the new test(s).
4. **Run your tests again.** If they fail you need to update your production code and retest. Otherwise go back to Step #1.

Why TDD?

There are several advantages to TDD:

1. It promotes a significantly higher-level of unit testing within the programming community.
2. It enables you to take small steps when writing software, which seems to be far more productive than attempting to code in large steps. For example, assume you add some new functional code, compile, and test it. Chances are pretty good that your tests will be broken by defects that exist in the new code. It is much easier to find, and then fix, those defects if you've written two new lines of code than two thousand. The implication is that the faster your compiler and regression test suite, the more attractive it is to proceed in smaller and smaller steps.
3. It promotes detailed design. Bob Martin [5] says it well "The act of writing a unit test is more an act of design than of verification. It is also more an act of documentation than of verification. The act of writing a unit test closes a remarkable number of feedback loops, the least of which is the one pertaining to verification of function". An interesting implication is that because unit tests provide other significant benefits to programmers, there's a much greater chance that they'll actually keep their "design" up to date.

A fair question to ask is: If TDD works for application development, shouldn't it also work for database development? My experience is the answer is yes. A test-driven database development (TDDD) approach provides the benefits of TDD, plus a few others which are database related. First, it enables you to ensure the quality of your data. Data is

an important corporate asset, yet many organizations suffer from significant data quality challenges (they also don't seem to have a strategy for database testing, if they've even talked about it at all). Second, it enables you to validate the functionality implemented within the database (as stored procedures for example) that in the past you might have assumed "just worked". Third, it enables data professionals to work in an evolutionary manner, just like application programmers.

The implication is that for TDDD to work, we must be able to refactor a database schema and regression test it effectively. Luckily both of these are fairly straight forward things to do.

Database Refactoring?

A database refactoring [6] is a simple change to a database schema that improves its design while retaining both its behavioral and informational semantics. Your database schema includes both structural aspects such as table and view definitions and functional aspects such as stored procedures and triggers. Examples of database refactorings include Rename Column, Apply Common Format, Migrate Method to Database, and Split Table and a catalog of over 65 refactorings at www.agiledata.org.

Database refactorings are conceptually more difficult than code refactorings: Code refactorings only need to maintain behavioral semantics, whereas database refactorings must also maintain informational semantics. Worse yet, database refactorings can become more complicated by the amount of coupling resulting from your database architecture. Coupling is a measure of the dependence between two items; the more highly coupled two things are, the greater the chance that a change in one will require a change in another.

Some project teams find themselves in a relatively simple, "single-application database" architecture, and if so they should consider themselves lucky because database refactoring is fairly easy in that situation – you merely change your database schema and update your application to use the new version of the schema.

LITruism

Here is a story I once heard. This concerns a university professor who attended a gathering where Dr. Albert Einstein was in attendance. He approached Dr. Einstein and said, "Dr. Einstein, I carry a little notepad around in my pocket and every time I have a good idea, I take out the notepad and I jot the idea down. In that way I don't forget it. What do you do when you have a good idea, Dr. Einstein?" Albert Einstein looked at the man in a perplexed manner, obviously thinking about the question. Then he turned to the professor and said, "You know, it is so very seldom that I have a good idea."

What is more typical is to have many external programs interacting with your database, some of which are beyond the scope of your control. In this situation you cannot assume that all the external programs will be deployed at once, and must therefore support a transition period during which both the old schema and the new schema are supported in parallel. This situation is more difficult because the individual applications will have new releases deployed at different times over the next year and a half.

Let's step through a quick example. You are about to implement a new requirement which involves working with the first names of customers. You look at the existing database schema for the *Customer* table and realize that the column name isn't easy to understand. You decide to apply the *Rename Column* refactoring to the *FName* column to rename it to *FirstName* so that the database design is the best one possible which allows you to implement the new requirement.

To do this you first introduce the *FirstName* column and a trigger which is required to keep the values in the columns synchronized – each external program accessing the *Customer* table will at most work with one but not both columns. At first, all production applications will work with *FName*, but over time they will be reworked to access *FirstName* instead. There are other options to do this, such as views or synchronization after the fact, but I find that triggers work best.

The *FirstName* column must also be populated with values from the *FName* column, which can easily be done with SQL code. After the transition period, you remove the original column plus the trigger. You remove these things only after sufficient testing to ensure that it is safe to do so. At this point, your refactoring is complete.

Database Testing?

The second part of TDDD is database regression testing [7]. I believe that there are two categories of database tests: interface tests and internal database tests. Interface tests validate what is going into, out of, and mapped to your database. If an organization is doing any database testing at all, my experience is that it is usually at the interface level. Internal database tests validate the internal structure, behavior, and contents of a database. These types of tests should validate:

- Scaffolding code (e.g. triggers or updateable views) which support refactorings
- Database methods such as stored procedures, functions, and triggers
- Existence of database schema elements (tables, procedures, ...)
- View definitions

- Referential integrity (RI) rules
- Default values for a column
- Data invariants for a single column
- Data invariants involving several columns

For database regression testing to work we need good testing tools. There are three important issues to consider when it comes to database testing tools:

1. You will need two categories of database testing tools (interface and internal) because there are two categories of database tests.
2. Testing tools should support the language that you're developing in. For example, for internal database testing if you're a Microsoft SQL Server developer, your T-SQL procedures should likely be tested using some form of T-SQL framework. Similarly, Oracle DBAs should have a PL-SQL-based unit testing framework.
3. You need tools which help you to put your database into a known state, which implies the need not only for test data generation but also for managing that data (like other critical development assets, test data should be under configuration management control).

To make a long story short, although we're starting to see a glimmer of hope when it comes to database testing tools, as you can see in the list below, we still have a long way to go. Luckily there are some good tools being developed by the open source software (OSS) community and there are some commercial tools available as well. Some database testing tools:

- [Data Factory](#)
- [Datatect](#)
- [DBUnit](#)
- [DTM Data Generator](#)
- [Mercury Interactive](#)
- [NDbUnit](#)
- [OUnit for Oracle](#) (being replaced soon by [Oute](#))
- [Rational Suite Test Studio](#)
- [SQLUnit](#)
- [TSQLUnit](#)
- [Turbo Data](#)
- [Visual Studio Team Edition for Database Professionals](#)
- [Web Performance](#)

In Conclusion

Just as agile application developers take a quality-driven, TDD approach to software development so can agile database developers. This requires the adoption of new techniques, in particular database refactoring and database regression testing. The agile community is raising the bar on the data community, are you ready for the coming changes?

References

1. Test Driven Development: A Practical Guide by Dave Astels
2. Test Driven Development: By Example by Kent Beck
3. Introduction to Test Driven Development by Scott W. Ambler
4. Refactoring: Improving the Design of Existing Code by Martin Fowler
5. Agile Software Development Principles, Patterns, and Practices by Robert C. Martin.
6. The Process of Database Refactoring by Scott W. Ambler
7. A Roadmap for Regression Testing Relational Databases by Scott W. Ambler

From James Bach

James Bach (<http://www.satisfice.com>) is a pioneer in the discipline of exploratory software testing and a founding member of the Context-Driven School of Testing. He is the author (with Kaner and Pettichord) of "Lessons Learned in Software Testing: A Context-Driven Approach". Starting as a programmer in 1983, James turned to testing in 1987 at Apple Computer, going on to work at several market-driven software companies and testing companies that follow the Silicon Valley tradition of high innovation and agility. James founded Satisfice, Inc. in 1999, a tester training and consulting company based in Front Royal, Virginia.

Rapid Testing For Rapid Maintenance

A correspondent writes:

"I have a test management problem. We have a maintenance project. It contains about 20 different applications. Three of them are bigger in terms of features and also the specs that are available. I am told that these applications had more than 1-2 testers on each of these applications. But in this maintenance project we are only 6-7 testers who are responsible to do the required testing. There will be a maintenance release every month and what it will deliver is a few bug fixes and a few CRs. What those bugs and CRs would be is not known in advance. Could you please suggest how to go about managing such kind of assignment?"

Okay, this is what I would call a classic rapid testing situation: lots of complexity, not a lot of time or people.

In tackling this problem, first I would analyze the context. Many images leap to mind when I hear words like "application" and "maintenance" and "project", but these images may be mistaken. I typically refer to a diagram called the Context Model (<http://www.satisfice.com/tools/satisfice-cm.pdf>) to help me think this through. In this case, the factors foremost in my mind are the following:

Consider your mission. What specifically do your clients expect from you? Do they need you to find important bugs quickly, or are there also other requirements such as the use of certain practices and tools, or the production of certain documentation? (*This is important for two reasons: it may be that there is no way for you to achieve the mission in your situation, in which case you'll need to renegotiate it; also, clarity of mission helps you avoid doing anything that isn't necessary to do. In your situation, you don't have the luxury of putting your process on cruise control.*)

Consider the quality goal. How important is reliability? What if you miss a bug? Are the customers of this product very sensitive to problems, or are they tough computer geeks who don't mind a little crash once in a while? Do these products have the capacity to harm anyone if they fail? (*This is important because higher risk justifies and demands a more expensive and meticulous testing approach.*)

Consider your team. Are your testers interested and able to do exploratory testing under pressure? Do they understand the products already or do they need to climb the learning curve rapidly? Also, are there any people not on the testing team who might be able to help test, such as tech support or documentation people? Even friendly users in the field might be able to help. (*This is important because I suspect that this situation will call for some creative and intensive testing by skilled testers. Testers who prefer to be spoon-fed rote test cases will probably be miserable.*)

Consider your test lab and materials. Do you have equipment, tests, test data, automation, or anything else that will help you retest the products over time? For instance, do you have a test coverage outline, which is much more flexible than a set of test cases? (*This is important because if you have existing tests or test documentation, then you have to figure out if it's actually helpful. Possibly the old tests weren't very good. Maybe the existing automation is broken and not worth fixing. Take stock of your testing infrastructure.*)

LITruism

Do you realize that the person or persons who invented a certification program were never certified?

Consider the overall testability of the products. Are these applications fundamentally easy to test or difficult to test? Can the programmers make them easier to test by adding logging, or scriptable interfaces? How much is there to test? Are the products modular, such that a problem in one part won't necessarily affect other parts? *(This is important because testability is critical to the consistent success of meagerly staffed test project.)*

Consider stability. Are these applications brittle or supple? Is it likely that the maintenance process will create more problems than it fixes? Are there robust unit tests, for instance? *(This is important because when programmers are working with a stable code base and when they do reasonable unit tests, the probability of a bad bug reaching you is lower, and it will require less effort for you to reach a level of comfort about the status of the product during your testing.)*

Consider the cohesiveness of the product line. Are these twenty applications completely separate, or do they interact and share data or components? Is there a flow of data among the individual applications? *(This is important because a cohesive, highly integrated set of applications can be tested together and against each other. The output of one may validate the output of another. The flipside of that is the increased chance of a fix in one causing a problem in the other.)*

Consider the availability of good oracles. How easy is it to validate that the outputs of a product are correct, or at least, sane? Will a bad problem also be an obvious problem, or is it possible that bad problems can occur in testing, but not be noticed by the tester? *(This is important because you may need to invest in tools, training, or reference materials to assure that you will spot a problem that does, in fact, occur.)*

Consider the development process. How and when will decisions be made about what to fix and what to change? When will you get the new builds? Will testing have a voice in this? Are you in good communication and reputation with the programmers? *(This is important because you can test the product better when fixes are made with testing requirements in mind. You won't be blindsided if you are involved in those discussions.)*

After looking the project over to figure out the general lay of the testing problem, I would have in mind some issues to raise with management and the programmers. In any case, the following is my first impression based on your question. You can consider this my default mental template for dealing with this kind of test project, subject to amendment based on what I discover from learning about the issues, above:

- The solution is probably not heavily documented manual test procedures. Those are expensive to

produce, expensive to maintain, and actually do little to help test a complex product. They are favored by managers who don't understand testing and large consulting companies who get rich by exploiting the ignorance of said managers. See almost any military software test plan for an illustration of this sad principle. See any test plan afflicted with *Sarbanes-Oxley* syndrome.

- The solution probably does not involve comprehensive test automation, unless you have a wonderful programmer on your team, and a highly scriptable set of products that don't change very much. However, if I did have a programmer on staff, I would look for non-comprehensive, "fire and run", agile test automation opportunities. See my paper on this for details (<http://www.satisfice.com/articles/agileauto-paper.pdf>).
- Make a set of test coverage outlines that address all your products. A test coverage outline is literally an outline of the structures, functions, and data that you might need to cover when testing those products. Start with something no more than, say, two pages long per application. Use Notepad, Excel, or some other very low formatting method of documenting it. Exotic formatting just slows you down. Use these outlines to plan and report your testing on the fly. (Use the Satisfice Test Strategy Model, available at <http://www.satisfice.com/satisfice-tsm-4p.pdf>, to help produce these outlines.)
- Make a list of risks for each application. By risks, I mean, what kind of bugs would be most worrisome if they were to creep into those products? For instance, for Microsoft Excel, I would think "math error" would be high on the list. I would need to have tests for mathematical correctness. (The Satisfice Test Strategy Model will help here, too.)
- Make a one or two-page outline of your test strategy. This is a list of the kind of test activities you think need to be done to address the major risk areas. Be as specific as you can, but also be brief. Brevity is important because you need to go over this strategy with your clients, and if it's just a page or two long, it will be easy to get a quick review. (See examples of test documentation and notes in the appendices of my RST class, at <http://www.satisfice.com/rst-appendices.pdf>)
- In producing your test strategy, look for test tools, techniques, or activities that might be able to test multiple products in the same activity. If your products are part of an integrated suite, this will be much easier. I would focus on scenario testing as a principle test development technique (see Cem

Kaner's wonderful article on that, which you also can find in the appendices of my RST class) and given that this is a maintenance project, I suspect that parallel testing (whereby you test a product against its earlier version in parallel) will also be a bread-and-butter part of the strategy.

- Seek variety and freshness in your testing. A risk with maintenance testing, especially under pressure, is that you lapse into following the same paths in the same way with the same data, cycle after cycle. This makes testing stale and minimizes the chance that you will find important bugs. Remember, you can't test everything, so that's why we need to take fresh samples of product behaviour each time, if that's feasible. Yes, you need to cover the same aspects of the product, again and again, but try to cover them with different tests. Use different data and do things in a different order. It also helps to use paired exploratory testing (two people, one computer, testing the same thing at the same time). Pairing injects a surprising amount of energy into the process.
- Although variety is important, it may be important for some tests to be repeated each test cycle in pretty much the same way. This can be useful if it's difficult to tell whether a complicated set of output is correct. By keeping some tests very controlled and repetitive, you can directly compare complicated output from version to version, and spot bugs in the form of subtle inconsistencies in the output. Hence, you may need to prepare some baseline test data and test lab infrastructure and preserve it over time.
- Brief your management and developers as follows: *"I want to serve you by alerting you to every important problem in these products before those problems get into the field. I want to do this without slowing you down. If I'm going to do a good job with such a small staff, then I need to know about fixes and changes as early as possible. I need to be involved in these decisions so that I can let you know about possible difficulties in retesting. If you are careful about the way to change the product, and if you share with me details about the nature of each fix, my team can test in a much more targeted, confident fashion. In general, the more testable this product is, the more quickly I can give you a good report on the status of each change, so keep that in the back of your mind as you go. Meanwhile, I commit to giving you rapid feedback. I will do my best to keep testing out of the bottleneck."*
- As for your test management strategy, you need to develop the equivalent of a two-minute drill for each product; a well organized test cycle. When a fix or

fixes come down the pipe, you should have someone testing that specific change within a few minutes, and you should have preliminary test results within a few minutes to an hour after that. You achieve this by the approach of exploratory testing using skilled testers. Assign the products to the testers such that each tester has a few products that they are required to master. Everyone should be up to speed on each product, but for each product there should be a "test boss" who is responsible for coordinating all testing effort. You can only make someone a test boss if they are reasonably skilled, so depending on your team, you may be the boss of all of it.

- If your products are not easy to install, you need to work on that problem. You can't afford to waste hours after the build just trying to get the products up and running. Remember: two-minute drill. Hup hup. When a product is released to your team, they should start exploring it like snakes on plane.
- You must develop skilled testers in your team if you don't already have them. I'm talking about tactically skilled testers: people who are comfortable with complexity, understand what test coverage is and what oracles are; people who can design tests to evaluate the presence of risk; people who can make observations and reason about them. I suggest using Cem Kaner's Black Box Software Testing video lectures to help you develop your folks. If they've never taken a testing class, my class might help, or Elisabeth Hendrickson's Creative Software Testing class would also work well. I have lots of written training materials on my site, too. See also the various articles on my blog, or for that matter, see my book *Lessons Learned in Software Testing*.
- If there isn't already a bug triage process in your group, establish one. I suggest that you run those meetings, unless the program manager insists. You need to get good at making the case for bug fixes. I have prepared a cheat sheet to help test managers with this, you can find it as an appendix to my process evolution article at <http://www.satisfice.com/articles/process-evolution.pdf>, but it's also included in the RST class appendices.
- Establish a testing dashboard either online or, as I prefer, on a whiteboard. It should be something that expresses, at a glance, your testing status, and does so in a way that answers the most important management questions. See my presentation on a Low Tech Testing Dashboard on my site at <http://www.satisfice.com/presentations/dashboard.pdf>

From Rex Black

With almost a quarter-century of software and systems engineering experience, Rex Black is President of RBCS, Inc., a consulting, assessment, outsourcing, and training company, providing industry leadership in software, hardware, and systems testing, for about fifteen years. RBCS has over 100 clients spanning 25 countries on six continents. Learn more about Rex and RBCS at www.rexblackconsulting.com.

Two Weeks to Better Testing

If you're like most testers, you are looking for practical ways to improve your testing. You are also time constrained and need to make improvements quickly that show fast results. Here I present three practical ideas which you can put into action in two weeks and which will make a noticeable difference.

Get Hip to Risk-Based Testing

I have a simple rule of thumb for test execution: Find the scary stuff first. How do we do this? Make smart guesses about where high-impact bugs are likely. How do we do that? Risk-based testing.

In a nutshell, risk-based testing consists of the following:

1. Identify specific risks to system quality.
2. Assess and assign the level of risk for each risk, based on likelihood (technical considerations) and impact (business considerations).
3. Allocate test effort and prioritize (sequence) test execution based on risk.
4. Revise the risk analysis at regular intervals in the project, including after testing the first build.

You can make this process as formal or as informal as necessary. I have helped clients get started doing risk-based testing in as little as one day, though one week is more typical. For more ideas on how, see my article, "Quality Risk Analysis", at www.rexblackconsulting.com/Pages/Library.htm, my book or Rick Craig's book on test management, *Managing the Testing Process* and *Systematic Software Testing*, or my book on test techniques, *Effective and Efficient Software Testing*.

Whip Those Bug Reports into Shape

One of the major deliverables for us as testers is the bug report. But, like Rodney Dangerfield, the bug report gets no respect in too many organizations. Just because we write them all the time doesn't mean they aren't critical—quite the contrary—and it doesn't mean we know how to write them

well. Most test groups have opportunities to improve their bug reporting process.

When I do test assessments for clients, I always look at the quality of the bug reports. I focus on three questions:

1. What is the percentage of rejected bug reports?
2. What is the percentage of duplicate bug reports?
3. Do all project stakeholder groups feel they are getting the information they need from the bug reports?

If the answer to questions one or two is, "More than 5%," I do further analysis as to why. (Hint: This isn't always a matter of tester competence, so don't assume it is.) If the answer to question three is, "No," then I spend time figuring out which project stakeholders are being overlooked or underserved. Recommendations in my assessment report will include ways to get these measures where they ought to be. Asking the stakeholders what they need from the bug reports is a great way to start—and to improve your relationships with your coworkers, too.

Read a Book on Testing

Most practicing testers have never read a book on testing. This is regrettable. We have a lot we can learn from each other in this field, but we have to reach out to gain that knowledge.

(Lest you consider this suggestion self-serving, let me point out that writing technical books yields meagre book royalties. In fact, on an hourly basis it's more lucrative to work as a bagger at a grocery store. Other benefits, including the opportunity to improve our field, are what motivate most of us.)

There are many good books on testing out there now. Here's a very small selection:

What You Want to Learn	Books
General tips and techniques for test engineers	<i>Effective and Efficient Software Testing</i> , Rex Black <i>A Practitioner's Guide to Software Test Design</i> , Lee Copeland
Object-oriented testing	<i>Testing Object-Oriented Systems</i> , Robert Binder
Web testing	<i>The Web Testing Handbook</i> , Steve Splaine
Security testing	<i>Testing Web Security</i> , Steve Splaine <i>How to Break Software Security</i> , James Whittaker
Dynamic test strategies and techniques	<i>Lessons Learned in Software Testing</i> , Cem Kaner et al <i>How to Break Software</i> , James Whittaker

Test management	<i>Managing the Testing Process</i> , Rex Black <i>Systematic Software Testing</i> , Rick Craig
Test process assessment and improvement	<i>Critical Testing Processes</i> , Rex Black <i>Practical Software Testing</i> , Ilene Burnstein
ISTQB tester certification	<i>Foundations of Software Testing</i> , Dorothy Graham et al <i>Software Testing Foundations</i> , Andreas Spillner et al <i>The Testing Practitioner</i> , ed. Erik van Veenendaal

I have read each of these books (some of which I also wrote or co-wrote). I can promise you that, if you need to learn about the topic in the left column of the table, reading one of the books in the right column will repay you in hours and hours saved over the years, as well as teaching you at least one or two good ideas you can put in place immediately.

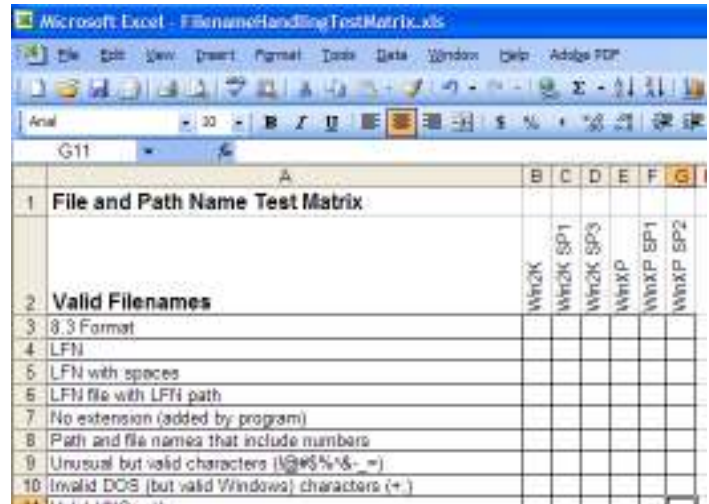
From Michael Bolton

Michael Bolton provides worldwide training and consulting in James Bach's Rapid Software Testing. He writes about testing and software quality in Better Software Magazine as a regular columnist, has been an invited participant at the Workshop on Teaching Software Testing in 2003, 2005, and 2006, and was a member of the first Exploratory Testing Research Summit in 2006. He is Program Chair for the Toronto Association of System and Software Quality, and an active member of Gerald M. Weinberg's SHAPE Forum. Michael can be reached at mb@developsense.com, or through his Web site, <http://www.developsense.com>

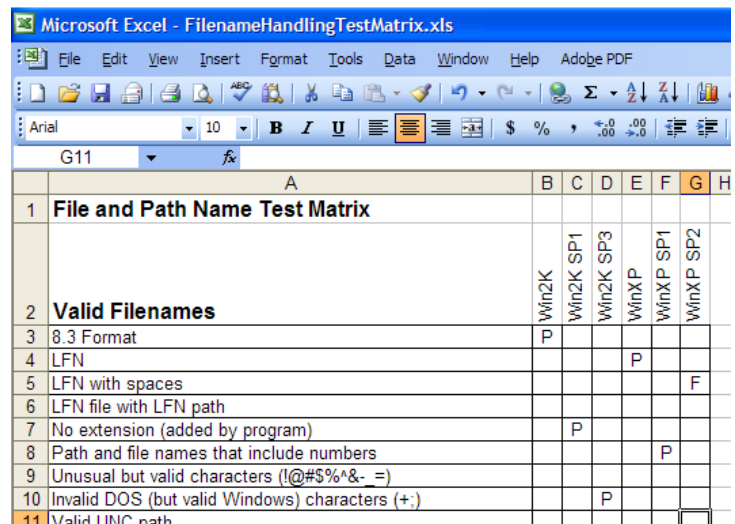
Conditional Formatting as a Testing Tool: Test Matrices and Blink Tests

Excel's conditional formatting is a handy feature that allows the user to see certain things more clearly. I've found it to be useful for a number of testing tasks.

Conditional formatting changes the display of data in a given cell or range of cells based on some criterion or formula. This allows us to see patterns of information at a glance. The test matrix is a powerful way of achieving and visualizing test coverage. A typical way of using a test matrix is to create a table that has a test idea on one axis and an application of that idea on another. Here's an example:



In this example, we have test conditions on the Y axis, and operating systems on which we'll apply those test conditions on the X axis. (LFN stands for "long file name".) Rather than doing a test for each of eight ideas and each of six operating systems, OS (which would take 48 tests in this example), we'll do an instance of each test in each column and each row. A heuristic is a fallible method for solving a problem. In this case, our heuristics are that a given test idea will reveal a bug on all operating systems, or that a given operating system will experience problems with multiple tests, and in general, the more we scatter entries over the table, the more likely we are to find a problem.



Here we've performed half a dozen tests, one for each operating system. We've inserted a P for Pass, and an F for "Fail" in each column. In addition, if we see a passing test that shows interesting behaviour, (though not technically a failure), we'll insert a W, for "Warning". Let's use conditional formatting to make the information pop out a little more clearly. Choose Excel's Format / Conditional Formatting menu:



Here we've instructed Excel to turn each cell green if there's a P within, red (with white text) if there's an F, and yellow if there's a W. Now the chart looks like this:

	A	B	C	D	E	F	G	H
1	File and Path Name Test Matrix							
2	Valid Filenames							
3	8.3 Format							
4	LFN							
5	LFN with spaces							
6	LFN file with LFN path							
7	No extension (added by program)							
8	Path and file names that include numbers							
9	Unusual but valid characters (/@%\$%^&* -)							
10	Invalid DOS (but valid Windows) characters (+)							
11	Valid UNC path							

The data pops out at us with significantly more impact. The next thing we would do would be to investigate to see if long file names with spaces are broken generally, or if there are other problems under Windows XP Service Pack 2.

Another use for conditional formatting is something that James Bach and I call a *blink oracle*. (An oracle is a heuristic principle or mechanism that we use to recognize a problem). A *blink test* is a test in which we use a blink oracle to defocus and remove some information in order to see patterns that might be otherwise invisible.

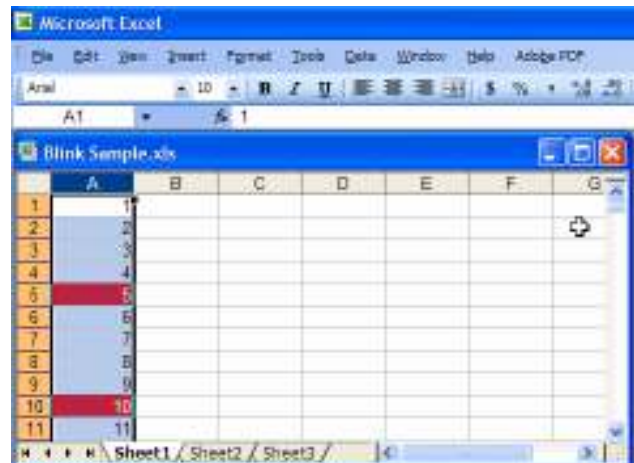
In the following (fairly contrived) example, I've used conditional formatting on a set of data. In a regular sequence of index numbers starting from one, I'd expect every fifth number to be divisible by five. If I had to check all of the numbers in the table, the process would be laborious and error-prone. Instead, I'll start by using conditional formatting to highlight the first cell if it's divisible by five.



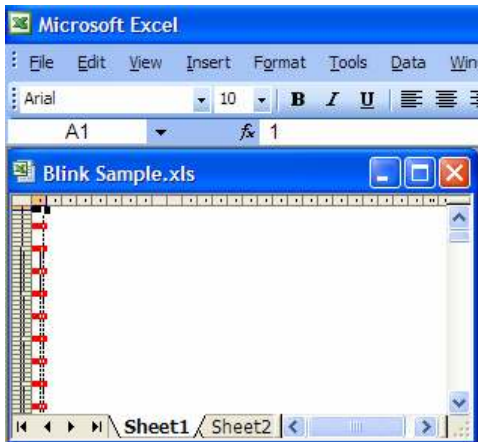
Then I'll copy that formatting condition to all of the cells in the column. Here's a quick way to do that: use Excel's Format Painter tool; it's the button on the toolbar that looks like a paintbrush.



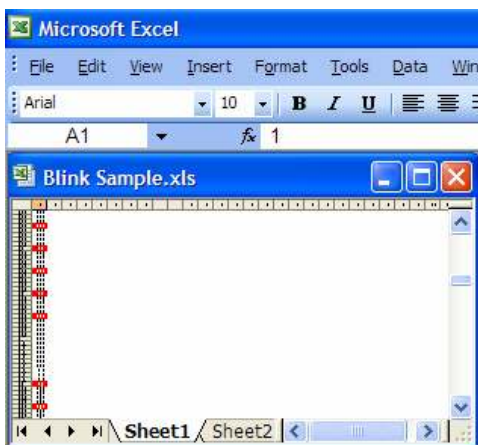
Click on the cell with the format you like; click on the paintbrush, then click on the cell, range, row, or column to apply the formatting.



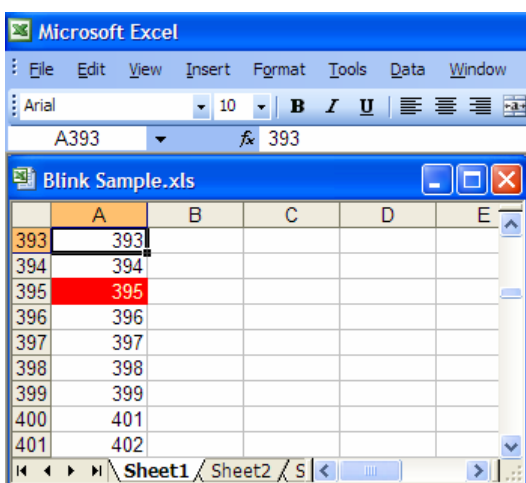
Here's the blink oracle part: I zoom the sheet down to 15% size. I can't see the data at all any more, but I can see the patterns in it.



As one might expect, since every fifth cell is divisible by five, there's a nice, regular-looking pattern to the data. But look what happens when I scroll down through the column quickly:



Wow! There's a break in the pattern! Let's zoom in on that:



Sure enough, there's a break in the sequence—the number 400 is missing. Our blink oracle has helped us to spot an

inconsistency. If I had inspected this listing by hand, it would have been easy to miss a single number.

As I noted above, this example is pretty trivial, but you can use the principle in many ways by using more elaborate conditional formulas and applying them to other kinds of data, such as records or log files. Apply conditional formatting to cells that contain a particular error message string, zoom out, and look for consistencies and inconsistencies in the data. Use formatting to highlight records that have something in common, and see what strikes your eye. Again, the general idea of a blink test is to expose patterns of similarities and differences by ignoring some specific information and paying attention to more general visual patterns. Excel's conditional formatting is one way for us to take advantage of our innate human pattern recognition skills.

From Duncan Card

Duncan Card is a Senior Technology Partner at the national law firm of Bennett Jones LLP in Toronto. He has been cited as one of the three most frequently recommended technology lawyers in Canada by Lexpert Magazine, and he is included in Thomson's The Leading 500 Lawyers in Canada (from 1997- 2006), in Woodward/White's, 2006 edition of The Best Lawyers In Canada, and in Euromoney's Guide To The World's Leading Technology Lawyers.

His book, Information Technology Transactions can be found at www.carswell.com

In the context of software development projects, I was recently asked what separated good risk managers from the truly great risk managers. My answer was that the truly great risk managers routinely step outside the box of accepted practice guidelines to identify and manage the actual causes of software development risk. Only that additional step of risk analysis and awareness will allow those managers to create effective strategies that avoid or mitigate the actual risks that threaten project success. When that very practical examination is undertaken, software project managers acquire both a better awareness of the actual risks that threaten the success of software projects and the strategies to manage those risks.

Here is an example. The security of a software development's environment will be high on any "best practice" risk management list. Although the usual security measures will likely include electronic and physical security protections (like passwords, access cards, biometrics, and dial-in pass codes), a critical examination of the actual causes of unauthorized access to the software development projects will reveal important causes of security breach that electronic and physical security strategies can't address. How do I know? In the mid 1990's, I asked the then General Counsel of the US'

National Security Agency, Richard Sterling, what he thought the NSA's most important security tool was. His answer surprised me. Without any hesitation whatsoever, he told me that the NSA's Employment Assistance Program was, by far, the most important security measure in its arsenal. Why? Simply because human behavior, due to frailty or imperfection, whether temptation or illness, is a leading cause of any project risk — even those of the NSA.

Even though hard statistics are hard to come by, I think that it is pretty fair to say that a significant percentage of security breaches that harm software development projects are committed by those who already have the pass codes and who otherwise have "the keys to the safe". Indeed, a significant number of crucial programming mistakes or omissions, sabotage occurrences, theft of valuable trade secrets, and even vandalism are committed by employees who have personal issues that need attention and care, such as mental health challenges, or drug, alcohol or gambling addictions (sometimes all three), or who have serious family or financial problems that can have a devastating affect on their judgement and behavior (even from the sleep deprivation alone). Often, those darker employment realities are ignored or swept under the workplace carpet. But in any given population, those problems do exist and they can have a serious impact on the welfare of the entire organization, let alone a software development project, if they are ignored. Once the brutal realities of those problems are faced as both human tragedies in need of care and as the security risks they are, then the onus is on the truly great risk managers to develop strategies that will proactively either avoid or mitigate those specific risks.

Over the course of my practice I have relied on Richard Sterling's advice many times, and even extrapolated it into other aspects of the "human risk factor". I routinely advise managers of large software development projects to work closely with experienced human resource managers to consider a wide range of practical risk management strategies, which may include: employment screening, including bonding applications, reference checks, and "employee fidelity" (the insurance industry term for employee honesty) enquiries; voluntary medical exams or drug testing programs, where such procedures are a condition of a high security position (these may be difficult to implement for incumbent employees); regular employee interviews by trained mental health professionals; so-called "whistle blower" support programs, where an employee who knows of the personal difficulties or challenges of another employee is encouraged to come forward so that the small (but troubling) problems of a colleague can be caught before they become big problems for both the individual and the company (e.g. Internet pornography or online gambling addictions are often spotted first by colleagues); and especially, an Employee Assistance Program that is provided by independent service providers, free of charge, and that will remain entirely confidential.

Again, the above example is only illustrative of the broader principle — that truly great risk management strategies require a practical appreciation of the actual risks that threaten the success of software development projects. Once those risks are identified, then those managers must take creative and proactive steps to address those specific risks. Those steps may include changing the way that the development project is being conducted or it may mean that managers must address matters that are somewhat external to the project, such as reaching out with a hand of care and assistance to employees. However, the example I used is only one example of that "best practice" methodology. Basically, no risk manager can address risks that they don't understand. It starts with rolling up one's "risk due diligence sleeves" and assessing the true causes for software project failure. Only when those are identified can those managers go to the next step of practically and efficiently managing those risks to promote software project success. Remember Albert Einstein's famous axiom, "The level of awareness to solve a problem is greater than the level of awareness that created the problem."

From Fiona Charles

Fiona Charles is a test project manager, writer and consultant, based in Toronto. With over 25 years experience in systems development and integration projects, she designs and implements practical software test processes, and manages testing at the program level on multi-project integrations for clients in a variety of industries, including retail, banking, financial services, telecommunications and health care. Fiona specializes in managing large-scale systems integration tests, and teaching others how to do it. She is the TASSQ Communications Chair.

Here's a quick tip. When I'm preparing for a systems integration test, I use different coloured highlighters to draw possible test paths on the architecture diagrams. It's the quickest and simplest way I've found to model the highest level of the test. And of course, it's easy to print out several copies of the diagram and try out different versions.

The marked-up diagram makes a simple graphic communication tool, to which most people respond enthusiastically. I can walk it around and get quick feedback and buy-in. I pin it up over my desk, and I often find that clients and project managers ask for colour photocopies so they can pin it up too.

The most useful diagrams are those that show all the data flows into, out of, and between systems. Often, these don't exist on integration projects when I get there. In that case, I draw them myself, working with knowledgeable people from each development or support team to identify all the ways data comes into their system, where it comes from, how it's

transformed, all the ways it goes out, and where, along with frequency, timing, and interface mechanisms. Then I assemble all the partial drawings into one or more comprehensive diagrams and get the teams to review.

This isn't a trivial task, but the diagram is essential to devise a good test strategy, and it usually becomes a valuable artefact for the other teams on the project. It's also a great way to learn about the systems and start building relationships with the teams we'll be working with during test execution.

From Cem Kaner

Cem Kaner, J.D., Ph.D., is Professor of Software Engineering at the Florida Institute of Technology. His primary test-related interests are in developing curricular materials for software testing, integrating good software testing practices with agile development, and forming a professional society for software testing. Before joining Florida Tech, Dr. Kaner worked in Silicon Valley for 17 years, doing and managing programming, user interface design, testing, and user documentation. He is the senior author of Lessons Learned In Software Testing with James Bach and Bret Pettichord, Testing Computer Software, 2nd Edition with Jack Falk and Hung Quoc Nguyen, and "Bad Software: What To Do When Software Fails" with David Pels.

Dr. Kaner is also an attorney whose practice is focused on the law of software quality. Dr. Kaner holds a B.A. in Arts & Sciences (Math, Philosophy), a Ph.D. in Experimental Psychology (Human Perception & Performance: Psychophysics), and a J.D. (law degree).

Visit Dr. Kaner's web sites: www.kaner.com and www.badsoftware.com.

From 1995-2000, most of my income came from commercial training.

Despite what seemed to be a limitless market for software testing courses, I decided to back away from commercial training because I didn't think it offered my clients enough value for the money.

The problem is that the short course is good for exposing people to an avalanche of new ideas but not good at helping people evaluate them or develop skills applying them. It is also good for introducing ideas that connect well to what people know, but not for shaping breakthroughs in their thinking and practice.

The problem that I see in our field is *not* that we don't all agree on terminology or on a group of "best practices." The

solution to our field's problems is *not* more training (let alone, certification) in a set of basics that were the same old basics 25 years ago.

The problem is that productivity of our development partners (programmers) has soared, while ours has grown incrementally.

- Some readers will remember back when a long program was 10,000 lines of code and probably written in COBOL—a language intentionally designed so that even nonprogrammer managers could read it. An individual tester could read an entire program, identify all the variables and use that knowledge as a basis for test planning. Back then it didn't take much more time to lovingly handcraft a set of tests than to handcraft the code to be tested.
- Today, my cell phone has over 1,000,000 lines of code. Generations of change in programming practice have made it possible to snap together huge programs in relatively short times. In contrast, the techniques we most often focus on in testing are still the ones developed in the 1970's or modest partial automations of them. Given a constant testing budget, every year we have a proportionally smaller impact on the code we work with, because the codebase expands so quickly.

As a trainer, I realized that a key difference lay in the quality of education of programmers. They had university degree programs focused on their work. New courses that focus on new paradigms are routine enhancements to university curricula. In contrast, there were no undergraduate programs focused on software testing and few graduate schools were interested in it. When I helped clients interview for staff, new graduates were as clueless about testing in 1999 as when I started hiring testers in 1983. Test management candidates were just as bad—the majority had never read a book on testing, let alone taken a course (even a three-day short course). That's not a foundation for change.

I decided to go back to university as a professor, even though that meant taking a big pay cut, to work on the educational issues. I followed three vectors:

- **Create a degree program in software testing.** Florida Tech let me chair its CS curriculum committee to explore this. **Surprisingly, the amount of actual coursework and project work available to devote to testing was limited.** We decided—I believe correctly—that a university degree in testing should also qualify its graduates for a programming career and so we designed ours to meet accrediting standards in Computer Science or Software Engineering. As a result, there was only room for a few courses on testing, human factors, customer satisfaction, and project economics. **What we**

realized was that most training for most testers would be on-the-job, even if we created the most test-intensive degree program we could. By the way, we eventually chose not to offer a testing degree. As we interviewed people who would hire our students, we realized that a testing degree would tend to lock our graduates into narrow career paths, so we decided instead to offer an unusually test-intensive software engineering program.

- **Develop new courses in testing.** The most interesting of these is an introduction to Java programming that requires first-year programming students to do everything test-first. Another is a programmer testing course that teaches test-first and API-level test techniques (for new code and maintenance work) to test-interested programming students (see <http://www.testingeducation.org/articles/ExperiencesTeachingTDD.pdf>). *I'll write more about these in later articles.*
- **Develop a new style of commercial education that exports the strengths of university instruction back to on-the-job training.** This is the focus of this article.

University education differs from commercial training in several important ways:

- **University students expect to develop skills.** They expect to learn how to do things and to get good at it. Think back to your course in Calculus, for example, or to any programming course. You didn't just learn definitions and descriptions of procedures. You learned how to work with them, and how to make your own stuff using them.
- **University students expect to spend a lot of time practicing the material they learn.** They do homework. They study for exams. Contrast this with the three days you spend in the commercial course. At night, do you do homework or do you do email and catch up on work you were supposed to get done that day at your job?
- **University students get coaching in their new skills and ideas.** They have labs and teaching assistants who show them how to do things and give feedback when the student has trouble doing them. Some courses do this formally with labs, others less formally, but good courses provide support structures for developing new skills.
- **University students get detailed feedback on their work.** Students submit homework and exams. They get grades. Sometimes they flunk. They value homework that prepares them for exams, even if the homework gets tedious, because they want the feedback they'll get from the homework and they want to improve enough to do well on the final exams or assignments.

- **Courses last a long time.** Students have time to practice, submit work, get feedback and change what they do based on the feedback, because the course is spread over three or six months.

So, how can we build time, coaching, and feedback into the commercial course?

The three-day course model doesn't work well for this. There just isn't enough time.

Anything but the short-course model doesn't work well for traveling consultants. The cost of travel is too high to fly a consultant/trainer into town for a short class every Tuesday for six months.

The new model will be up soon (probably by September, 2006) at <http://www.satisfice.com/moodle> – in the meantime, you can view most of the course components at <http://www.testingeducation.org/BBST>

Descriptions of the approach, for academic teaching, are at <http://www.kaner.com/pdfs/kanersloan.pdf> and <http://www.kaner.com/pdfs/kanerfiedleractprint.pdf>

For live instruction, the new course structure works like this:

- The lectures are on video. Students watch them before coming to class.
- Students take an open-book multiple choice quiz before coming to class, to help reinforce the basic concepts.
- Class time is spent on discussions and labs—puzzling through the new ideas or applying them, rather than encountering them for the first time.
- New ideas are developed further with preparation of homework and practice for exams.

There is no reason this has to be restricted to a university. You can do it at your own company.

The videos and instructional materials are available for free, under a Creative Commons license.

Imagine a structure like this:

- Tuesday night, a group of testers watch a lecture on video. Maybe they do this at work, maybe at home.
- Wednesday at noon, they meet with the learning-group leader. This might be the test manager or a test lead. It might be a local consultant or an in-house trainer. This person leads a discussion of the material from the night before. *Is this stuff applicable to our company? Do we already know how to do it? Do we understand it? What could we do this week to try to apply it to our current projects?*

- Over the week, the students try to apply it to their current project. The learning-group leader drops by from time to time asking how it's going, maybe sharing experiences from others who are trying to apply this to some other product or other part of this product. Some testers pair up, applying the technique together to some aspect of what they're testing.
- Next Monday, they meet to discuss how the application went. Maybe they decide this technique is not applicable at their company. Maybe they decide it is very cool. Maybe they decide they need another week of practice. If so, they postpone the next video, but might do some additional reading.
- Next Tuesday night (unless they decided to postpone this), the testers watch the next video, setting up the next week's work.

This is not a short course. It might take a year to go through a full course. (My one-semester course involves 45 instructor-contact hours with students.) But at the end of the year, people know how to do what they've learned, they have experience based opinions about the things they've learned, and they've used them on their projects—probably doing better jobs on those projects, over time, as they applied new skills and continued to apply the ones that seemed most relevant / valuable.

You can do this today with the materials at <http://www.testingeducation.org/BBST> (some companies are doing it).

You'll be able to do it a little more easily with the next generation of support at <http://www.satisfice.com/moodle> – and that will either be free or with paid support (your choice) from a consultant who might give online guidance, online grading feedback, or make visits to your company. (It's like linux—the software is free, and you can find service providers for a wide range of custom-for-you support, if you want it.)

I've spent an enormous amount of time developing this approach over the last 6 years. The videos alone represent about 2000 hours of scripting, videotaping and editing.

My intuition is that this hybrid has the potential to become the instructional engine for teaching the advanced skills in software testing and test automation that we need to develop as a field.

If you use them, I would very much appreciate hearing about your experiences.

This article is based on research that was supported by NSF Grant EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers." Any opinions, findings and conclusions or recommendations expressed in this material are those of the

author(s) and do not necessarily reflect the views of the National Science Foundation.

From Joe Larizza

Joseph (Joe) Larizza is a Quality Manager, for The RBC Dexia Investor Services. He is a member of the Board of Directors for the Toronto Association of Systems and Software Quality and is an Advisor to the Quality Assurance Institute. He is a Certified System Quality Analyst and holds a Bachelor of Arts Degree in Economics.

One of the best QA experiences I have had recently is sharing ideas about the future with my fellow QA peers – as always very interesting discussions. Few of us take the time to think about the future and how we can challenge existing practices. However, this article is not about the future but rather endorses a tip intended to assist today with your testing engagements. I call it random automated testing or RAT for short.

Just like real life, rats have a way of uncovering unforeseen issues, forcing us to take action. For example, a small hole in the exterior wall of a house, usually around a venting pipe is an attractive entrance for rats as winter approaches. A colleague had to clean his entire basement to address his rat issue. As a result of this adventure, he also noticed several leaks within his foundation walls requiring repair. These random events are important to testers as they help us uncover dormant system defects.

Random testing is intuitive for testers; however when combining our exploratory testing skills with test automation, we seem to leave this work to our manual efforts. Test automation is not the easiest thing to do and in my early days, I forced myself to automate a GUI application - my challenge for the year. I noticed that the method I deployed to execute my test cases did not allow for random events, i.e. the test scripts executed the test cases using the same path each time on which they were run. This to me was not right. Manually testing includes the execution of different paths within your testing scenarios, i.e. exploratory testing. If manual testers could execute different paths then our automated test scripts should as well, right?

I proceeded to engage my QA peers and ask the question – “how do you create automated testing scripts to randomly test all the paths within an application?” An innocent question which produces great conversation but offered few leads to my problem or possible solution. I quickly concluded that “QA” was struggling with test automation and random testing was the least of everyone's concerns.

My thoughts then turned Sir Isaac Newton. What would he do to resolve the issue? In physics, we are instructed to measure and repeat each sequence very carefully - dead end for obvious reasons. But Sir Isaac Newton had a random event as the story goes to influence his thinking. A direct impact to the noodle - an apple falling on his head and it occurred to me that scheduling random events was the answer. Let me explain.

An effective way of organizing your automated testing scripts is to separate your test data, expected results, test programs, actual testing results and testing steps a.k.a data driven automation testing model. The data driven automation model will execute the test scenario by first reading the testing steps, retrieving the necessary data to execute and then compares actual results with expected results. Hope you are still with me?

The key to random automated testing is the layout of your testing steps into clear paths. The simplest method uses an Excel spread sheet. Each cell within the spread sheet represents an action or value required for testing. Figure one represents a simple GUI application – name and address application.

Figure One

Salutation Dropped combo Box	First Name Box	Last Name Box	Address Box	Submit or return to top
Req'd Field	Req'd Field	Req'd Field	Optional	Req'd Field
Capt.	James	Kirk	Space One	Submit

Figure one does not appear to be anything special. Actually, it looks very basic and in reality, most testers create spreadsheets in a similar fashion to organize their testing. However, the key is the interpretation of the table.

By creating a random generator, each cell may or may not be selected. For example, the steps and data selected could be salutation field - "Capt", last name field - "Kirk" and submit field - "Submit". Figure two is an example if numerous test cases were randomly generated.

Figure Two

Salutation Dropped combo Box	First Name Box	Last Name Box	Address Box	Submit or return to top
Req'd Field	Req'd Field	Req'd Field	Optional	Req'd Field
Capt.		Kirk		Submit

	James			Return to top
		Kirk		Submit
Capt.				Submit

Once the steps — test scenarios and data — are created, they can be driven into the application. The actual results of the execution can be compared to the expected results which are created at the same time the test scenarios are created. In this example, the data can only be submitted into the database if all the mandatory fields have values and the submit option is selected. We expect that in the random test cases generated within figure two, data will not be submitted into the database. (Fig 3)

Figure Three

Salutation Dropped combo Box	First Name Box	Last Name Box	Address Box	Submit or return to top	
Req'd Field	Req'd Field	Req'd Field	Optional	Req'd Field	Expected Results
Capt.		Kirk		Submit	Unable to submit missing data
	James			Return to top	Return to the top
		Kirk		Submit	Unable to submit missing data
Capt.				Submit	Unable to submit missing data

During the actual execution of the test cases, it was found that the last scenario was able to submit "Capt" without any other data for the required mandatory fields, i.e. system defect. Imagine creating random automated testing cases as above (test data and expected results) for a complex application! Let the RAT run free overnight and see what it has found while you were sleeping!

Things to remember

1. Mandatory field ... need this information for test case validation
2. Drop down or selection boxes ... random selection required here
3. General data ... create negative testing data (users do make mistakes)
4. Random generator ... keeps it simple

5. Store your actual results ... see point (6)
6. Expected results ... generally, use the Mandatory field as primary and Drop Down etc. as secondary. This way expected results can be created as the scripts are running
7. Looping ... let it run
8. Performance testing, stability testing etc. ... don't limit yourself - look for other ways to utilize your scripts
9. Figure one: Excel spread sheet is a simplified example used for presenting the concept — random automated testing. Testing tools generally provide their own databases where you can define field names, field values, etc.

From Richard Bornet

Richard Bornet has been in the software business for over 20 years. The last ten he has spent running various testing departments and creating innovative approaches to improve testing. He specializes in test automation and is the inventor of Scenario Tester and co-inventor of Ambiguity Checker software. He can be reached at rbornet@eol.ca or by phone at 416-986-7175

ERE: The Expected Results Engine

Test automation is still in its infancy. I have seen many automation efforts go for naught, or provide minimal benefit. And yet it is my belief that test automation, done properly, can not only replace most manual testing, but can become the de-facto standard of how all testing is done.

The secret, of course, is in the phrase “done properly”.

This article is the story of a problem and a solution. It recounts an experience we had and tells how the model we employed actually changed the whole approach to testing, and made *automation*, not manual testing, the order of the day.

The problem

We had an application which calculated the pricing that should be charged for different products. The values were based on complex calculations; they factored in costs, expenses, market conditions, marketing strategies, store location, dates and other profiles that were kept on the system. Testing the calculations manually was possible, though a major undertaking.

The problem was that every time a new scenario or nuance was introduced, the calculations changed. The effort to re-test the calculations manually was so great, it was avoided in most cases. So what to do?

The solution – first cut

The first approach was to create a spreadsheet. The tester would input all the data (the expenses, costs, conditions, profiles and so on) and the spreadsheet would calculate the answers. In other words, the spreadsheet was simply a tool for easier manual testing.

Soon it became very apparent that it took much too long to enter all the information into the spreadsheet. Just getting all the profiles was a long and laborious effort and involved many SQL queries.

The solution – second cut

We then added some VBA scripts to our spreadsheet to access all the profiles automatically. This cut down on the amount of data that the user had to type in, but it was still too much. The VBA script accessed data in the database, but the users still had to enter into the spreadsheet the data they would have had to type into the application screens.

The solution – third cut

We expanded the VBA scripts further so that we only needed to enter the product number, testing environment and screen, and the script would pull all the necessary information and do the calculations automatically. This information was generally pulled from the databases and sometimes had to be pulled from screens.

This was a significant improvement, but the tester still needed to compare the values from the spreadsheet to the values on the screen.

The solution – fourth cut

Next, we attached the spreadsheet and the script to our test automation. We had structured our automation around a utility (Scenario Tester) that allowed the users to input values and specify navigational flow, then the automation scripts would pick up this information and input the scenario. For the users, it was a bit like “fill in the blanks” testing. Here is the scenario with its navigational flow, here are the values I want to execute, I press a button and it does it.

Once we attached the spreadsheet to the test automation, we could test the expected results. The test scenario would run, and when the appropriate screen came up, the spreadsheet would be called to work out the correct expected values, and the test automation would test that the correct values were in fact appearing on the screen.

The tester was then presented with a comparison of actual vs. expected results, with any differences if any, highlighted.

The implications

Now this may seem like a nice common sense idea though not particularly simple to execute. We have found in using it, however, that the implications are significant.

What we have is a calculation engine, or to coin a new phrase, an “expected result engine” (ERE). Some people have called these engines ‘test oracles’, but for this article we will use the term ERE.

The first and most important aspect of having an ERE, is that you can stop worrying about the expected result. You plug in your inputs, and the engine tells you whether you got the right answer.

Before I talk about the advantages, let me deal with the objections. Whenever we mention this concept we are always answered with a “But...”

The objections – 1. Too time consuming

Having talked to enough people, I already hear the number one objection, “It is so time consuming and difficult to create an ERE!” Yes it is time consuming, but no more so than doing the testing manually. You still have to do the calculations; you still have to figure out the logic of what should take place. However, if you do it manually, you have to do it over and over. So overall, given the time it takes, especially for complicated functionality, it may be much more efficient to build the ERE and then reuse it, than to constantly perform the tests manually.

The objections – 2. Too difficult to handle change

“Well what happens if they change the formulas?”, I hear you cry. Well, what do you do if you’re testing manually? You redo all the tests manually, of course! If you have a hundred manual scenarios... I’m sure you get the picture. We just had to change the ERE and re-run the scenarios using automation.

The objections – 3. Who guarantees the ERE is correct?

“Well who tests the ERE? How do we know the ERE is right?” The same question applies to manual testing: how do we know that the manual result the tester got is right? In practical reality, if the tester has a different result than the screen, the tester investigates to find out whether the screen is wrong or the tester’s calculations are wrong. The same applies to the ERE. If there is an error in the ERE, then it will not match the results in the application, and investigation will show that the ERE needs adjusting.

The objections – 4. Aren’t you writing the application twice?

“So what you’re really doing is duplicating the application functionality in your spreadsheet – aren’t you just writing the application twice?” Not quite, but let’s be honest – it will be a developer writing the ERE, not a tester. Creating a spreadsheet or a piece of code which does the calculations is not the same amount of work as creating a whole application with screens and data. **You don’t have to duplicate every test.** This is an important point: an ERE is a good use of resources to test complex calculations and logic, it is not necessarily the right thing to use for testing boundary conditions. In many cases, you can create a baseline and use that, rather than using an ERE. An example would be testing error messages – an ERE could be overkill, saving the expected results as a baseline and testing against that could be much simpler.

It is true that EREs are time consuming to build. We know – we build them. But when you are one day from implementation and you realize that you need to test one more scenario, it is a great relief to be able to enter some inputs and have the automation tell you if you got the right answer.

So now let’s take a look at advantages of an ERE.

More concentration on Scenarios.

A “scenario” is basically “what the tester is going to do”. For example, I want to create a new customer of a particular type. Different types of customers provide different inputs for formulas. For example, it is important to test conditions such as: “under 18”, “over 65”, “home owner” or “renter”, different salary ranges, etc.

Having an ERE allows you to concentrate on your inputs, rather than the expected results. You can generate many business scenarios because you know the automation will tell you if you have the right answer. More importantly, you may have a very complex set of calculations, and you know you should test every path to the calculation. This is unmanageable as a manual task because of the number of permutations and combinations, each one requiring a new manual calculation. But the automation can be set up to input all the combinations, and the ERE tells you if they pass or fail.

Techniques like Requirement Based Testing can now concentrate on the inputs and not worry about the expected results. This saves everyone a lot of time and aggravation.

You even have the ability to do “what ifs”. A tester can create a customer with some set of characteristics without having to know what the expected results will be, and it doesn’t matter,

as the ERE will tell the tester whether the correct expected result was obtained.

No baselines

The baseline is the “right answer”. We save it and then we use that as our expected result. This only works if given the same inputs, we always get the same “right answer”.

Many systems now use models where the data is not fixed. The results may vary by what date or what day of the week it is. There may be inputs from external sources which are factored in; for example currency exchanges, stock prices, and so on. These may come from other systems testers have no control over, and the actual values may change frequently.

So in this case, we don’t get the same results every time we run the calculation. In other words, there is no simple way of setting up a baseline.

We have worked in organizations with this kind of situation, and I know the amount of work testers have to go through to solve these problems. A common solution is to build whole isolated environments with databases that have to be re-set at the beginning of each test run so they have the same starting points. This can be a huge effort involving large amounts of resources and planning. I have even seen a testing department still running tests with a starting date in March of 2000, because of the need for fixed data. Some testing departments have to spend a great deal of effort to coordinate their test data with other departments or institutions. Others write special routines to populate data which they need for testing. Still others run two sets of tests, one with the old code, and the other with the new code. The old is used as a baseline for the new, and there are constraints on when the tests can be run, for example the new and the old have to run the same day.

If the testers need a fixed baseline every time a new testing cycle starts, then all the data and the databases have to be re-set.

All of that goes away with an ERE. It doesn’t matter whether it is Monday, or Saturday, beginning or middle of the month; or the dollar has gone up or down. These permutations are factored in, and the expected results are still tested.

New paradigm for testing

The basic ERE model is, “Here are my inputs, now tell me whether I got the right answer”. This creates a new paradigm for testing.

Test automation becomes the primary way to test an application

Most testing is still manual. Testers enter inputs into an application and then check the expected results manually. Both the entry and the testing can now be automated.

The simplest model of this is a little utility into which the tester enters some values, presses a button, then the automation enters the values and then tests the expected results. But this can be made much more sophisticated.

For new functionality testing, testers need somewhere to store their scenarios with all the navigation and inputs. From long experience, spreadsheets do not suffice for this. But if they have the right software, it can be quicker to enter the data into the input software and run the tests, than to do the tests manually. In addition, the testers have the scenarios saved, allowing them to easily re-run the tests.

Time is saved on the input side, because automation can enter data much faster than a human being. But the real time is saved on the testing side, where testers do not have to calculate the expected results. Tools calculating results are much faster and much more likely to get the right answer than anyone with a calculator.

But what automation and EREs really do is increase the ability of testers to throw more tests at an application, especially if the logic of application can be accessed in some way other than a GUI. Here testers can create, often with the help of tools, large numbers of permutations and combinations of data, resulting in many varied scenarios. The ERE will validate the results and pick up unexpected or incorrect outcomes. This dramatically increases test coverage.

Moving towards systems that monitor

And why not go one step further and use an ERE to monitor production? As users enter data, they can be monitored, and the ERE can test whether the right results are being obtained. Incorrect results can be flagged. These could be due to defects, an incomplete ERE, or a user entering a scenario no one ever thought possible, so it was not tested and accounted for.

Production-monitoring EREs not only move testing into a new realm, but move it closer to hardware testing. In the hardware world, engineers have access to algorithms that allow for more complete testing. They also build self-tests into hardware so as to test on the fly. This is no longer a distant possibility in the realm of software testing – it is now a real option.

A possible objection is that this could adversely affect performance. But this is a technical problem that can be solved.

Conclusion

Test Automation and EREs save time on the input side, because automation can enter data much faster than a human being. But the real time is saved on the testing side, where testers do not have to concern themselves with calculating the expected results. The ERE can calculate the results much more quickly and is much more likely to get the right answer than anyone with a calculator or pen and paper.

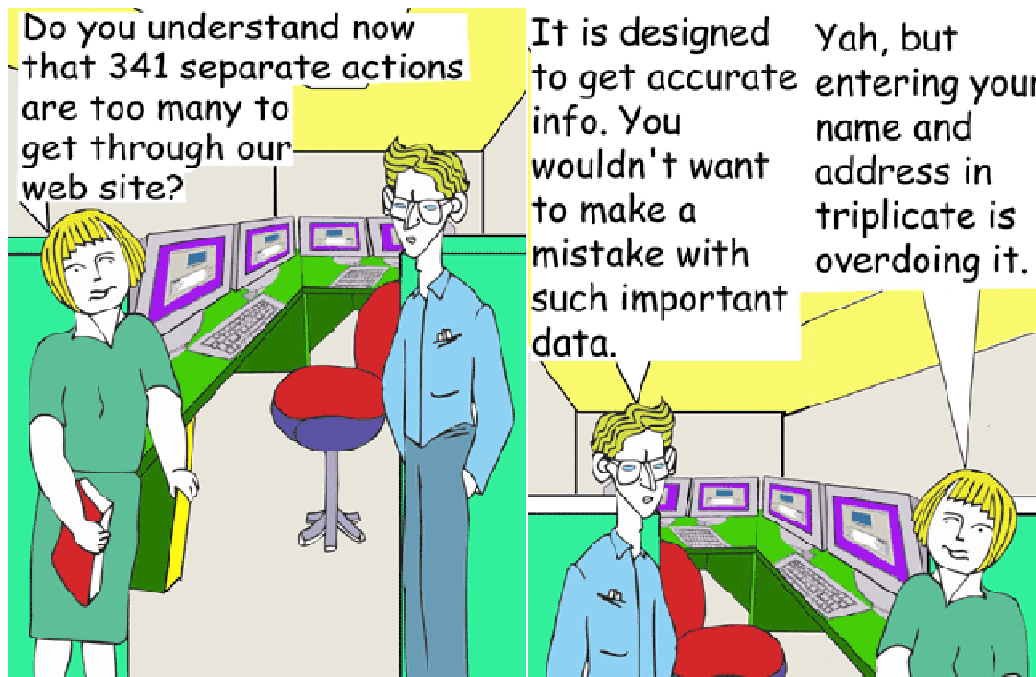
EREs eliminate much of the time and effort of maintaining systems that exist only to store baselines.

EREs and Automation allow testers to dramatically increase the number of tests that can be executed, thereby increasing test coverage.

Finally, EREs, as they become more sophisticated, may have a role not just in testing but also in monitoring production software.

Next Issue – Call for Articles

The main topic of the next magazine will be “The User.” We hear a lot about the user in software development, but how important is the user? Many projects act as if the user does not exist. How important is User Acceptance Testing? We are interested in your thoughts. If you have any **practical** and/or **innovative** suggestions or experiences on how to effectively involve the user, we would be very interested in hearing them.



Quality Cost Analysis: Benefits and Risks

Dr. Cem Kaner

"To our readers.

In the April edition of the magazine we published an article that had been submitted to us entitled "What is Quality Cost Analysis?" Text appeared in this article which was strikingly similar to an article published in 1996, by Dr. Cem Kaner. We were uncomfortable with this similarity and have chosen to take the following steps :

We have removed the article from the April edition and re-posted the magazine on the web site.

We have apologized to Dr. Cem Kaner for any inconvenience or embarrassment that may have been caused.

We also apologize to you the readers.

We have been honoured by Dr. Kaner who graciously allowed us to re-publish his original and seminal article on the subject. For this we are very grateful.

Again our apologies to everyone, and we hope that you truly enjoy his article."

"Because the main language of [corporate management] was money, there emerged the concept of studying quality-related costs as a means of communication between the quality staff departments and the company managers."¹

Joseph Juran, one of the world's leading quality theorists, has been advocating the analysis of quality-related costs since 1951, when he published the first edition of his *Quality Control Handbook*. Feigenbaum made it one of the core ideas underlying the Total Quality Management movement.² It is a tremendously powerful tool for product quality, including software quality.

What is Quality Cost Analysis?

Quality costs are the costs associated with preventing, finding, and correcting defective work. These costs are huge, running at 20% - 40% of sales.³ Many of these costs can be significantly reduced or completely avoided. One of

the key functions of a Quality Engineer is the reduction of the total cost of quality associated with a product.

Here are six useful definitions, as applied to software products. Figure 1 gives examples of the types of cost. Most of Figure 1's examples are (hopefully) self-explanatory, but I'll provide some additional notes on a few of the costs:⁴

- **Prevention Costs:** Costs of activities that are specifically designed to prevent poor quality. Examples of "poor quality" include coding errors, design errors, mistakes in the user manuals, as well as badly documented or unmaintainably complex code.
- Note that most of the prevention costs don't fit within the Testing Group's budget. This money is spent by the programming, design, and marketing staffs.
- **Appraisal Costs:** Costs of activities designed to find quality problems, such as code inspections and any type of testing.
- Design reviews are part prevention and part appraisal. To the degree that you're looking for errors in the proposed design itself when you do the review, you're doing an appraisal. To the degree that you are looking for ways to strengthen the design, you are doing prevention.
- **Failure Costs:** Costs that result from poor quality, such as the cost of fixing bugs and the cost of dealing with customer complaints.
- **Internal Failure Costs:** Failure costs that arise before your company supplies its product to the customer. Along with costs of finding and fixing bugs are many internal failure costs borne by groups outside of Product Development. If a bug blocks someone in your company from doing her job, the costs of the wasted time, the missed milestones, and the overtime to get back onto schedule are all internal failure costs.
- For example, if your company sells thousands of copies of the same program, you will probably print several thousand copies of a multi-color box that contains and describes the program. You (your company) will often be able to get a *much* better deal by booking press time in advance. However, if

you don't get the artwork to the printer on time, you might have to pay for some or all of that wasted press time anyway, and then you may have to pay additional printing fees and rush charges to get the printing done on the new schedule. This can be an added expense of many thousands of dollars.

Some programming groups treat user interface errors as low priority, leaving them until the end to fix. This can be a mistake. Marketing staff need pictures of the product's screen long before the program is finished, in order to get the artwork for the box into the printer on time. User interface bugs - the ones that will be fixed later - can make it hard for these staff members to take (or mock up) accurate screen shots. Delays caused by these minor design flaws, or by bugs that block a packaging staff member from creating or printing special reports, can cause the company to miss its printer deadline.

Including costs like lost opportunity and cost of delays in numerical estimates of the total cost of quality can be controversial. Campanella (1990)⁵ doesn't include these in a detailed listing of examples. Gryna (1988)⁶ recommends against including costs like these in the published totals because fallout from the controversy over them can kill the entire quality cost accounting effort. I include them here because I sometimes find them very useful, even if it might not make sense to include them in a balance sheet.

- **External Failure Costs:** Failure costs that arise after your company supplies the product to the customer, such as customer service costs, or the cost of patching a released product and distributing the patch.
- External failure costs are huge. It is much cheaper to fix problems before shipping the defective product to customers.

Some of these costs must be treated with care. For example, the cost of public relations efforts to soften the publicity effects of bugs is probably not a huge percentage of your company's PR budget. You can't charge the entire PR budget as a quality-related cost. But any money that the PR group has to spend to specifically cope with potentially bad publicity due to bugs is a failure cost.

I've omitted from Figure 1 several additional costs that I don't know how to estimate, and that I suspect are too often too controversial to use. Of these, my two strongest themes are cost of high turnover (people quit over quality-related frustration - this definitely includes sales staff, not just development and support) and cost of lost pride (many people will work less hard, with less care, if they believe that the final product will be low quality no matter what they do.)

- **Total Cost of Quality:** The sum of costs: Prevention + Appraisal + Internal Failure + External Failure.

Figure 1. Examples of Quality Costs Associated with Software Products.

<i>Prevention</i>	<i>Appraisal</i>
<ul style="list-style-type: none"> • Staff training • Requirements analysis • Early prototyping • Fault-tolerant design • Defensive programming • Usability analysis • Clear specification • Accurate internal documentation • Evaluation of the reliability of development tools (before buying them) or of other potential components of the product 	<ul style="list-style-type: none"> • Design review • Code inspection • Glass box testing • Black box testing • Training testers • Beta testing • Test automation • Usability testing • Pre-release out-of-box testing by customer service staff

<i>Internal Failure</i>	<i>External Failure</i>
<ul style="list-style-type: none"> • Bug fixes • Regression testing • Wasted in-house user time • Wasted tester time • Wasted writer time • Wasted marketer time • Wasted advertisements⁷ • Direct cost of late shipment⁸ • Opportunity cost of late shipment 	<ul style="list-style-type: none"> • Technical support calls⁹ • Preparation of support answer books • Investigation of customer complaints • Refunds and recalls • Coding / testing of interim bug fix releases • Shipping of updated product • Added expense of supporting multiple versions of the product in the field • PR work to soften drafts of harsh reviews • Lost sales • Lost customer goodwill • Discounts to resellers to encourage them to keep selling the product • Warranty costs • Liability costs • Government investigations¹⁰ • Penalties¹¹ • All other costs imposed by law

What Makes this Approach Powerful?

Over the long term, a project (or corporate) cost accounting system that tracks quality-related costs can be a fundamentally important management tool. This is the path that Juran and Feigenbaum will lead you down, and they and their followers have frequently and eloquently explained the path, the system, and the goal.

I generally work with young, consumer-oriented software companies who don't see TQM programs as their top priority, and therefore my approach is more tactical. There is significant benefit in using the language and insights of quality cost analysis, on a project/product by project/product basis, even in a company that has no interest in Total Quality Management or other formal quality management models.¹²

Here's an example. Suppose that some feature has been designed in a way that you believe will be awkward and annoying for the customer. You raise the issue and the project manager rejects your report as subjective. It's "not a bug." Where do you go if you don't want to drop this issue? One approach is to keep taking it to higher-level managers within product development (or within the company as a whole). But without additional arguments, you'll often keep losing, without making any friends in the process.

Suppose that you change your emphasis instead. Rather than saying that, in your opinion, customers won't be happy, collect some other data:¹³

- **Ask the writers:** Is this design odd enough that it is causing extra effort to document? Would a simpler design reduce writing time and the number of pages in the manual?
- **Ask the training staff:** Are they going to have to spend extra time in class, and to write more supplementary materials because of this design?
- **Ask Technical Support and Customer Service:** Will this design increase support costs? Will it take longer to train support staff? Will there be more calls for explanations or help? More complaints? Have customers asked for refunds in previous versions of the product because of features designed like this one?
- **Check for related problems:** Is this design having other effects on the reliability of the program? Has it caused other bugs? (Look in the database.) Made the code harder to change? (Ask the programmers.)
- **Ask the sales staff:** If you think that this feature is very visible, and visibly wrong, ask whether it will interfere with sales demonstrations, or add to customer resistance.
- **What about magazine reviews?** Is this problem likely to be visible enough to be complained about by reviewers? If you think so, check your impression with someone in Marketing or PR.

You won't get cost estimates from everyone, but you might be able to get ballpark estimates from most, along with one or two carefully considered estimates. This is enough to give you a range to present at the next project meeting, or in

a follow-up to your original bug report. Notice the difference in your posture:

- You're no longer presenting *your opinion* that the feature is a problem. You're presenting information collected from several parts of the company that demonstrates that this feature's design is a problem.
- You're no longer arguing that the feature should be changed just to improve the quality. No one else in the room can posture and say that you're being "idealistic" whereas a more pragmatic, real-world businessperson wouldn't worry about problems like this one. Instead, *you're* the one making the hard-nosed business argument, "This design is going to cost us \$X in failure costs. How much will it cost to fix it?"
- Your estimates are based on information from other stakeholders in this project. If you've fairly represented their views, you'll get support from them, at least to the extent of them saying that you are honestly representing the data you've collected.

Along with arguing about individual bugs, or groups of bugs, this approach opens up opportunities for you (and other non-testers who come to realize the power of your approach) to make business cases on several other types of issues. For example:

- The question of who should do unit testing (the programmers, the testers, or no one) can be phrased and studied as a cost-of-quality issue. The programmers might be more efficient than testers who don't know the code, but the testers might be less expensive per hour than the programmers, and easier to recruit and train, and safer (unlike newly added programmers, new testers can't write new bugs into the code) to add late in the project.
- The depth of the user manual's index is a cost-of-quality issue. An excellent index might cost 35 indexer-minutes per page of the manual (so a 200 page book would take over three person-weeks to index). Trade this cost against the reduction in support calls because people can *find* answers to their questions in the manual.
- The best investment to achieve better quality might be additional training and staffing of the programming group (prevent the bugs rather than find and fix them).

- You (in combination with the Documentation, Marketing, or Customer Service group) might demonstrate that the user interface must be fixed and frozen sooner because of the impact of late changes on the costs of developing documentation, packaging, marketing collaterals, training materials, and support materials.

Implementation Risks

Gryna (1988)¹⁴ and Juran & Gryna (1980)¹⁵ point out several problems that have caused cost-of-quality approaches to fail. I'll mention two of the main ones here.

First, it's unwise to try to achieve too much, too fast. For example, don't try to apply a quality cost system to every project until you've applied it successfully to one project. And don't try to measure all of the costs, because you probably can't.¹⁶

Second, beware of insisting on controversial costs. Gryna (1988)¹⁷ points out several types of costs that other managers might challenge as not being quality-related. If you include these costs in your totals (such as total cost of quality), some readers will believe that you are padding these totals, to achieve a more dramatic effect. Gryna's advice is to not include them. This is usually wise advice, but it can lead you to underestimate your customer's probable dissatisfaction with your product. As we see in the next section, down that road lies LawyerLand.

The Dark Side of Quality Cost Analysis

Quality Cost Analysis looks at the company's costs, not the customer's costs. The manufacturer and seller are definitely not the only people who suffer quality-related costs. The customer suffers quality-related costs too. If a manufacturer sells a bad product, the customer faces significant expenses in dealing with that bad product.

The Ford Pinto litigation provided the most famous example of a quality cost analysis that evaluated company costs without considering customers' costs from the customers' viewpoint. Among the documents produced in these cases was the Grush-Saunby report, which looked at costs associated with fuel tank integrity. The key calculations appeared in Table 3 of the report:¹⁸

Benefits and Costs Relating to Fuel Leakage
Associated with the Static Rollover Test Portion of FMVSS 208

Benefits

Savings - 180 burn deaths, 180 serious burn injuries, 2100 burned vehicles
Unit Cost -- \$200,000 per death, \$67,000 per injury, \$700 per vehicle
Total Benefit - 180 x (\$200,000) + 180 x (\$67,000) + 2100 x (\$700) = \$49.5 million.

Costs

Sales - 11 million cars, 1.5 million light trucks.
Unit Cost -- \$11 per car, \$11 per truck
Total Cost - 11,000,000 x (\$11) + 1,500,000 x (\$11) = \$137 million.

In other words, it looked cheaper to pay an average of \$200,000 per death in lawsuit costs than to pay \$11 per car to prevent fuel tank explosions. Ultimately, the lawsuit losses were much higher.¹⁹

This kind of analysis didn't go away with the Pinto. For example, in the more recent case of *General Motors Corp. v. Johnston* (1992)²⁰, a PROM controlled the fuel injector in a pickup truck. The truck stalled because of a defect in the PROM and in the ensuing accident, Johnston's seven-year old grandchild was killed. The Alabama Supreme Court justified an award of \$7.5 million in punitive damages against GM by noting that GM "saved approximately

\$42,000,000 by not having a recall or otherwise notifying its purchasers of the problem related to the PROM."

Most software failures don't lead to deaths. Most software projects involve conscious tradeoffs among several factors, including cost, time to completion, richness of the feature set, and reliability. There is nothing wrong with doing this type of business tradeoff, consciously and explicitly, unless you fail to take into account the fact that some of the problems that you leave in the product might cost your customers much, much more than they cost your company. Figure 2 lists some of the external failure costs that are borne by customers, rather than by the company.

Figure 2. Comparison of External Failure Costs Borne by the Buyer and the Seller

<i>Seller: external failure costs</i>	<i>Customer: failure costs</i>
These are the types of costs absorbed by the seller that releases a defective product.	These are the types of costs absorbed by the customer who buys a defective product.
<ul style="list-style-type: none"> • Technical support calls • Preparation of support answer books • Investigation of customer complaints • Refunds and recalls • Coding / testing of interim bug fix releases • Shipping of updated product • Added expense of supporting multiple versions of the product in the field • PR work to soften drafts of harsh reviews • Lost sales • Lost customer goodwill • Discounts to resellers to encourage them to keep selling the product • Warranty costs • Liability costs • Government investigations • Penalties • All other costs imposed by law 	<ul style="list-style-type: none"> • Wasted time • Lost data • Lost business • Embarrassment • Frustrated employees quit • Demos or presentations to potential customers fail because of the software • Failure when attempting other tasks that can only be done once • Cost of replacing product • Cost of reconfiguring the system • Cost of recovery software • Cost of tech support • Injury / death

The point of quality-related litigation is to transfer some of the costs borne by a cheated or injured customer back to the maker or seller of the defective product. The well-publicized cases are for disastrous personal injuries, but there are plenty of cases against computer companies and software companies for breach of contract, breach of warranty, fraud, etc.

The problem of cost-of-quality analysis is that it sets us up to underestimate our litigation and customer dissatisfaction risks. We think, when we have estimated the total cost of quality associated with a project, that we have done a fairly complete analysis. But if we don't take customers' external failure costs into account at some point, we can be surprised by huge increased costs (lawsuits) over decisions that we thought, in our incomplete analyses, were safe and reasonable.

¹ Gryna, F. M. (1988) "Quality Costs" in Juran, J.M. & Gryna, F. M. (1988, 4th Ed.), *Juran's Quality Control Handbook*, McGraw-Hill, page 4.2.

² Feigenbaum, A.V. (1991, 3rd Ed. Revised), *Total Quality Control*, McGraw-Hill, Chapter 7.

³ Gryna, F. M. "Quality Costs" in Juran, J.M. & Gryna, F. M. (1988, 4th Ed.), *Juran's Quality Control Handbook*, McGraw-Hill, page 4.2. I'm not aware of reliable data on quality costs in software.

⁴ These are my translations of the ideas for a software development audience. More general, and more complete, definitions are available in Campanella, J. (Ed.) (1990), *Principles of Quality Costs*, ASQC Quality Press, as well as in Juran's and Feigenbaum's works.

⁵ *Principles of Quality Costs*, ASQC Quality Press, Appendix B, "Detailed Description of Quality Cost Elements."

⁶ "Quality Costs" in Juran, J.M. & Gryna, F. M. (1988, 4th Ed.), *Juran's Quality Control Handbook*, McGraw-Hill, pages 4.9 - 4.12.

⁷ The product is scheduled for release on July 1, so your company arranges (far in advance) for an advertising campaign starting July 10. The product has too many bugs to ship, and is delayed until December. All that advertising money was wasted.

⁸ If the product had to be shipped late because of bugs that had to be fixed, the direct cost of late shipment includes the lost sales, whereas the opportunity cost of the late shipment includes the costs of delaying other projects while everyone finished this one.

⁹ Note, by the way, that you can reduce external failure costs without improving product quality. To reduce post-sale support costs without increasing customer satisfaction, charge people for support. Switch from a toll-free support line to a toll line, cut your support staff size and you can leave callers on hold for a long time at their expense. This discourages them from calling back. Because these cost reductions don't increase customer satisfaction, the seller's cost of quality is going down, but the customer's is not.

¹⁰ This is the cost of cooperating with a government investigation. Even if your company isn't charged or penalized, you spend money on lawyers, etc.

¹¹ Some penalties are written into the contract between the software developer and the purchaser, and the developer pays them if the product is late or has specified problems. Other penalties are imposed by law. For example, the developer/publisher of a computer program that prepares United States taxes is liable for penalties to the Internal Revenue Service for errors in tax returns that are caused by bugs or design errors in the program. The publishers are treated like other tax preparers (accountants, tax lawyers, etc.). See *Revenue Ruling 85-189* in *Cumulative Bulletin*, 1985-2, page 341.

¹² I am most definitely not saying that a tactical approach is more practical than an integrated, long-term approach. Gryna notes that there are two common approaches to cost-of-quality programs. One approach involves one-shot studies that help the company identify targets for significant improvement. The other approach incorporates quality cost control into the structure of the business. (Gryna, 1988, in Juran, J. M. & Gryna, F. M. (1988, 4th Ed.), *Juran's Quality Control Handbook*, McGraw-Hill, pages 4.2 onward.) The one-shot, tactical approach can prove the benefit of the more strategic, long-term system to a skeptical company.

¹³ Be sensitive to how you do this. If you adopt a tone that says that you think the project manager and the programming staff are idiots, you won't enjoy the long-term results.

¹⁴ in Juran, J. M. & Gryna, F. M. (1988, 4th Ed.), *Juran's Quality Control Handbook*, McGraw-Hill, pages 4.27-4.28.

¹⁵ *Quality Planning and Analysis* (2nd Ed.), McGraw-Hill, pages 30-32. Also, see Brown, M. G., Hitchcock, D. E. & Willard, M. L. (1994), *Why TQM Fails and What To Do About It*, Irwin Professional Publishing.

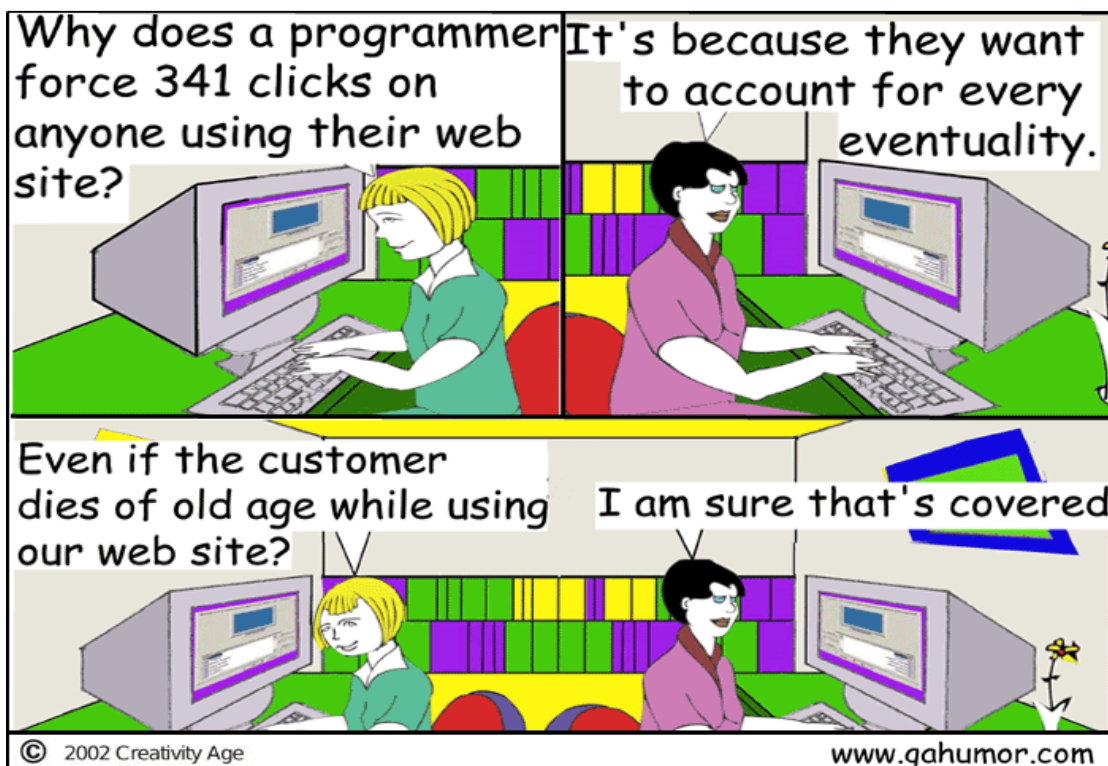
¹⁶ As he says in Juran, J.M. (1992), *Juran on Quality by Design*, The Free Press, p. 119, Costs of poor quality "are huge, but the amounts are not known with precision. In most companies the accounting system provides only a minority of the information needed to quantify this cost of poor quality. It takes a great deal of time and effort to extend the accounting system so as to provide full coverage. Most companies have concluded that such an effort is not cost effective. [¶] What can be done is to fill the gap by *estimates*, which provide managers with approximate information as to the total cost of poor quality and as to where the major areas of concentration are."

¹⁷ "Quality Costs" in Juran, J.M. & Gryna, F. M. (1988, 4th Ed.), *Juran's Quality Control Handbook*, McGraw-Hill, pages 4.9 - 4.12.

¹⁸ This table is published in Keeton, W. P., Owen, D.G., Montgomery, J. E., & Green, M.D. (1989, 2nd Ed.) *Products Liability and Safety, Cases and Materials*, Foundation Press, page 841 and Posner, R.A. (1982) *Tort Law: Cases and Economic Analysis*, Little Brown & Co., page 225.

¹⁹ *Grimshaw v. Ford Motor Co.* (1981), (California Court of Appeal), *California Reporter*, volume 174, page 348.

²⁰ *Southern Reporter*, 2nd Series, volume 592, pages 1054 and 1061.



Events, Conferences, Education and Services

Conference Watch

Here are some up-coming conferences:

QAI International Quality Conference
October 2-6, 2006
Toronto, ON
<http://www.gaicanada.org/conferences>

STARWest
October 16-20, 2006
Anaheim, CA
<http://www.sqe.com/starwest>

QAI Annual Software Testing Conference and Testing Manager's Workshop & Conference
November 13-17, 2006
Orlando, FL
http://qaiworldwide.org/conferences/nov_2006/

STAREast
May 14-18, 2007
Orlando, FL
<http://www.sqe.com/stareast/>



and



present

Designing For Usability ***for Web-based and Windows Applications*** ***A Two-Day Workshop in Toronto*** ***November 28 & 29, 2006***

TASSQ, the Toronto Association of Systems and Software Quality, is pleased to present a two day course, sponsored by TASSQ and taught by James Hobart, an internationally recognized user interface design consultant. Mr. Hobart's insight is sought by leading publications including PC Week, Information Week, InfoWorld and UNIX Review. The Course is intended to explain how to apply the concepts of human centered design to the paradigm of web development.

Attendees will learn how to

- Develop a detailed understanding of your users through task analysis, mental models, and user profiles
- Proper layout and design techniques
- Learn new design modeling techniques
- Create and implement in-house web standards
- Provide feedback and find usability issues
- Plan and conduct an effective usability test
- Design more successful applications
- Validate and defend important design decisions

Who Should Attend

The course is designed for corporate or commercial testers, developers and analysts that are, or plan to be, involved in software or web projects. Anyone concerned with developing well-designed web-based and windows applications, including individuals that will gather user requirements or end-users themselves will also benefit from attending.

Cancellations/Substitutions

If you are registered but cannot attend, TASSQ must be notified via email: admin@tassq.org by November 13, 2006, 2007 in order to receive a full refund (less \$25 admin fee). Failure to notify will result in the forfeit of any obligation by TASSQ. If you cannot attend but wish to send a substitution, please notify TASSQ via email. TASSQ reserves the right to cancel the seminar based on insufficient registrations wherein a full refund will be provided to registrants.

Date: November 28 & 29, 2006
9:00 am to 5:00 pm

Location: St. Michael's College -Sr Common Rm
University of Toronto, 81 St. Mary Street,
Toronto (near Museum Subway Stop)

Fees: \$900 for TASSQ members
\$1100 for non-members or \$900 -group of 3+

Registration Deadline: November 3, 2006

Included in the Fees:

- Workshop instruction & complete set of templates to implement the process within one's organization
- Registrants will be responsible for their own lunch/not included in seminar fee.
- Discounted overnight accommodation (\$119.99/\$129.99 single/double occupancy) available at Holiday Inn Midtown at www.holidayinn.com/torontomidtown. Book online, Corporate, Group, IATA, Corporate ID 100217931

DESIGNING FOR USABILITY – 2 Day Course

COURSE SUMMARY

Learn how to define user goals and business needs while applying proven design techniques to ensure highly usable and successful applications.

Learn from the experts who have been delivering success in this field for over a decade! We will show you how to adopt a user-centric perspective, apply a proven process for identifying true user requirements, developing and validating conceptual models, and creating designs that are highly usable.

This class is designed for corporate or commercial testers, developers and analysts that are, or plan to be, involved in software or web projects. Anyone concerned with developing well-designed web applications, including individuals that will gather user requirements or end-users themselves will also benefit from attending.

WHO SHOULD ATTEND

- ▶ **Testers** who are responsible for testing web-based and Windows applications.
- ▶ **Project Managers** who are responsible for establishing or managing application development strategies.
- ▶ **Project Leaders** who need to know proven steps for creating useable software.
- ▶ **GUI Designers** who need to know how and when to use controls when creating user interfaces.
- ▶ **Software Developers** who are looking to expand their knowledge web and windows application design.
- ▶ **Webmasters** who are responsible for managing and implementing web technology.
- ▶ **Analysts** who are responsible for documenting requirements.
- ▶ **End Users** who need to understand the principles of good user interface design techniques.

WHAT YOU WILL LEARN

This two-day class for developers, end-users, interaction designers, and managers explains how to apply the concepts of human centered design to the paradigm of web and application development.

Attendees will learn how to:

- ▶ Develop a detailed understanding of your users through task analysis, mental models, and user profiles
- ▶ Proper layout and design techniques
- ▶ Learn new design modeling techniques
- ▶ Create and implement in-house web standards
- ▶ Provide feedback and find usability issues
- ▶ Plan and conduct an effective usability test
- ▶ Design more successful applications
- ▶ Validate and defend important design decisions

BENEFITS TO YOUR COMPANY

- ▶ Implement successful web and windows applications
- ▶ Reduce deployment costs by web-enabling your existing applications
- ▶ Increase productivity with highly usable applications
- ▶ Avoid costly design mistakes
- ▶ Implement a repeatable successful design approach
- ▶ Be more thorough when testing applications.

Attendees will walk away with a **complete set of templates** for quickly implementing this process within their organization.

Classic Systems Solutions, Inc.
The Usability Engineering Experts™

101 B Sandcreek Rd., Suite 209 • Brentwood, CA 94513

☎ (925) 308-7686 • 📠 (925) 516-9658

🌐 www.classicsys.com • 🌐 www.quivguide.com

Who is James Hobart?

James Hobart is an internationally recognized User Interface design consultant and president of Classic System Solutions, Inc. He specializes in the design and development of large-scale, high-volume user interface design. He is an expert in GUI design for transaction processing systems and web and portal application strategies.

Mr. Hobart has over twenty years of software development experience and over sixteen years of user interface design experience. He has successfully managed and participated in numerous software projects, as well as advised a number of Fortune 1000 firms on their technology direction. He has helped a large number of firms develop in-house GUI standards and software user interface designs. He has assisted in the design and implementation of systems for the Finance, Transportation, Retail, Insurance, Public Utilities, Government, Distribution, Medical, Manufacturing and Software development tool sectors.

Additionally, he has successfully utilized Agile techniques with clients and has assisted with their adoption of user centered design techniques. He has worked with a wide array of development tools using distributed object and web-based technology.

Mr. Hobart served on the advisory board for the International Windows Summit, and has served as GUI Track Chairman for Window World/Comdex and Software Development conferences. He also speaks on the public seminar track for Digital Consulting (DCI). He speaks regularly about user interface design topics and topics at national conferences. His industry insights are frequently sought by leading publications including PC Week, Information Week, InfoWorld and UNIX Review.

Designing for Usability Course Testimonial

Many years ago, I was designing software GUI interfaces for a living. I felt I had a knack for this, and managed to produce some very exciting and different software. Some of the GUIs had received excellent reviews from many of the major PC reviewers. GUI design was my area of expertise.

Because of this, I was asked by a consulting company to teach a GUI design course for them. I had to go to Chicago to become certified.

Now I have to be honest, I went to Chicago with a slight air of superiority – “I am not sure what they can teach *me*.” Jim Hobart was the instructor. To cut a long story short, I was humbled. Jim was a great instructor, I learned a lot, and the course was a great deal of fun.

Before this course, I believed that good GUI design was an art which necessitated true creativity and deep insights into people. There is still some truth to that, but this course taught a strategy on how to get there. This is a strategy everyone can follow. Whereas before I would make design decisions “because I have a hunch and it feels right,” now I had a language with which I could explain to people why these decisions were taken.

And that’s what this course does. It clearly demonstrates how important a good GUI is to the success of a project. It gives everyone a strategy and a language which helps them create really useful GUIs. It brings all the interested parties onto the same page, and it shows testers how to test GUIs.

Whether you are designing a Client Server Application or a Web-based Application, it is the one course everyone should take: testers, developers, managers, business analysts and users. This course brings these different groups together.

If you only take one course this year, I would recommend this one.

Richard Borne

ISTQB Certified Tester Advanced Test Management

October 30-November 2, 2006

Toronto, ON

The International Software Testing Qualifications Board supports a single, globally-recognized certification for software and system testing professionals. Why settle for anything less? Rex Black Consulting Services, Inc., offers a Foundation Level course and the first Advanced Level courses available in the United States and Canada.

Register today at www.rexblackconsulting.com.

Advanced Test Management covers;

- Definition of Software Testing
- The Generic Test Process
- Test Management
- Test Effort Estimation
- Bug Management
- Testing and Risk Management
- The Test Team
- Organizational Forms of Software Testing
- Testing and Process Improvement

Why get certified?

The International Software Testing Qualifications Board (ISTQB™) was founded in 2002. The ISTQB™ is a non-profit organization comprised of the most widely respected software testers in the world, including Rex Black, President of Rex Black Consulting Services, Inc. The ISTQB supports an internationally accepted title of “ISTQB™ Certified Tester.” The ISTQB provides syllabi and sets guidelines for accreditation and examination for the national boards. Accreditation and certification are regulated by the national boards. Currently, the national boards include American, Australia, Austrian, Bangladesh, Brazil, Chinese, Canadian, Danish, Dutch, Finnish, French, German, Indian, Israeli, Japanese, Korean, Latin American, Norwegian, Polish, Portuguese, Russian, South East European, Spanish, Swedish, UK, and Ukraine.

RBCS’ ISTQB certification courses are taught only by elite instructors who hold both Foundation and Advanced Level certifications. Following this course, attendees may choose to take the ISTQB Advanced Test Management exam. Attendees who pass the exam become ISTQB™ Advanced certified. This certification is internationally recognized by the ISTQB™ and by each of the national boards. Imagine having a professional qualification with that kind of global recognition. Certification allows you to demonstrate extensive knowledge of testing that will contribute to your career growth, distinguish your resume, and prove your professionalism to peers and managers. Don’t wait!

Who should attend?

All Advanced Level exam takers should have successfully completed the Foundation Level exam and have sufficient practical experience. Verification is available for a nominal fee from the American Software Testing Qualifications Board and is required to take an Advanced exam. For more information please visit www.astqb.org

Fees

Advanced Management \$2,500 USD

Fees include tuition, course materials, continental breakfast, lunch and snacks, certificate of completion, and \$200 ISTQB™ exam fee. TASSQ members are eligible for a 10% discount on course tuition. If you bring a friend, your friend will also receive a 10% discount.

Cancellations

Registrants who fail to attend are responsible for 100% of the course tuition. Registrants are entitled to a 100% refund if cancellations are received at least 30 days prior to event.

RBCS reserves the right to cancel an event or make changes to an event schedule. If an event is cancelled or rescheduled, RBCS will refund 100% of the tuition.



Presents

Just in Time (JIT) Testing

A Three-Day Workshop in Toronto

February 21,22,23, 2007

TASSQ, the Toronto Association of Systems and Software Quality, is pleased to present this high demand, three day course, sponsored by TASSQ and taught by Robert Sabourin, a 25-year software development veteran. The JIT Workshop is designed to help testers, developers, test leads, and managers in learning how to complete testing effectively in turbulent development projects.

If you want to be ready for almost anything, whilst knowing what *not* to test within tight project schedules, then this workshop is for you! Robert Sabourin will apply real techniques to real projects, including examples drawn from many sectors including insurance, banking, telecommunications, medical software, multilingual computing, security software and others.

After completing the Workshop, you will know how to get effective testing done in the volatile environment of a Web/e-Commerce software project. Here is a brief outline of the topics covered:

- I. Test Planning and Organization Techniques
- II. Exploratory Testing
- III. Deciding how to Focus Test
- IV. Tracking
- V. Scheduling
- VI. Triage

Who Should Attend

Anyone concerned with software testing including Test Leads, Developer Leads, Managers and Executives.

Cancellations/Substitutions

If you are registered but cannot attend, TASSQ must be notified via email: admin@tassq.org by **January 10, 2007** in order to receive a full refund (less \$25 admin fee). Failure to notify will result in the forfeit of any obligation by TASSQ. If you cannot attend but wish to send a substitution, please notify TASSQ via email. TASSQ reserves the right to cancel the seminar based on insufficient registrations wherein a full refund will be provided to registrants.

Date: February 21-23, 2007
8:30 am to 4:30 pm

Location: Etobicoke or downtown Toronto
Exact Location TBD mid October/06

Fees: \$1000 for TASSQ members
\$1150 for non-members

Registration Deadline: December 22, 2006

Included in the Fees:

- Workshop instruction
- CD & Laminated "Cheat Sheet"
- Registrants will be responsible for their own lunch/not included in seminar fee.
- Discounted overnight accommodation to be made available - details to follow.

Space is limited to the first 25 people to register.

No tape recording will be permitted under any circumstances.

Please complete the form on next page and mail to TASSQ together with payment.

About Robert Sabourin

Robert Sabourin has more than twenty-five years of management experience, leading teams of software development professionals. A well-respected member of the software engineering community, Robert has managed, trained, mentored, and coached hundreds of top professionals in the field. He frequently speaks at conferences and writes on software engineering, SQA, testing, management, and internationalization. The author of *I am a Bug!*, the popular software testing children's book, Robert is an adjunct professor of Software Engineering at McGill University