Association for Software Testing

CAST 2008: Beyond the Boundaries

July 14-16, 2008 – Toronto, Canada

http://www.cast2008.org

Proceedings

Table of Contents

Association for Software Testing

CAST 2008: Beyond the Boundaries

July 14-16, 2008 – Toronto, Canada

http://www.cast2008.org

Program: Day 1 – Monday, July 14, 2008

Time	Room: Terrace 3rd Floor	Room: Saint David 3rd Floor	Room: Saint Patrick North 3rd Floor	Room: Saint Patrick South 3rd Floor	Room: Colony West
7:30 – 9:00 AM	Continental Breakfast (Room: 3rd Floor Foyer)				Registration
9:00 – 10:30 AM	**Jerry Weinberg:** *The Tester's Communication Clinic*	**Scott Barber**: *Performance Testing Software Systems*	**Julian Harty**: *Mobile Wireless Test Automation*	**Hung Nguyen**: *From Craftmanship to Leadership*	
10:30 – 10:45 AM	Break (Room: 3rd Floor Foyer)				
10:45 AM - Noon	**Jerry Weinberg:** *Continued*	**Scott Barber**: *Continued*	**Julian Harty**: *Continued*	**Hung Nguyen**: *Continued*	
Noon - 1:00 PM	Lunch (Room: 3rd Floor Foyer)				
1:00 - 2:45 PM	**Jerry Weinberg:** *Continued*	**Scott Barber**: *Continued*	**Julian Harty**: *Continued*	**Hung Nguyen**: *Continued*	
2:45 - 3:00 PM	Break (Room: 3rd Floor Foyer)				
3:00 – 5:00 PM	**Jerry Weinberg:** *Continued*	**Scott Barber**: *Continued*	**Julian Harty**: *Continued*	**Hung Nguyen**: *Continued*	
5:00 – 7:00 PM	Break				
7:00 – 9:00 PM	Birds of a Feather Proposal: Teaching of Software Testing (Cem Kaner, Becky Fiedler)	AST SIG Meetings (SIG Leads)	Get your questions answered about AST (Michael Kelly)	Open Space	

Confer:
The focus of CAST is on the "confer" part of the word "conference". You will find colored index cards in your packet. These are meant to signal the facilitator when you have questions. When the speaker is speaking, you may ask "Clarifying Only" questions. But when the speaker is done, it's "Open Season" at which point you can raise any of three cards:

- **Green**: this "On Stack" card signals the facilitator that you have a question that relates to the current thread of discussion.
- **Blue:** this "New Stack" card signals that you have a question or comment unrelated to the current thread.
- **Red:** the "Burning Issue" card. This is used when you are urgently compelled to interrupt the speaker. It can be a point-of-order, an argument, a problem with the facility acoustics, or something you need to get off your chest quickly because you've been provoked in a way that's meaningful to you. Once it is used, however, it is taken away (except when you use it to flag facility acoustics problems).

AST annual meeting and elections:

The AST is a non-profit professional association dedicated to improving the practice of software testing by advancing the science of testing and its application. Members and non-members may be interested in hearing how the AST operates and what its plans are for the coming year. The AST is run by members who volunteer as a nominated, elected slate of officers. The AST elections for their Board of Directors will be held at 4:00 pm on Tuesday, July 15. The AST Annual Membership Meeting where election results will be announced and an overview presentation is made to the membership will be held from 4:00 to 5:00 pm on Wednesday, July 16.

Tutorials:
If you signed up for a particular tutorial, you may switch to another, provided there are seats left in the room for the tutorial-in-progress you want to join.

Sponsors:
In return for the support of our sponsors, we have given them space in Colony West, with vendor presentations taking place in Terrace 3rd Floor. See the CD for additional Sponsor materials.

Program: Day 2 – Tuesday, July 15, 2008

Sessions	Room / Location				
Time	**Room: Colony Ballroom 2nd Floor**	**Room: Saint David 3rd Floor**	**Room: Saint Patrick 3rd Floor**	**Room: Terrace 3rd Floor**	**Room: Colony West**
7:30 – 8:45 AM	Continental Breakfast and Networking (Room: 3rd Floor Foyer)				
8:45 – 9:00 AM	Opening Remarks				Registration and Sponsors booths
9:00 -10:30AM	**KEYNOTE - Gerald M. Weinberg:** *Lessons from the Past to Carry into the Future*				
10:30 – 10:45 AM	Break (Room: Colony West)				
	Speaker Introductions:				
10:45 – 11:45 AM	**Martin Taylor**: *Visualization and Statistical Methods*	**Michael Bolton and Jonathan Kohl:** *Testing and Music*	**Diane Kelly and Rebecca Sanders:** *The Challenge of Testing Scientific Software*	*Vendor Presentations*	
11:45 – 1:00 PM	Lunch				
1:00 – 2:30 PM	**KEYNOTE - Robert Sabourin:** *Applied Testing Lessons from Delivery Room Labor Triage*				
2:30 – 2:45 PM	Break (Room: Colony West)				
	Speaker Introductions:				
2:45 – 3:45 PM	**Doug Hoffman:** *Lessons for Testing from Financial Accounting*	**Jerry Kominar:** *Sleight-of-Quality*	**Morven Gentleman:** *Measuring File Systems*	*Vendor Presentations*	
3:45 – 4:00 PM	Break (Room: Colony West)				
4:00 – 5:00 PM	AST Board of Director's Election				
5:00 - 6:00 PM	AST Board Of Director's Meeting				

Association for Software Testing

CAST 2008: Beyond the Boundaries

July 14-16, 2008 – Toronto, Canada

http://www.cast2008.org

Program: Day 3 – Wednesday, July 16, 2008

Sessions	Room / Location				
Time	**Room: Colony Ballroom 2nd Floor**	**Room: Saint David 3rd Floor**	**Room: Saint Patrick 3rd Floor**	**Room: Terrace 3rd Floor**	**Room: Colony West**
7:30 AM – 8:45 AM	Continental Breakfast and Networking (Room: Colony West)				Registration and Sponsors booths
8:45 – 9:00 AM	Opening Remarks				
9:00 – 10:30 AM	**KEYNOTE - Cem Kaner, JD, PhD:** *The Value of Checklists and the Danger of Scripts*				
10:30 – 10:45 AM	Break (Room: Colony West)				
	Speaker Introductions:				
10:45 – 11:45 AM	**Steve Richardson and Adam Geras**: Seeking Data Quality	**Adam White:** *Software Testing To Improv*	*Lightning Talks*	*Vendor Presentations*	
11:45 – 1:00 PM	Lunch (Room: Colony West)				
1:00 – 2:30 PM	**KEYNOTE -Brian Fisher:** *The New Science of Visual Analytics*				
2:30 – 2:45 PM	Break (Room: Colony West)				
	Speaker Introductions:				
2:45 – 3:45 PM	**Bart Broekman:** *Testing Fuzzy Interfaces*	**Adam Goucher:** *Lessons in Team Leadership from Kids in Armor*	**Scott Barber:** *Testing Lessons From Civil Engineering*	*Vendor Presentations*	
3:45 – 4:00 PM	Break				
4:00 – 5:00 PM	AST Membership Meeting				

Post-Conference Tutorial: Day 4 – Thursday, July 17, 2008

Time	Room: Terrace 3rd Floor	Health Sciences Building of the University of Toronto - 155 College Street - Room 106
7:30 – 9:00 AM	Continental Breakfast	
9:00 – 10:30 AM	**Jerry Weinberg:** *The Tester's Communication Clinic*	**Live! AST Instructors' Orientation Course Jumpstart Tutorial** Hosts: Cem Kaner, Becky Fiedler, Scott Barber
10:30 – 10:45 AM	Break	
10:45 AM - Noon	**Jerry Weinberg**: *Continued*	**Live! AST Instructors' Orientation Course Jumpstart Tutorial** *Continued*
Noon - 1:00 PM	Lunch	
1:00 - 2:45 PM	**Jerry Weinberg:** *Continued*	**Live! AST Instructors' Orientation Course Jumpstart Tutorial** *Continued*
2:45 - 3:00 PM	Break	
3:00 – 5:00 PM	**Jerry Weinberg:** *Continued*	**Live! AST Instructors' Orientation Course Jumpstart Tutorial** *Continued*

Sponsors

The following organizations are generous sponsors of CAST 2008.

Gold Sponsors

DevelopSense

Developsense, founded and run by Michael Bolton, is based in Toronto, Canada. We teach provide software testing, consulting, and coaching services all over the world.

www.developsense.com

LogiGear Software Testing

www.logigear.com

LogiGear offers innovative software testing services, including integrated test automation and global resource solutions. Since 1994, LogiGear has created unique solutions that specifically meet the needs of clients in many industries, ranging from Fortune 500 companies to early-stage startups. With facilities in the US and Asia, LogiGear's solution helps companies increase test coverage, improve software quality, lower testing costs and speed time-to-market. For more information, visit http://www.logigear.com.

Skytap VIRTUAL LAB

www.skytap.com

Skytap is the leading provider of cloud-based virtualization solutions available as secure, on-demand services over the Web. Skytap solutions enable IT and development teams to rapidly develop, test, deploy and manage applications in a virtual environment, dramatically increasing their ability to respond to business needs and shorten time to market, while reducing overhead and cost.

Skytap Virtual Lab is an automated virtual lab management solution available as a service. Skytap's Virtual Lab offers organizations the ability to provision virtual labs in minutes, automate the set-up and tear-down of complex, multi-tiered environments, and better collaborate across globally distributed teams using shared virtual infrastructure.

RBCS
TIME TESTED. TESTING IMPROVED.
www.RBCS-US.com

www.rbcs-us.com

Rex Black Consulting Services, Inc. (RBCS) delivers insight and confidence to companies, helping them get quality software and hardware products to market on time, with a measurable return on investment. RBCS is both a pioneer and leader in quality hardware and software testing - through ISTQB and other partners we strive to improve software testing practices and have built a team of some of the industry's most recognized and published experts.

RBCS' team of international consultants deliver customized training, consulting, and outsourcing services to companies that are looking to improve their software testing and quality assurance processes. Companies that have leveraged the RBCS team have reduced development and support costs while assuring the best quality products are delivered to customers.

SOFTWARE QUALITY ENGINEERING

www.sqe.com

Software Quality Engineering assists software professionals and organizations interested in boosting productivity, improving software practices, delivering more customer value, and increasing ROI. Software Quality Engineering hosts three of the industry's most recognized software quality conferences including the STAR conference series and the Better Software Conference & EXPO. Offering a large variety of software training, Software Quality Engineering delivers software improvement methodologies to organizations both large and small. Our team of internationally recognized experts, with hands-on software testing and development experience, develop courses to help both software teams and organizations improve the bottom line. We also produce Better Software magazine and StickyMinds.com-the most comprehensive resources for helping you produce better software-and provide research through our various eNewsletters, books, and other publications. Software Quality Engineering arms you with the power of information.

SearchSoftwareQuality.com
Helping you develop, deploy and manage quality software.

www.SearchSoftwareQuality.com

Ensuring software quality requires more than just identifying and fixing bugs - it requires use of the correct methodologies, processes and tools throughout the software development lifecycle. SearchSoftwareQuality.com's mission is to educate Application Development Managers, Test/QA managers, Software Project Managers and Business Analysts on the key activities, processes and tools needed to produce and deliver robust, high quality software on time, to specification and within budget.

tacit knowledge

www.tacitknowledge.com

Tacit Knowledge is an enterprise software consultancy with offices in San Francisco, New York, Moldova, and Guadalajara. We develop custom applications for both Fortune 500 and startups, with a particular emphasis on software stabilization and optimization, continuous deployment, and agile program management.

TESTED & APPROVED QA CONSULTANTS

QA Consultants is the largest Quality Assurance and Testing consultancy in Canada and one of the biggest and most established in North America. This year alone QA Consultants has already successfully delivered over 250 projects. NEW – Many Fortune 1000 companies are now taking advantage of QA Consultants latest offering - Local Testing resources at Offshore prices.

www.QAconsultants.CA

Silver Sponsors

www.edistatesting.com

Edista Testing Institute
Edista Testing Institute is setup as an academic intervention built for enabling talent acquisition and talent transformation in Software Testing. Our core focus is to enable interventions of learning, certifications, assessment, and community development, and thus furthering the development of the Software Testing Ecosystem. The institute focuses on creating products, providing services using the learning products and engines, and amplify the same with the use of technology.
ETI also contributes to the development of the ecosystem through an online community Test Republic.

www.mentora.com

Mentora Group, INC. tests, hosts and manages business applications. We specialize in performance testing of eCommerce sites and ERP suites from Oracle, PeopleSoft and SAP using commercial and open-source tools. We host and manage business-critical applications in eCommerce, Healthcare, Insurance and Oracle Apps. Headquartered in Atlanta, we have offices in Boston and Washington DC.

Sirius Software Quality Associates provides a one of a kind suite of tools designed from the ground up to support exploratory and manual testing. Their complete end to end solution, TestExplorer, covers all aspects of manual testing, from charter management to test recording and execution, with integrated defect management, metrics, and trend analysis. Sirius SQA also provides consulting and training in support of all aspects of software testing.

www.sirius-sqa.com

karen n. johnson

www.karennjohnson.com

Karen N. Johnson, Software Testing Consultant
Karen N. Johnson is a software testing consultant located in Chicago, Illinois. She has extensive experience in software testing and test management. She serves as a Director for the Association for Software Testing and is a program co-chair for the 2007 Conference for the Association for Software Testing.

www.perftestplus.com

PerfTestPlus offers advising, consulting and training services as well as resources to bring software testing expertise and thought-leadership to organizations seeking to push their testing beyond "state-of-the-practice" to "state-of-the-art." Our testing services are designed and delivered by name-brand testing professionals who subscribe to our integrity driven, value focused philosophy."

softwaretestinghelp.com
passion for testing...

SoftwareTestingHelp.com is a blog purely focused on software testing and quality assurance. We have many articles on manual and automation testing, helping testing professionals in their day to day testing life. SoftwareTestingHelp.com is now having worldwide reader base of more than 4000 software testing professionals. This is a testers community helping each other to advance in their career and learning advanced software testing practices.

www.softwaretestinghelp.com

testingReflections.com

testingReflections.com is your one-stop for **software testing blogs**, aggregating many of the best blogs and articles on software testing from around the web into one convenient place. Established in 2004, testingReflections.com now contains nearly 4000 blog articles covering functional testing, performance testing, unit testing, test-driven development and just about all things 'Software Testing'...

> *"testingReflections keeps up with the most interesting testing blogs so I don't have to. I come for the feeds, and stay for the comments."* **- *James Bach***

Workroom Productions

Workroom Productions is a London-based consultancy, formed in 1994 by James Lyndsay, specialising in software test strategy.

www.workroom-productions.com

TEJAS Software Consulting

Tejas Software Consulting
Danny R. Faught, proprietor of Tejas Software Consulting, is an independent consultant, trainer, and author who helps organizations manage the quality of their software. Danny focuses on efficient exploratory testing and practical test automation. He is also the maintainer of testingfaqs.org, a resource for test tools, conferences, courses, and consultants.

http://tejasconsulting.com/

Bronze Sponsors

Centro de Ensayos de Software

www.ces.com.uy

The CES blends academic methods, proven industry practices, and the experiences of their staff and alliances to deliver exceptional software testing services including:
Consulting: We'll advise your company in testing process improvement, testing strategy, test automation.
Training: We develop and deliver custom training for undergarduates, postgraduages, and professional teams in various areas of software testing.
Testing: We'll test your applications in an effective and deliberate manner based on your context and objectives while focusing on your business risks and needs.
At the CES, we are dedicated to leveraging all of the resources available through our staff, the university, and our allilances to not only exceed our clients expectations on individual contracts, but to look beyond individual contracts to build lasting relationships that benifit both the CES and our clients.

Solution Delivery Partners Inc.

www.sdp-inc.com

SDP takes a simple approach to delivering client success: Educate - Demonstrate - Execute. We provide a unique educational experience for IT professionals by breaking away from traditional delivery formats. While SDP instructors will deliver the fundamentals of IT throughout the course of training, we specialize in interactive, customized workshops to suit your needs. We then provide hands-on mentorship programs to help individuals apply knowledge gained to real-world circumstances. Finally, through our strategic partnerships we can assist organizations in critical areas such as designing, building and delivering business-driven technology solutions, software tool evaluations and support (HP Mercury, IBM Rational & Compuware), and web design, development and testing. SDP is committed to being your partner in success.

Inflectra™

www.inflectra.com

Inflectra™ is a privately held software company dedicated to helping our customers - large corporations, small businesses, professional services firms, government agencies and individual developers – with the means to effectively and affordably manage their software development lifecycles, so as to decrease the time to market and increase return on investment.

SpiraTest™ is a powerful Quality Assurance and Project Management solution that manages a software project's requirements, scope, use-cases, tests, releases and bugs and issues in one environment, with complete requirements traceability throughout. Customers can track every bug or issue back to the test case and test step that being executed, and from there back to the underlying requirement needing fulfillment.

TestersDesk.com from ValueMinds Solutions

www.testersdesk.com

TestersDesk.com is an Online Toolbox/Platform for Software Testers. It provides Online Tools FREE to use, that testers can leverage to increase Test Productivity in the areas of Test Design/Construction, Test Data Generation, Test Diagnosis and many other miscellanous areas of Test Engineering.

The philosphy of www.testersdesk.com is to research-innovate-implement-consolidate a series of system-independent online tools that Software Testers can use, there by creating a one-stop tool platform for the betterment of Software Test Engineering. TestersDesk.com is a complimentary function to everything and every tool in Software Testing, and intends to provide lateral support.

The Online Platform is developed by ValueMinds Solutions, that also performs consultation services and training (through the name TECHTEACH) in Hyderabad, India.

Software Test & Performance Magazine

www.stpmag.com.com

Software Test & Performance is the leading magazine for test/QA professionals. STP is published monthly and reaches more than 25,000 software development managers, project and team leaders, and Test & QA managers. STP helps readers improve the efficiency of their individual and teams software QA and testing processes, as well as help them improve the performance of their in-house applications pre- and post-deployment. Apply for a FREE subscription at www.stpmag.com.

NVP Software Testing

www.nvp-inc.com

NVP Software Testing, Inc., provides industry-leading software quality assurance and testing services, customized to address client's business requirements. These services are provided by qualified talent utilizing modern industry standards and proven best practices to provide exceptional value to clients. NVP offers Consulting, Management, Automation, and Training and Mentoring services.

www.quardev.com

Quardev Laboratories (www.quardev.com) is an outsource software testing and technical writing company in Seattle. Quardev focuses on onshore solutions for rapid exploratory and testing and technical writing contracts. Jon Bach, a recognized expert in exploratory testing and Test Lead management, is Quardev's Corporate Intellect Manager and lead consultant.

Listed in order of appearance:

Keynote Speaker Bios

Gerald M. Weinberg

Fifty years ago, in 1958, Jerry established the very first separate software testing group, to aid in producing life-critical software for Project Mercury. Jerry will speak of many steps, done and not yet done, needed to complete the task of creating a true software testing profession.

For the last 50 years, Gerald Weinberg has worked on transforming software organizations. For example, in 1958, he formed the world's first group of specialized software testers. He is author or co-author of may articles and books, including The Psychology of Computer Programming and the 4-volume Quality Software Management series. He is perhaps best known for his training of software leaders, including the Amplifying Your Effectiveness (AYE) conference and the Problem Solving Leadership (PSL) workshop.

Robert Sabourin

Robert Sabourin has been involved in all aspects of development, testing and management of software engineering projects since graduating from McGill University in 1982. He is the Director of Research and Development at Purkinje Inc, a Montreal based International firm specialized in developing medical software. Robert was the Manager of Software Development at Alis Technologies for over ten years. He has built several successful software development teams and champions the implementation of "light effective process" to achieve excellence in delivering timely commercial quality software solutions.

Robert is a frequent guest lecturer at McGill University where he relates theoretical aspects of Software Engineering to real world examples and demonstrations. Recently Robert has completed a short book illustrated by his daughter Catherine. I Am a Bug (ISBN 0-9685774-0-7) uses the style of a children's book to explain elements of the software development process in a fun, easy-to-read format.

Robert has been the author of several papers and presentations relating to software development at a number of international conferences.

Cem Kaner, JD, PhD

Cem Kaner is a Professor of Software Engineering at the Florida Institute of Technology. He is the senior author of three books: Testing Computer Software; Lessons Learned in Software Testing; and of Bad Software. He also leads the AST-BBST course series project. Prior to going back to school in 2000, Dr. Kaner worked in Silicon Valley for 17 years as a programmer, tester, technical writer, human factors analyst, salesperson, attorney, manager (testers, writers, programmers, projects), director, development consultant, and free-lance teacher. He holds a Ph.D. (in human experimental psychology), a J.D. (law degree), and a B.A. in No Declared Major (mainly math and philosophy). For his work on the law of software quality, he was elected to the American Law Institute in 1999.

Brian Fisher

Brian Fisher is an Associate Professor in the School of Interactive Arts and Technology (SIAT) at Simon Fraser University and Associate Director of the Media and Graphics Interdisciplinary Centre (MAGIC) at the University of British Columbia. He is also a member of the SFU Centre for Interdisciplinary Research in the Mathematical and Computational Sciences, and the UBC Brain Research Centre and Institute for Computing, Intelligent and Cognitive Systems. His research focuses on the cognitive science of human interaction with information systems, with the goal of developing new theories, methods, and methodologies for development and evaluation of technology to support human understanding, decision-making, operation management, and collaboration. This is done in collaboration with the US National Visualization and Analytics Centre and its regional centres for applications in disaster relief and anti-terrorism and with the Boeing Company on understanding aircraft safety, reliability, and maintainability data. In addition to his SIAT courses Brian has taught in Business, Computer Science, Engineering, Kinesiology, and Psychology and is currently collaborating with the SFU Business School to build an interdisciplinary graduate curriculum in visual analytics.

Session Speaker Bios

Martin Taylor

Martin Taylor has been a software developer for 30 years. For the past 6 years he has specialized in the development of Automated Testing Frameworks. He is currently the Sr. Test Automation Specialist in the Engineering Services group at Texas Instruments calculator division.

Michael Bolton and Jonathan Kohl

Michael Bolton has been teaching software testing on five continents for eight years. He is the co-author (with senior author James Bach) of Rapid Software Testing, a course that presents a methodology and mindset for testing software expertly in uncertain conditions and under extreme time pressure. He is also the Program Chair for TASSQ, the Toronto Association of System and Software Quality, and a co-founder of the Toronto Workshops on Software Testing. He has a regular column in Better Software Magazine, writes for Quality Software (the magazine published by TASSQ), and very sporadically produces his own newsletter. Michael lives in Toronto, Canada, with his wife and two children. He can be reached at mb@developsense.com, or through his Web site, http://www.developsense.com.

Since 1998, Jonathan Kohl has worked on a variety of software development projects, mostly as a software tester. He consults on software testing, and teaches testing skills to a variety of audiences. He is dedicated to pushing the craft of software testing forward. He is equally at home working on software development projects, or training, teaching and mentoring testers who are looking to improve their skills. He also speaks on software testing and development issues for software conferences, workshops, and user group meetings. He supports and contributes to Open Source testing tools. He has a website at http://www.kohl.ca.

Diane Kelly and Rebecca Sanders

Diane Kelly is an assistant professor in the department of mathematics and computer science at the Royal Military College of Canada in Ontario. She has over twenty years of software development experience in industry, mainly in the areas of nuclear power generation and bulk power transmission. Most of her work has been with safety-related software. Her research interests blend her industrial experience with software engineering in looking for useful and useable software engineering methods to improve the quality and maintainability of engineering and scientific based software.

Rebecca Sanders recently completed a master's degree in computer science at Queen's University in Ontario. Her thesis was The Development And Use of Scientfic Software. Diane Kelly was one of her thesis supervisors.

Doug Hoffman

Douglas Hoffman has over thirty years experience in software quality assurance and has earned degrees in Computer Science, Electrical Engineering, and an MBA. He is currently employed by Hewlett-Packard as a QA Program Manager. He is a Founding Member and a past Director of the Association for Software Testing. He has been a participant at dozens of software quality conferences and Program Chairman for several international conferences on software quality. He was among the first to earn a Certificate from ASQ in Software Quality Engineering (ASQ-CSQE), has been certified in quality management (ASQ-CQMgr), and is an ASQ Fellow. He is active as a Fellow of the ASQ, participating in the Silicon Valley Section, Software Division, and the Software Quality Task Group (SSQA), and is also a member of the ACM and IEEE. He is current Auditor and Past Chairman of the SSQA and is the Immediate Past Chairman of the Silicon Valley Section of the ASQ.

Jerry Kominar

Jerry Kominar graduated from the University of Guelph with a Bachelor of Arts Honours Degree (Co-op) double majoring in Computer Information Science and Fine Arts. He is currently working as a QA Team Lead at Research in Motion in Waterloo. His team tests public key infrastructure and other security related products for the BlackBerry Handheld. He has been in QA for approximately 4.5 years.

Morven Gentleman

Morven Gentleman is a professor in the computer science department of Dalhousie University in Halifax. His research areas include software engineering, concurrency, computer architecture, and mathematical software.

Adam Geras and Steve Richardson

Adam Geras has been in the IT industry for 19 years as a developer, architect, researcher, tester, and test manager. His research at the University of Calgary centred on test-driven development from both the developer and customer perspectives. Most recently Adam has been keen on using dynamic languages for scripted and unscripted testing and is the author of PSExpect, an open source testing framework based on Powershell, Microsoft .NET-based scripting language for system administrators (and testers!). His job at Ideaca as chief methodologist is to promote project health awareness and project wellness, with a testing slant.

Steve Richardson also works at Ideaca.

Adam White

Adam White is a manager of Test Engineering and Escalations at PlateSpin Ltd. At PlateSpin they test their products through the use of context driven testing; meaning they use the right techniques and tools when it is the right time. He is continually developing his ability to find flaws in enterprise software and deliver facts to stakeholders. He am a student of the context driven school of testing under the tutelage of Michael Bolton and James Bach. He just got married and has a web site at http://www.adamkwhite.com.

Bart Broekman

Bart Broekman has been a software test practitioner since 1990. He started his testing career at Philips Data Systems in the test team of an operating system kernel. Five years later he joined Sogeti where he fulfilled assignments ranging from test automation to organising and managing large test projects. He participated in European embedded software research projects (ITEA) and is co-author of a book on test automation and testing embedded software.

Adam Goucher

Adam Goucher has been testing software for the last 10 years. In that span he has worked at a range of companies ranging from one of the big-5 Canadian banks to a start-up. Recently he has been running organizations' QA / Test departments and helping them improve their processes from a Quality perspective. Currently he is doing that for The Jonah Group. He is heavily influenced by the Context-Driven school of testing, though tries to keep out of the politics of the various factions of the testing world. He has a website Quality through Innovation at http://adam.goucher.ca.

Scott Barber

Scott Barber is the Chief Technologist of PerfTestPlus, Executive Director of the Association for Software Testing Co-Founder of the Workshop on Performance and Reliability and co-author of *Performance Testing Guidance for Web Applications*. Scott thinks of himself as a tester and trainer of testers who has a particular passion for performance testing software systems. He is an international keynote speaker and author of over 100 articles on software testing. He is a member of ACM, IEEE, American MENSA, the Context-Driven School of Software Testing and is a signatory to the Manifesto for Agile Software Development.

Tutorial Speaker Bios

Gerald M. Weinberg

See the Keynote above.

Scott Barber

See the Session Speaker above.

Julian Harty

Julian Harty has lots of experience in software testing and for the last year has worked for Google as a senior QA Engineer. He is passionate about improving software quality, and how software testing fits as a part of software quality. He is a frequent author and speaker at testing conferences.

Hung Nguyen

Hung Nguyen is CEO, President, and Founder of LogiGear and is responsible for the company's strategic direction and executive business management. He's been a leading innovator in software testing, test automation, testing tool solutions and testing education programs for the last two decades.
Mr. Nguyen is coauthor of the top-selling book in the software testing field, Testing Computer Software (Wiley, 2nd ed. 2002) and other publications including Testing Applications on the Web (Wiley, 2nd ed. 2003). His experience over the past two decades includes leadership roles in software development, quality, product and business management at Spinnaker, PowerUp, Electronic Arts, Palm Computing and other leading companies. A frequent speaker at industry events and a contributor to many industry publications, Nguyen also teaches software testing at LogiGear University, and at the University of California Berkeley Extension and Santa Cruz Extension in San Francisco and Silicon Valley.

Program: Day 1 – Monday, July 14, 2008

Time	Room: Terrace 3rd Floor	Room: Saint David 3rd Floor	Room: Saint Patrick North 3rd Floor	Room: Saint Patrick South 3rd Floor	Room Colony West
7:30 – 9:00 AM	Continental Breakfast (Room: 3rd Floor Foyer)				Registration
9:00 – 10:30 AM	**Jerry Weinberg:** *The Tester's Communication Clinic*	**Scott Barber**: *Performance Testing Software Systems*	**Julian Harty**: *Mobile Wireless Test Automation*	**Hung Nguyen**: *From Craftmanship to Leadership*	
10:30 – 10:45 AM	Break (Room: 3rd Floor Foyer)				
10:45 AM - Noon	**Jerry Weinberg:** *Continued*	**Scott Barber**: *Continued*	**Julian Harty**: *Continued*	**Hung Nguyen**: *Continued*	
Noon - 1:00 PM	Lunch (Room: 3rd Floor Foyer)				
1:00 - 2:45 PM	**Jerry Weinberg:** *Continued*	**Scott Barber**: *Continued*	**Julian Harty**: *Continued*	**Hung Nguyen**: *Continued*	
2:45 - 3:00 PM	Break (Room: 3rd Floor Foyer)				
3:00 – 5:00 PM	**Jerry Weinberg:** *Continued*	**Scott Barber**: *Continued*	**Julian Harty**: *Continued*	**Hung Nguyen**: *Continued*	
5:00 – 7:00 PM	Break				
7:00 – 9:00 PM	Birds of a Feather Proposal: Teaching of Software Testing (Cem Kaner, Becky Fiedler)	AST SIG Meetings (SIG Leads)	Get your questions answered about AST (Michael Kelly)	Open Space	

Tutorials

Jerry Weinberg: The Tester's Communication Clinic

Working with hardware and software is only half of the professional tester's job–and though hard enough, it's not nearly as hard as the other half, working with people. In this interactive workshop, you'll obtain new strategies for coping with your most serious communication problems–with managers, developers, customers, and other testers. Gerald Weinberg helps you affirm your most successful strategies, while sharing other techniques you may not have thought of. Learn to be more sensitive to management desires and more influential in obtaining effective developer responses. Real-life communication situations of the participants will be used to illustrate practical application of various communication models, including:

- The Satir Interaction Model
- The Congruence Model
- Personality Types
- Modality Preferences

Scott Barber: Performance Testing Software Systems: Analyzing Performance Test Data

Performance Testing frequently generates very large volumes of data. That data usually requires significant analysis before findings are made and recommendations are delivered. To make matters more complex, even though there is a large volume of data, there are typically an insignificant number of tests conducted for most data reduction methods to be statistically valid. Finally, many of the statistical methods that are frequently used are either mis-used or mis-understood.

One of the Performance Testing Software Systems (PTSS) series of workshops, Analyzing Performance Test Data is targeted for anyone who analyzes performance test results data. It focuses on how to make sense out of performance test data to improve findings and recommendations to help achieve business objectives, reduce project risk, and avoid bad press. Further, it teaches methods for visually reporting results of performance tests that are less prone to misinterpretation than reporting complex statistics the audience is unlikely to understand. Finally, this workshop provides you with the knowledge you need to use statistics correctly to help you understand the data.

PTSS is a unique series of workshops that employ heuristic approaches to performance testing that focus on mitigating risks to the business and satisfying end users in commercially driven software development environments. This approach marries the software testing insights of James Bach, Rob Sabourin, Cem Kaner and many other members of the Context-Driven School of software testing with the performance testing insights of Alberto Savoia, Ross Collard, Roland Stens, and the rest of the WOPR (Workshop On Performance and Reliability) community. The approach has a track record of success with regard to adequately mitigating business risk in time to keep pace with the commercial aspects of the project. The Microsoft patterns & practices book *Performance Testing Guidance for Web Applications* by J.D. Meier, Scott Barber, Carlos Farre, Prashant Bansode, and Dennis Rea complements the material presented in this workshop.

Julian Harty: Mobile Wireless Test Automation

Automated testing is becoming a generally accepted "good practice" suitable for many situations. There are plenty of tools, frameworks and practices which are used throughout the software industry. However, automated testing of mobile wireless applications is not yet mature, particularly when the testing includes 100"s of different, disparate devices, multiple natural languages, network operators, etc.

This tutorial is based on current experiences and practices at Google and will explain some of the challenges, difficulties and even the successes gained over the last 12 - 18 months of hands-on test automation of mobile wireless applications. The scope of test automation includes:

- Web applications
- J2ME applications
- Native applications (e.g. Symbian, Blackberry, Windows Mobile)
- Testing of the servers that support the devices
- Rendering issues
- Performance testing

Key Points:

- Learn about some of the unique challenges of test automation for mobile wireless applications. Learn how some of these automation challenges can be addressed.
- Be prepared to get involved in the tutorial and share your problems and experiences with a group of peers.

Hung Nguyen: From Craftmanship to Leadership

From a business perspective, every company is in the business of creating products or services. In turn, those products and services generate revenue, and hopefully, profit after expenses. From a business viewpoint, it's about money and results. So when we talk about quality, it is essential to consider quality in its financial context. (Of course there is non-financial value such as human life affected by poor quality product but we won't talk about that here). Software testing plays a key role in improving the quality of software-related products and services. On the flipside, the cost of testing software can reach up to 40% of the overall software development cost (based on our internal study). As a testing professional, what can we do to move beyond our craft, and contribute as a leader? In that leadership role, we can potentially maximize our effectiveness in helping the business deliver profit and result.

Much has changed in software testing over the past two decades, yet many of the principles stay the same. In this tutorial, we will discuss a "macro" approach to software testing. We will be focusing on leadership skills and thinking out-of-the-box in the context of software testing. Most of my professional career is in the software industry. While I have worn many hats throughout, from testing to programming, software test management to software development management, technical management to business, studying to teaching, software consumer to designer, virtually all facets of my work involve software testing. I have also had experience in building and running a company of software testing professionals from a hundred-fifty staff (dot-com), to twenty staff (dot-bomb), and now to four hundred staff (globalization); opportunities of working with and testing for hundreds of companies and different products in various industries, interfacing with software testers, business analysts, developers, managers, and C-level executives, I want to share with you what I've learned and am still learning about developing leadership skills in software testing.

Program: Day 2 – Tuesday, July 15, 2008

Sessions	Room / Location				
Time	**Room: Colony Ballroom 2nd Floor**	**Room: Saint David 3rd Floor**	**Room: Saint Patrick 3rd Floor**	**Room: Terrace 3rd Floor**	**Room: Colony West**
7:30 – 8:45 AM	Continental Breakfast and Networking (Room: 3rd Floor Foyer)				
8:45 – 9:00 AM	Opening Remarks				Registration and Sponsors booths
9:00 -10:30AM	**KEYNOTE - Gerald M. Weinberg:** *Lessons from the Past to Carry into the Future*				
10:30 – 10:45 AM	Break (Room: Colony West)				
	Speaker Introductions:				
10:45 – 11:45 AM	**Martin Taylor**: *Visualization and Statistical Methods*	**Michael Bolton and Jonathan Kohl:** *Testing and Music*	**Diane Kelly and Rebecca Sanders:** *The Challenge of Testing Scientific Software*	*Vendor Presentations*	
11:45 – 1:00 PM	Lunch				
1:00 – 2:30 PM	**KEYNOTE - Robert Sabourin:** *Applied Testing Lessons from Delivery Room Labor Triage*				
2:30 – 2:45 PM	Break (Room: Colony West)				
	Speaker Introductions:				
2:45 – 3:45 PM	**Doug Hoffman:** *Lessons for Testing from Financial Accounting*	**Jerry Kominar:** *Sleight-of-Quality*	**Morven Gentleman:** *Measuring File Systems*	*Vendor Presentations*	
3:45 – 4:00 PM	Break (Room: Colony West)				
4:00 – 5:00 PM	AST Board of Director's Election				
5:00 - 6:00 PM	AST Board Of Director's Meeting				

Keynotes

Lessons from the Past to Carry into the Future – Jerry Weinberg

Fifty years ago, in 1958, Jerry established the very first separate software testing group to aid in producing life-critical software for Project Mercury. Jerry will speak of many steps, done and not yet done, that he believes are needed to complete the task of creating a true software testing profession.

Applied Testing Lessons from Delivery Room Labor Triage – Robert Sabourin

This talk presents several labor triage examples from recent cases at the Royal Victorial Hospital in Montreal Canada. The authors walk through these experiences and draw parallels to software testing triage including decision making about bugs (assessing severity, criticality, establishing priority), focusing testing and requirement change management. Cases presented illustrate circumstances in which triage nurses drop existing protocols and use their own intuition to guide decision making assessment and action in critical cases.

Association for Software Testing

CAST 2008: Beyond the Boundaries

July 14-16, 2008 – Toronto, Canada

http://www.cast2008.org

Session

Visualization and Statistical Methods
In High Volume Test Automation of Embedded Devices

Martin Taylor

Tuesday, July 15, 2008
10:45am - 11:45am

Room: Colony Ballroom 2nd Floor

Overview

This presentation is an industrial experience report from Texas Instruments Education Technology (TI Calculators) division. Martin will describe how his team combined high volume test automation techniques, data visualization, and statistical regression methods to detect memory leaks, device crashes, and performance problems in the new TI-Nspire(tm) math and science learning handhelds. Martin also reports how these techniques have been used at TI to compare results between various TI-Nspire(tm) software builds.

Visualization and Statistical Methods in High Volume Test Automation of Embedded Devices

C. Martin Taylor
Texas Instruments Inc.
Education Technology (Ed. Tech.)
7800 Banner Drive, MS 3908
Dallas, TX 75251, USA
1-214-567-2249

cmtaylor@ti.com

ABSTRACT

This paper is an Industrial Experience Report from Texas Instruments Education Technology (TI Calculators) division. It describes how we combined High Volume Test Automation (HVTA) techniques, Data Visualization, and Statistical Regression methods to detect memory leaks, device crashes, and performance problems in the new TI-Nspire™ math and science learning handhelds. The paper also reports how these techniques have been used at TI to compare results between various TI-Nspire™ software builds.

1. INTRODUCTION

1.1 High Volume Test Automation & RRR

High Volume Test Automation (HVTA) techniques involve the execution of large numbers of automated tests that already pass in order to expose longevity errors that are generally hard to find. [KanerMcGee04] Kaner and his co-authors reported in a keynote address to STAR East [KanerBondMcGee04] that the HVTA technique of Extended Random Regression (ERR) was used successfully at "Mentsville" to find timing problems, memory corruption (including stack corruption), and memory leaks. This technique of repeatedly executing a set of already passing automated tests in a randomized fashion, renamed "Repeated Random Regression" (RRR) by McGee and Taylor, has been in use at Texas Instruments Ed. Tech. division since mid-2004.

1.2 RRR Testing of TI-Nspire™ Devices

The TI-Nspire™ and TI-Nspire™ CAS (Computer Algebra System) math and science learning handhelds [TIEdTech08], illustrated in Figure 1, are embedded devices that contain a custom ASIC with an integrated ARM 926 processor. The TI-Nspire™ devices have 32 Mb of RAM for program execution and 32 Mb of Flash memory for persistent storage. They have a USB port that can be used to communicate with a PC, a scientific measurement probe or another TI handheld device.

2008 Annual Conference of the Association for Software Testing (CAST), July 14-16, 2008, Toronto, Ontario, Canada.

Figure 1 - TI-Nspire™ and TI-Nspire™ CAS

Since 2003 the Test Automation & Tools team at Texas Instruments Ed. Tech. division has developed a framework known as "TI-CAT" (for "TI Calculator Automated Testing") that enables automated testing of TI calculators. [Taylor05] This framework supports two different varieties of automated testing: System testing at the user interface level and software library testing at the Application Programming Interface (API) level [Taylor06]. This paper discusses RRR techniques applied to automated tests at the user interface level.

For automated testing of the TI-Nspire™ devices we have written a TI-CAT Test Engine that runs on a PC and communicates with a Test Engine Service that runs on the device. This pair of communicating Test Engines allows us to send virtual keystrokes and mouse movements to drive the device and to request screen images and other status information from the device. One such piece of status information is "RAM Available" (i.e. the amount of free RAM memory) available on the device. The Test Engine on the PC side automatically requests this information from the device at the start of every test. Each test appends a row of comma separated value (CSV) data, including the "RAM Available" information, to a summary file that can later be used to generate graphs of "RAM Available".

For testing the TI-Nspire™ devices we have a set of about 50 automated Build Acceptance Tests (BAT) that exercise the device software in a broad but shallow manner. In addition to these BAT tests, each application on the device has a set of regression tests, many of which are also automated. When we do a RRR test run on a TI-Nspire™ device we use all of the automated BAT and regression tests that pass on a given build of the embedded software.

2. APPLICATIONS OF RRR TESTING

2.1 Memory Leaks in the Test Software?

Before using the TI-CAT and TI-Nspire™ test software to detect memory leaks in the embedded software we should prove that the test software itself does not have any memory leaks. To do this we wrote a simple automated test that sent the necessary keystrokes to the TI-Nspire™ device to display the "About" dialog box, captured this screen, then removed the dialog box. This exercises all the parts of the test and communications software but only involves a minimum of functionality in the TI-Nspire™ application software. We ran this test 1000 times and captured the RAM Available at the beginning of each test run. As Figure 2 illustrates, the linear regression slope of RAM Available is very close to zero, indicating no obvious memory leaks in this simple exercise of the test and communications software.

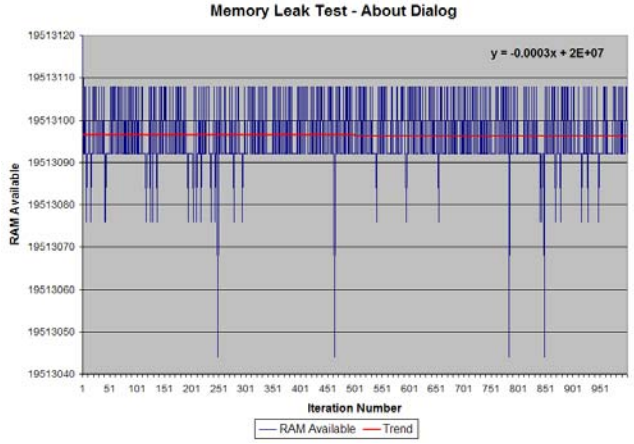


Figure 2 – No Memory Leaks in Test Software

2.2 Detecting Large Memory Leaks

During the TI-Nspire™ software version 1.2 development cycle, we had about 100 automated tests available for RRR testing. These were a mixture of BAT tests and functional tests of the Lists and Spreadsheet (L&S) application. When we plotted the RAM Available data for the very first RRR run we got some interesting results immediately (see Figure 3).

The coloured dots on the graph indicate particular test cases that may be of interest, since they seem to precede an obvious large memory leak. The most obvious of these is the test named "BAT_032". Examining the steps of this test we found that it does a number of things that could be causing a memory leak:

- Runs the Graphs & Geometry (G&G) application
- Runs the Calculator application
- Runs the Lists & Spreadsheets (L&S) application
- Puts the 3 applications in a single-page layout
- Defines functions in G&G & Calculator
- Puts the defined functions into a Function Table in L&S

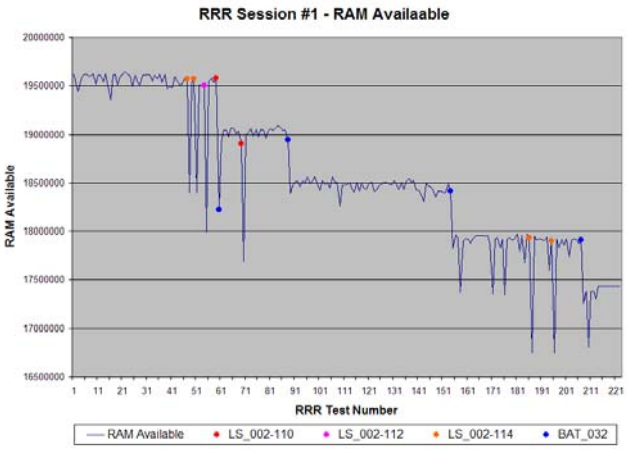


Figure 3 - RRR Testing Session #1

To determine the specific source of the memory leak we modified this test to get & report RAM Available after each step. We then ran the test a few times and plotted the RAM Available (see Figure 4). The red line in Figure 4 seems to show that one specific step has the memory leak.

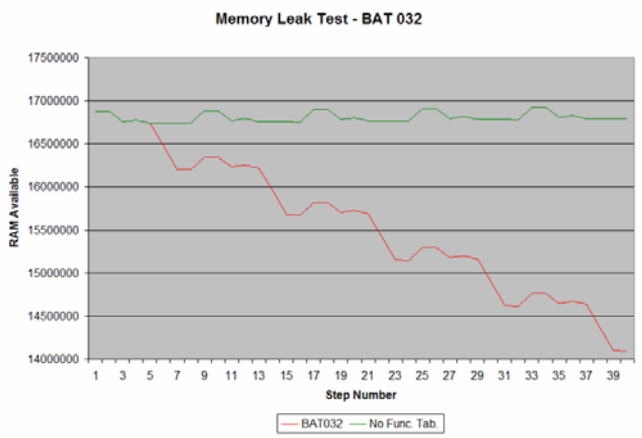


Figure 4 - RAM Available per Step of BAT_032

We used another column in the spreadsheet to remove the RAM Available delta due to the "Function Table" step and graphed this as the green line. This line seems to have a flat trend, so the memory leak is likely in the Function Table feature of the L&S application, but what part of this feature?

- Entering/Leaving Function Table with an empty table?
- Putting functions into a Function Table?

To further isolate the memory leak we wrote some exploratory tests, one to just enter and leave Function Table feature, and another to create 2 simple functions and add them to a Function Table. We ran each of these tests many times and plotted the RAM Available. The "Empty Function Table" test results are shown in Figure 5.

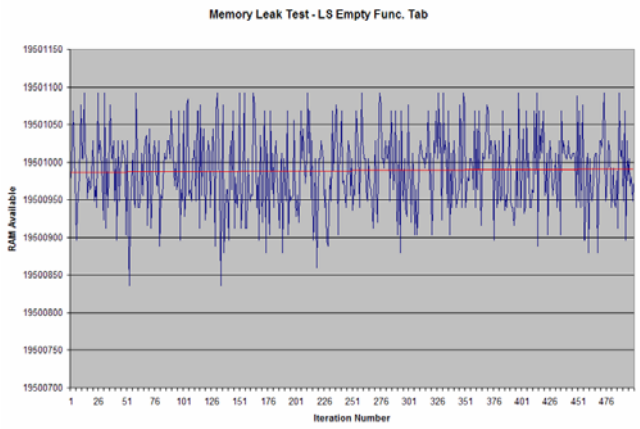


Figure 5 - "Empty Function Table" Memory Usage

The RAM Available pattern here is "noisy", but the trend is definitely flat. The RAM Available plot for the "add 2 Functions to a Function Table" test is shown in Figure 6.

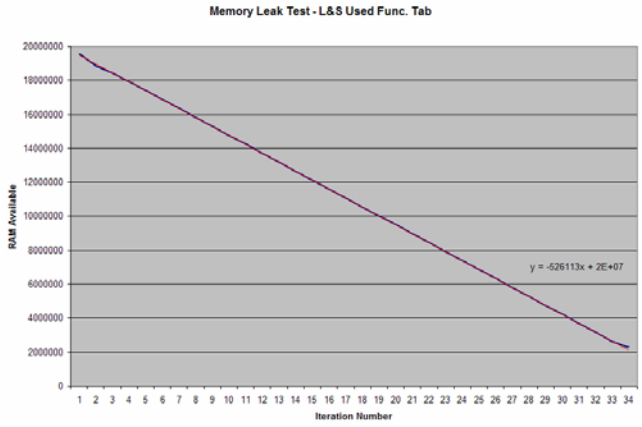


Figure 6 - L&S Function Table Memory Leak

This Graph shows a definite, large memory leak of about 526K per iteration. Providing this data to the developers in a bug report allowed them to quickly find and fix this serious memory leak.

2.3 Detecting Device Crashes

When a software error causes a TI-Nspire™ device to crash, the device automatically re-boots itself. The Test Engine that is driving the device recognizes this by a temporary loss of USB connectivity. The Test Engine then waits for connectivity to be restored after the re-boot before proceeding with the next test. Thus, in an RRR run, the next test after a re-boot takes longer than normal to run (waiting for the re-boot) but then continues as normal.

When a TI-Nspire™ device is rebooted, its "RAM Available" is reset to the maximum allowed by the available physical memory on the device. These maximum values are readily visible on a graph of "RAM Available" vs. "RRR Test Number" as seen in Figure 7.

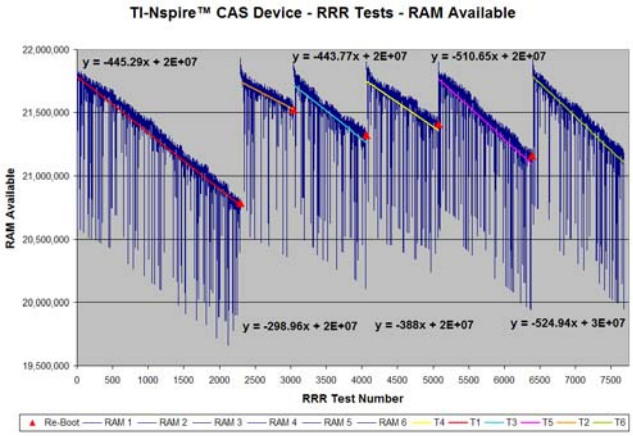


Figure 7 - RRR Results Showing Device Crashes

Looking at the test summary data from this RRR run we were able to determine that every device crash happened during the execution of the same BAT test. Taking the steps from this test, we used exploratory testing techniques to try various sub-sets of the steps until we found a minimal set of steps that caused a device crash. These minimal steps were added to a bug report which assisted the developers in finding and fixing this device-crashing bug.

2.4 Measuring Small Systemic Memory Leaks

The presence of small systemic memory leaks can readily be seen from a plot of RAM Available data from a RRR test run. For example, if we take the RAM Available data before the first re-boot in Figure 7 and plot it we get the graph shown in Figure 8. This shows that the TI-Nspire™ 1.3 software had one or more small systemic memory leaks that averaged a loss of 445 bytes per test iteration.

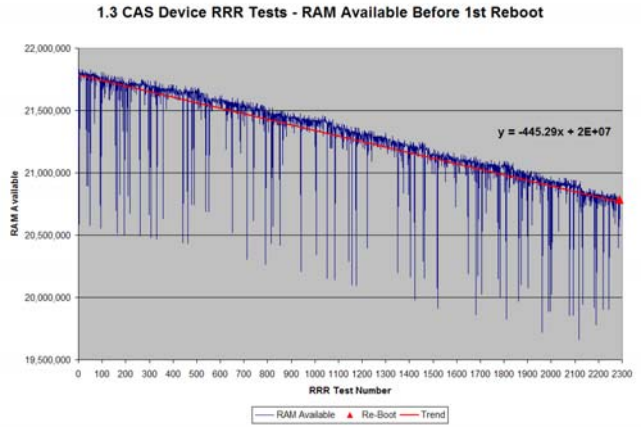


Figure 8 – Small Systemic Memory Leak in TI-Nspire™ 1.3

While these graphical analysis and HVTA techniques do nothing to identify the causes of such small systemic memory leaks, they can be used to identify and measure changes in memory usage between software versions.

During the TI-Nspire™ 1.4 development cycle we worked with the developers of the Lists & Spreadsheet (L&S) application for the TI-Nspire™ to measure changes in memory usage by that

specific application. The first step in this work was to do a RRR test run on the 1.3 version of the software that only tested the L&S application. These results would become the baseline for comparison with similar results from various software builds throughout the 1.4 development cycle. These 1.3 baseline results are shown in Figure 9.

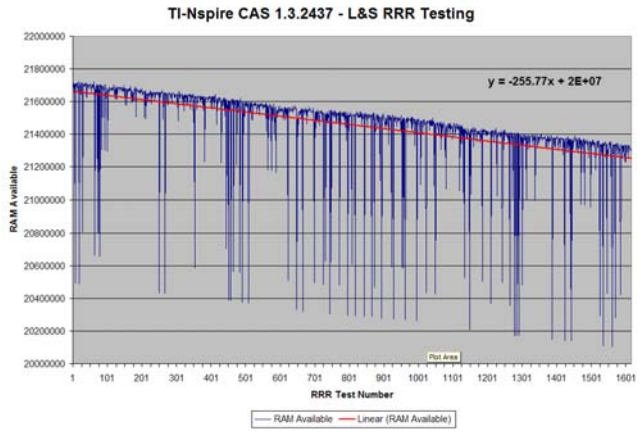


Figure 9 - L&S 1.3 Baseline Memory Usage

There are two statements that can be made about L&S memory usage from this baseline 1.3 graph. One is that there is a small systemic memory leak of about 255 bytes per test, as illustrated by the linear regression trend line. The other is that many of the tested L&S features use a large amount of memory, as illustrated by the deep "spikes" in RAM Available.

The L&S developers were tasked with improving memory usage during the 1.4 development cycle. Early in this development cycle we repeated the same random sequence of L&S RRR tests on the current development build (1.4.2996) and plotted the results of this run overlaid on the results from the baseline run. This is shown in Figure 10 which shows "good news" and "bad news". The good news is that the overall memory usage by the L&S application has been greatly improved. This can be determined from the much shorter "spike depth" in the 1.4.2996 data. The dotted red line tracks the maximum memory usage and can seen to be much better than the similar trend (dotted gray line) in version 1.3.2437. The bad news is that the memory leaks have grown from an average of 255 bytes per test to 601 bytes per test.

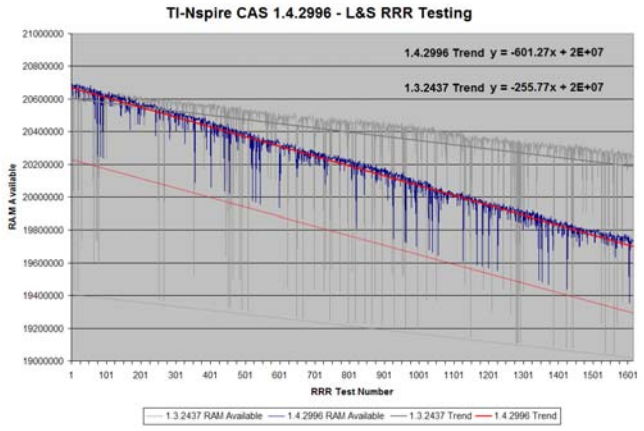


Figure 10 - L&S Memory Usage, Build 1.4.2996

Later in the 1.4 development cycle we ran the same random sequence of L&S RRR tests on the current development build (1.4.7459) and plotted the results of this run overlaid on the results from the baseline and 1.4.2996 runs. The results, shown in Figure 11, show a further deterioration from the 1.3 baseline measurement. Both the overall memory usage, indicated by the "spike depth", and the long term memory usage trend have deteriorated. The memory leak trend is now estimated at 1,293 bytes per test, up from 601 bytes per test in build 1.4.2996.

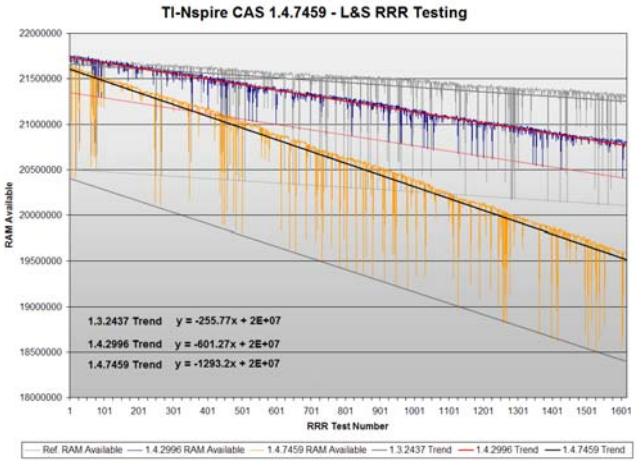


Figure 11 - L&S Memory Usage, Build 1.4.7459

This information is current at the time this paper was submitted for publication and the developers are responding to this news with renewed efforts to find and fix memory leaks in the L&S application. The project still has about 6-8 weeks to run, so any updates to this progress will be presented at the CAST 2008 conference in July.

3. APPLICATIONS OF OTHER HVTA TECHNIQUES

3.1 Detecting Performance Problems

During the investigation of the test that caused a device crash, described in section 2.3, we made an interesting observation about performance. When the TI-Nspire™ saves a document it suggests a file name to the user. The first saved file name suggestion is "Document1". If the user chooses this name, then the next saved file name suggestion is "Document2". The automated test we were using saved its created document using this default file name. As more documents were saved, it seemed to take longer and longer to execute.

To investigate this apparent performance degradation we set up an HVTA test run where the same test was executed repeatedly and the time to complete each execution of the test was recorded. The resulting graph of execution time is shown in Figure 12. The performance of "Save Document" degrades exponentially, as can be seen from the exponential trend line fitted to the data in Figure 12. This test was initially done during the TI-Nspire™ 1.2 development cycle.

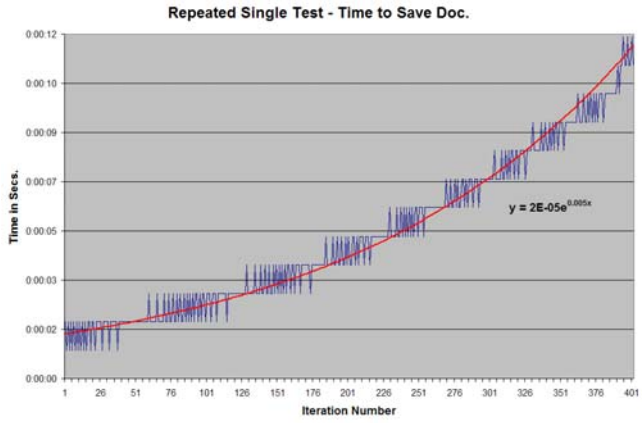


Figure 12 - Repeated "Save Doc." Times

Taking that result as a baseline, we re-ran the same test to measure the performance of this feature during the 1.3 development cycle. Unfortunately the results showed that the "Save Document" performance deteriorated a little between versions 1.2 and 1.3 as illustrated in Figure 13.

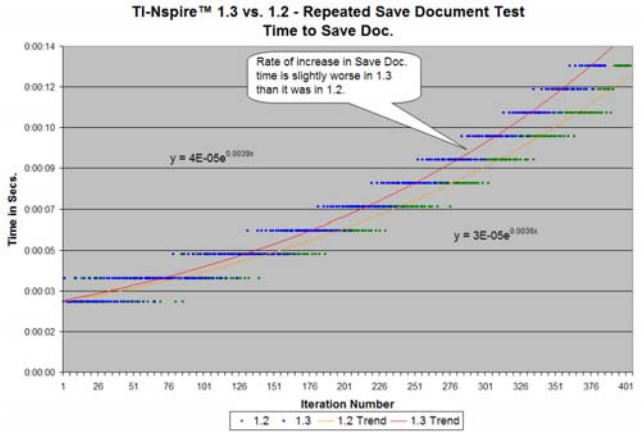


Figure 13 - Save Doc. Time in TI-Nspire™ 1.3 vs. 1.2

The developers have speculated that this may be an issue in the underlying file system used on the TI-Nspire™, but further work is needed to determine and fix the root cause.

4. CONCLUSIONS

4.1 Conclusions

High Volume Test Automation can produce a high volume of numerical result data. We have shown various examples of how graphing and performing statistical analysis on this data can extract information useful to testers and developers of the system under test. The human eye remains the most sophisticated pattern recognition system available to us. By graphing HVTA outputs such as RAM Available and Execution Time, the human viewer can quickly recognize patterns that indicate severe memory leaks, system crashes, small systemic memory leaks and performance deterioration. This graphical presentation, complemented with statistical regression analysis, is a powerful communications tool that enables testers and developers to better understand the behaviour of the system under test.

4.2 Future Directions

We continue to experiment with RRR testing as well as ways of visualizing and analyzing the results. Currently we run the RRR tests on the L&S application about once a week. We plan to do similar RRR runs for memory leak detection and comparison on other individual TI-Nspire™ applications and on the full system.

Currently the post-run analysis of the RRR result data is done manually using spreadsheets so we are considering building automated tools to do this, including graph generation and regression analysis. Similarly any "drill-down" analysis of a single test, like that described in section 2.2, is currently done manually. To automate this, we have a prototype tool that runs a single test a given number of times and automatically captures memory and timing data at every step of the test. We hope to continue to find new ways of leveraging our automated test data to expose valuable information about the systems under test.

ACKNOWLEDGMENTS

The author would like to thank the following people: Pat McGee, former PhD intern at TI Ed. Tech., for introducing me to HVTA techniques and helping me to implement RRR testing; Glen Thornton, Engineering Director at TI Ed. Tech., for allowing and encouraging me to publish this data about the TI-Nspire™ development and testing process; Johnny Schmittou, Software Developer at TI Ed. Tech., for his encouragement and collaboration in applying these techniques to the TI-Nspire™ L&S application; and Adam Goucher and Dan Hoffman for their reviews and feedback during the writing of this paper.

5. REFERENCES

[KanerMcGee04] Cem Kaner & Pat McGee

"Experiments with High Volume Test Automation",
ACM SIGSOFT Software Engineering Notes, Volume 29, Issue 5 (September 2004)
http://www.kaner.com/pdfs/MentsvillePM-CK.pdf

[KanerBondMcGee04] Cem Kaner, Walter P. Bond, Pat McGee

"High Volume Test Automation",
Keynote Address, STAR East (May 20, 2004)
http://www.kaner.com/pdfs/HVAT_STAR.pdf

[TIEdTech08] Texas Instruments, Inc.

TI-Nspire™ corporate website
http://www.ti-nspire.com/

[Taylor05] C. Martin Taylor

"Automated Testing of Embedded Systems with TestDirector and Python"
2005 Texas Instruments Software Symposium (Internal Event)
http://events.sc.ti.com/sw2005/presentations.asp

[Taylor06] C. Martin Taylor

"Automated Testing of Library APIs in the TI-CAT Test Development Kit"
2006 Texas Instruments Software Symposium (Internal Event)
http://events.sc.ti.com/sw2006/presentations.asp

Association for Software Testing

CAST 2008: Beyond the Boundaries

July 14-16, 2008 – Toronto, Canada

http://www.cast2008.org

Session

Testing and Music:
Parallels in Practice, Skills and Learning

Michael Bolton and Jonathan Kohl

Tuesday, July 15, 2008
10:45am - 11:45am

Room: Saint David 3rd Floor

Overview

Many metaphors--besides engineering--can value to learning about patterns and principles in software testing. Music affords such a metaphor. Both testing and music are performed in a variety of contexts, providing different values for different people. Both fields are suffused with traditions; both involve dynamics between scripted and exploratory processes. In this presentation we'll try to discover comparisons and contrasts that might help to advance learning and provide new metaphors for testing.

Testing and Music: Parallels in Practice, Skills, and Learning

Michael Bolton
mb@michaelbolton.net

Jonathan Kohl
jonathan@kohl.ca

Abstract

For years, people have appealed to engineering as the dominant metaphor for how software testing should be done. Yet there are many other metaphors that could provide value to learning about patterns and principles in software testing. As professional testers and non-professional musicians, we observe that music affords such a metaphor. What are the parallels between music and testing—especially exploratory approaches to testing? What can we learn from the similarities?

In this paper, we explore traditions and contexts; structures of performance in music and testing; ideas associated with tension and resolution; the role of scripting and other artifacts in design, performance, and learning; and skills development.

Traditions and Contexts

Both testing and music are performed in a wide variety of contexts, with different audiences, different practitioners, and different values for different people. Both fields involve, to a large degree, socially constructed activities, and are suffused with traditions.

Living traditions depend on tension between three points of view: a classical aspect, which preserves the foundation of the practice, but which views change and diversity as a threat to the purity of the art; a state of the practice, which advances slowly while absorbing some forms of change and resisting others; and an avant-garde, which stretches the limits and boundaries of the state of the art by bringing in influences from outside and synthesizing new forms, but which may not feel beholden to the classical foundations. The classicists and the avant-garde tend to argue with one another, while the middle ground simply proceeds without paying too much attention to the other extremes.

In music, cultures and subcultures abound, cross-pollinating one another. For example, Irish traditional dance music, a subculture of Celtic traditions that also include Scottish, Breton, Welsh, Cape Breton, has at least four dominant sub-styles (Clare, Sligo, Donegal, and Cork/Sliabh Luachra). The blues has Chicago, Memphis and Mississippi Delta traditions, as well as boogie woogie, country blues and blues rock. One example of a veritable hotbed of colliding musical styles is Zydeco, a style of music created in Louisiana. Zydeco began as a fusion of Creole, Cajun (from "Acadian"—itself a blend of French and Celtic styles from Canada's east cost) and traditional American music. It has further evolved to include influences from blues, jazz, gospel, and other popular North American music as well as Caribbean influences such as salsa, rumba and calypso, among others. It's not uncommon to hear hip hop styling or steel drums behind the driving accordion that is a trademark of the genre.

It's important for both testers and musicians to perform in a manner appropriate to the context. Electronic dance music tends not to go over well in a church that favours Gregorian plainchant for its liturgical music. On the other hand, in a supportive context, musical styles can blend, leading to fusion and innovation that can advance new traditions and recall old ones. The startlingly successful Enigma recordings of the early 1990s may have contributed to the success of the fusion of plainsong and chant with dance music during that decade.

Testing has been (somewhat controversially) categorized into schools [Pettichord2003]. We assert that these schools can be seen as analogous to musical traditions. Schools of testing can be strengthened by adding techniques and models associated with other schools. In software testing, opponents to the schools categorization often claim that all testing styles are the same, and that division is unnecessary. In music, diversity is embraced and encouraged, and has led to discovery and growth. In testing, we can also profit from identifying different ideas, styles and interpretations. The recognition of specialization and focuses from each of these areas will lead to more discovery, communities of practice, and growth for the whole software testing community. To deny the differences of ideas in testing is akin to saying that music is only "music" and any attempt to identify differing styles, traditions and genres is divisive and problematic.

Structures

A musical piece has structure whether it is rehearsed or improvised, played from a score or played extemporaneously. Several elements of the piece's structure are determined by time. The *tempo* of the piece refers to its speed or pace. *Rhythm* refers to the duration of a series of notes and the ways in which they are grouped together. A *bar*, or *measure*, is a means of dividing the rhythm into sequences of beats; in most popular Western music, bars in "straight time" tend to have four beats each (bars in "waltz time" have three); these associations and divisions give the piece its *meter*. *Loudness* and *accents* give *dynamics* to a piece. Other structural elements are determined by *tones*; the *pitch* (or frequency) of the notes

in the piece; the *key* of the piece (the tonic or harmonic centre of the piece, that forms the base of a scale) and the *mode*, which specifies the intervals between each note in the scale. *Melody* (the tune, sometimes known as the *air* in traditional music) is the fundamental theme of the piece. *Chords*—combinations of three or more notes played at the same time—and *contour*—changes or progressions from one note or chord to the next—also give structure to a piece. We only scratch the surface here.[1]

Blues music has a very common pattern of chords, expressed as the "twelve bar blues". (The Lieber and Stoller song "Kansas City" is an example of this pattern; so is their song "Hound Dog", best known as performed by Elvis Presley.) In a jam session, someone need only say "twelve bar blues, in (the key of) A", and everyone will have a common framework in which to play. Experienced musicians will need no other direction than that to begin playing music together. Improvisations and variations on the structure give richness and interest to the music.

Irish traditional music also has very common structures. Reels have four beats to the bar; jigs have six. (An easy way to keep them straight: the jig has the rhythm "jiggedy, jigeddy", and the reel doesn't; it has a rhythm that can be approximated by saying "wish I had a motorcycle".) Tunes are usually played as a pattern of "aabb"—eight bars with a given melodic line (a), a second eight bars that repeat that line or provide a slight variation (also (a); these two lines together are comprise the "A section"); and then eight bars with a different melodic line (b), and a second eight bars that recall that line and resolve toward the tonic or root of the chord (the latter two lines are the "B section"). In a "set" the players tend to play several tunes (typically three), playing each tune three times through and the switching to another tune, generally in the same rhythm, although the key may change (or "modulate") between tunes. Irish music has very strong and intricate melodies. Tunes tend to be in the keys of D, G, or A, since these are keys that are relatively easy to play on the fiddle, the flute, and the tin whistle, instruments that dominate the genre.

Remarkably, many capable musicians have a limited knowledge of musical theory. Nonetheless, these structures are sufficiently powerful that non-theorists can perform music just fine. That is, they appear to have an intuitive grasp of the structures. A regular participant in Irish traditional music sessions will know hundreds of tunes; a blues player will adapt easily to the many variations on the basic themes of the genre. Yet these people may have had little or no formal musical training, and may not be able to read music. So how do they remember pieces? According to Levitin, they rely on a structure for their memory, and the details fit into that structure. [Levitin2006, p. 213]

Testing also has structures, both in design and performance. We would argue that exploratory testing is structured in ways that are analogous to improvisational music. As a relatively new tradition, many of the structures haven't been named or even noticed, particularly for exploratory approaches. Practitioners—even experts—sometimes have a difficult time articulating what they do. Some experts can often play well only inside their own contexts. The cognitive patterns are mysterious and we're still learning how to understand them and the ways in which they interact with other contextual elements of testing.

Some of the musical genres and styles we have introduced can also be described as *patterns* and *practices* with related *techniques*. Musicians will often learn the patterns of what sounds right in a particular genre, without knowing the theory behind what they are performing. In testing, we have observed different kinds of patterns, practices and techniques as well, and many testers have a limited knowledge of testing or computer science theory. Both of us work as testing trainers, and have noticed that many testers will immediately try similar kinds of patterns or techniques when given a particular program to test as an exercise. For example, when posed with an application that has input fields, most testers will try overflow attacks, or will try to enter in values of the wrong type. Similarly, musicians adapt when playing music, depending on the genre. Again, we scratch the surface here – there are many patterns that are often used in testing and in music.

James Bach suggests that the structure of exploratory testing comes from many sources: test design heuristics; chartering; time boxing; perceived product risks; the nature of specific tests; the structure of the product being tested; the process of learning the product; development activities; constraints and resources afforded by the project; the skills, talents, and interests of the tester; the overall mission of testing; and the testing story. [BachRST] James and Jon Bach have proposed lists of exploratory skills, tactics and dynamics that refer to patterns of performance [JamesBach2005]. Bach, Mike Kelly, Jonathan Kohl, Scott Barber, Ben Simo, and other testers have suggested mnemonics to remember guidewords heuristics. (Kohl introduces the idea of comparing music and testing mnemonics in [Kohl2007])

In the most extreme form of improvisation, avant garde musicians abandon not only score but also structure. The result is occasionally interesting, but isn't usually popular to listeners other than devotees. It is often performed as an experimental exercise in the attempt to discover something new. Avant-garde music tends to help create new genres, and provide a space to help new ideas foster and grow. Without experimentation, discovery of new forms or new fusions of old forms can be suppressed. Testing is fundamentally experimental and investigative, and would profit greatly from avant-garde ideas both on testing projects and in the software testing community as a whole.

Since software testing is not generally performed for entertainment, or practice in the same way that performing music is, there are other areas where the analogy breaks down. Since testing is about discovering and reporting important information, combinations of practices and techniques that would be unlistenable in a musical setting are areas of discovery in testing. Since we are not usually performing for an audience who is expecting to see or hear

[1] An excellent description of these elements of musical structure can be found in Levitin, *This Is Your Brain On Music*. [Levitin2006]

something pleasing, we could have more room for experimentation in testing.

In music, composition and performances are often critiqued according to different standards and aesthetics. In testing, we often talk about test coverage, but we don't often evaluate how effective our testing is. There are potential lessons we could draw from musical critiques, and take form, structure, variation and diversity of approach into account as we evaluate our software testing efforts.

Tension and Resolution

In composition and performance, music often exhibits aspects of tension and resolution. A typical piece of music is written in a given key, and typically the piece ends on the tonic, the first note or root of a scale in that key, accompanied by the fundamental chord that shares the name of that key. Patterns of notes and chords in that scale create suspense that is resolved by a return to the root.

In a testing session, tension and resolution revolve around testing ideas, rather than musical notes. Tension and suspense are generated by a test idea, a question about the system under test. Resolution comes with an answer to that question, produced by operating and observing the product. We see another parallel between music and testing. Too much tension raises discomfort; too much resolution becomes boring, tedious repetition. Testers and musicians alike need to find a balance between tension and resolution, and to find this balance, they need a mix of knowledge, skill and creativity. [Kohl2007]

Feelings of tension and resolution in music are also felt and observed by the audience as they listen to a live or recorded performance. Music practitioners also learn from watching others at work. Performing music is related to the practice and skill development of a musician, and the listening enjoyment of the audience. In music, most of the information guides and is locked up in the performance. In music, impressions are generally about the qualities of the performance itself.

The elements and focus of the performance is one area where our analogy breaks down to some degree. In both testing and music, the audience derives an impression from the performance, and much of this impression is sustained after the fact. But software testers generally do not perform their work in concert halls, nor in front of audiences. Their work is conducted in relative isolation, with a different goal in mind: to gather as much important information for stakeholders as they can. Testing is not usually done for the benefit of an audience watching the tester doing his work. Instead, the value for the audience is in the information derived from the performance, rather than the performance itself.

There are some exceptions on the performance issue in software testing. Some exploratory testing teachers such as James Bach, Jon Bach, Michael Bolton, Jonathan Kohl and others do live testing demonstrations. With the rise in popularity of video on the web, many are recording test sessions for the benefit and enjoyment of others—typically other testers, or people who wish to learn about testing. The difference in these performances is that they are usually done for teaching purposes, not for the viewing or listening enjoyment of a broad audience. Like musicians, testers can learn from watching others perform. Furthermore, differences in styles and genres become much more apparent when demonstrated. We see this as an opportunity for testing education.

Scripting

In both music and testing, there is a dynamic between scripted processes (in which the ideas come from some person or agency at some point in the past) and improvisational or more exploratory processes (in which ideas are created and discovered on the fly, during performance).

For a given activity, we define scripted and exploratory approaches to be at opposite ends of a continuum. In a scripted approach, the process of design and execution of the activity are separated in time, and typically in the person performing them. Some person composes, designs or synthesizes ideas in advance of the activity, and commits them to some medium—typically in a textual or written form. The person performing the activity interprets the text and is guided by those ideas. We define the degree to which an activity is scripted as the extent to which the idea and the precise steps to exercise it are specified in advance; the extent to which those ideas guide the person performing the steps; and the degree to which learning associated with design is separated from learning associated with the activity. An exploratory approach is one in which design and execution happen simultaneously, not separated either by time or by person. Instead, composition and performance happen in a way that responds to context; to the skills of the performer; to what just happened; and to a consensus, often unspoken, on what should happen next. Learning about design and learning about the activity are not separated; they too happen simultaneously.

A purely scripted approach in music is a very strict interpretation of the piece as composed, typically by reading a score. At the opposite pole from playing a piece by reading a score is playing a piece "by ear". Music played by ear is played without sheet music or with minimal guidance from it. Instead, the musician learns the piece and its structure by listening to others play it. Playing by ear is sometimes but not always associated with improvisation, in which musicians compose and perform their ideas simultaneously. The players make choices about what to play based on the structure of the piece; skills in listening to and observing other performers; technical and physical skills; the emotions and mood of the players. Successful improvisation requires skill, and top performers study to develop a large breadth and depth of musical theory and technical proficiency on their instruments in order to successfully and creatively improvise. A purely exploratory approach in music performance is free-form improvisation. There are many variations in between. Few musicians can achieve a purely scripted interpretation. Conversely, very few (if any) musicians have the skills and ability to only play music that is influenced by the last note that was played in free-form improvisation.

Western classical music is highly scripted in the form of a *score*. A score, or sheet music, uses a highly specific notational system that allows performers to reproduce the basis of the piece with their voices or instruments. A score typically specifies the melody—the tonal and rhythmic patterns of the notes to be played. The score may also identify harmony—other notes or chords to be played at the same time as the melody, possibly identifying different notes or countermelodies for various instruments; the tempo—the speed at which the piece is played; volume; accents; and even bow strokes. Despite this rich, detailed, well-disseminated, and shared "language" for written music, it is difficult to perform music exactly the way the composer intended. Performance on non-electronic instruments will always include variations in intonation, timbre volume, dynamics, and embellishments. These variations might be subtly nuanced, or performed with a flourish; they might remain quite faithful to the original or common or they might be dramatic reinterpretations.

When performing a well-known piece of music, there are scripted and unscripted dynamics at play. Even when played from a score and when under the direction and supervision of the author of the music, subtle variations creep in. Reproducing the composer's ideas is particularly hard with musical pieces that have been around for centuries, because we don't have the composer around anymore to consult. Bodies of traditional interpretation tend to arise around pieces as they age. In popular or traditional music, people frequently play without any sheet music at all. This might suggest that there is a great deal of freedom in unscripted music, but this isn't always the case. In most styles of music—such as Irish traditional music—players adhere strongly to melody, even in the absence of a score. In Indian classical music, music is not recorded in scores; instead, the scripts are passed down through an oral tradition. Music students are taught by a teacher or guru in this manner: The teacher plays or sings a part, and has the student repeat it. While this teaching style differs from Western classical music where the music is written down, both are using scripted approaches: the ideas come from some time in the past, and from another person.

There are several factors that influence the decision to be faithful to the script. One factor in playing a piece from a score is the level of detail in the sheet music itself. A Western classical orchestral piece tends to be very highly scripted. Because many instruments and players must be coordinated to achieve a precisely desired effect, the individual lines of music may be laid out very specifically. Nonetheless, the performance is still strongly influenced by the individual musicians' playing styles and the interpretation of the orchestra's conductor. By contrast, in general, scoring for popular music—if used at all—tends to be less detailed. A typically arrangement of a song provides detailed music for piano, the melody line for a singer, and chords for guitar; other scores contain only the melody line and the chords.. The score itself affords the opportunity for a cognitively engaged player to bring some level of variation and interpretation to the performance. In fact, such scores mandate interpretation because they are sparsely detailed. At the other extreme, software can be programmed to play music such that it is very precise in repeating what is input from a score, but it tends to be boring and tedious, rarely as interesting and as pleasant to listen to as real performers are. In popular music, audiences and performers alike tend to allow a lot of room for improvisation and spontaneous discovery.

Scripted and exploratory approaches to testing are similarly on opposite ends of a continuum. In a scripted approach, the processes of test design and test execution are separated by time, and typically by person performing them. A test designer develops test ideas, and records them in advance; the person performing the test is guided by these ideas. The degree to which a test is scripted is the extent to which the test idea and the steps to exercise it are specified in advance.

An exploratory approach is one in which design and execution are not separated, either by time or by person. Instead, the tester performs each test in a way that can incorporate all of his or her knowledge of the program, right up to the result of the last test. Steps and test ideas are not specified in advance, and they may be recorded in great detail or not at all.

When a tester is working without a script, what can we expect to happen? If the test is memorized, or they have watched other testers perform the test, they may follow it as closely as they would if they had a recorded test script in front of them. If the test is not memorized, or has not been repeated so many times that it has become routine, we may see similar creative effects in testing as in improvised music.

In improvisational music, playing a euphonious note that fits with the ensemble and advances their discovery and engagement with the piece is important; in exploratory testing, performing some activity that fits with the project and advances discoveries and engagement with the product is important. In improvisational music, playing the right note is not so terribly important, but playing *a* right note is very important. If you wish to control the sound of the piece, emphasize scripting; if you wish to extend possible interpretations and knowledge, emphasize improvisation and exploration. In exploratory testing, our work is not as visible in the way music performance is, and we certainly can't hear what our tests are doing (unless we are testing music software.) Therefore, we have far fewer constraints when we improvise than our musical counterparts. We have less of a framework to work from, but more possibilities for discovery.

Automating tests is the strongest guarantee that they will be repeated exactly the same way, but like automating music, the lack of interpretation in execution can limit the results. A computer can only find the problems we predict and program it to find. Repeating scripted tests over and over can get boring, tedious, and may only feel like idea resolution, without the vital tension created by curiosity [Kohl2007]. At the other end of the spectrum, there is testing that is improvisational: exploratory testing. In the musical realm, electronic, or computer-assisted musical devices are fused with human efforts. This allows the musician to explore and create music that they would not be able to do completely on their own without the aid of

tools. Similarly, in testing, we can use automation tools to help us work more creatively, and perform tasks that would be impossible without a machine helping us. [KohlM&M2007]

Skill and Skills Development

Both music and testing can be done easily by people without skill, but the perceived value of each is greatly enhanced by skill. Skill itself is enhanced by practice, the engagement of the performer, performance, knowledge of structures, and mnemonics that foster rapid learning.

In *This Is Your Brain on Music*, Daniel Levitin recounts considerable research into skills development in music. Levitin points out that skill and success in the music business are not strongly related; there are too many vagaries of timing, luck, and the whims of popular culture, and he acknowledges that expertise is a social judgment. For this reason, research involving assessments of musical skill has tended to focus more on technical achievement and innovation, and less on aesthetic appeal or popular acceptance.

Formal training (or its absence) is not necessarily associated with perception of musical skill. Many popular, skilled, and respected musicians, whether in popular music (Frank Sinatra, Louis Armstrong, John Coltrane, Eric Clapton, Stevie Wonder, Joni Mitchell, Irving Berlin), traditional music (Tommy Potts, Frankie Kennedy), or classical music (Gershwin, Mussorgsky and Beethoven) received little or no formal instruction.

Two key factors that do make a difference, according to Levitin, are emotional engagement and practice. The best students of music (and of other disciplines, according to related research are those that have practiced the most. Ten thousand hours of practice is required to develop world-class expertise[2]. Expertise in music, especially in music that is not heavily scripted, is associated with memory; and strength of a memory is related to the number of times that the original stimulus has been experienced.

The strength of the memory, and the associated development of expertise, is a function of emotional engagement—how much the user cares about the experience. To perform well, says Levitin, we have to pay attention and we have to care. More caring leads to more attention, and both caring and attention lead to neurological changes that mark experiences and memories as important.

Those who have not studied music may be surprised to find the great scientific, mathematical, philosophical and artistic energy that has been put into music over the years. In *Temperament: How Music Became a Battleground for the Great Minds of Western Civilization*, Stuart Isacoff mentions some of the people who were involved in solving problems in music: Pythagoras, Galileo, Kepler, Descartes, Newton, Huygens, da Vinci, Rousseau and others. [Isacoff2001] Music has a surprising depth into many areas of thought and study. Similarly, software testing is influenced by many disciplines, and has a surprising depth in many fields because of the vast number of technologies in use around the world. Music is not limited to learning the mechanics and rules to create and perform music, but is full of scientific, mathematical, social, political and artistic problems. Software testing is not limited to the execution of tests, and is also full of similar issues as music. We haven't learned enough about them all yet, and don't have the benefit of the many years and research that have been poured into music. We still have much to discover and learn about both.

Since the research that Levitin details on learning and music is consistent with learning in other disciplines, there are likely to be parallels that play out in testing. We propose the following hypotheses:

- We suggest that what we know about learning argues strongly for giving testers stimulating work that engages them, and argues against putting testers into situations where they simply repeat activities with which they are not engaged.
- Like music, developing testing skills requires development and practice. The software testing community could learn from musical counterparts as we develop exercises and practice software testing.
- Testing has very little of the aspects of physical performance, found in musical performance, that can obtain some benefit from rote repetition; there are few "muscle memory" skills in testing, but there are cognitive skills. Testing work that is boring or uncreative is less likely to be memorable, and thus less likely to lead to learning.
- Testing training that involves memorization of testing terms for the purpose of passing a certification test is unlikely to contribute much to the quality of software testing. In music, written theory exams don't begin until Grade 5; all testing and certification up to that point is based on performance. We hypothesize that the emphasis on technical terms found in current testing certification schemes adds little or nothing to the development of skill, just as the learning of musical terms contributes little to the quality of performance. As with musical performance, testing training that involves experiential learning, on-the-job training, coaching, and mentoring, will result in the development of skills.
- Schools of thought in testing ought to be encouraged, with more research into the differences and diversity of styles, genres and subgenres publicized for the learning profit of the software testing community as a whole. Avant garde, or cutting edge, experimental testing ideas and techniques should be encouraged, not written off. The resulting examples, cross-pollination and feedback loops would add more diversity to software testing.
- The manufacturing metaphor in software development is old, tired and often inappropriate. Even new variations like "Lean Manufacturing" do little to add to a software development field that is heavily based

[2] Levitin refers to Anders Ericsson, FSU.

on design. Other creative, design-heavy fields should be explored, even artistic ones such as music.

Clearly more research is called for. The software testing discipline, like music, can be subtly complex and surprising. Merely taking an engineering or manufacturing view and trying to automate away the human labour-intensive side of music hasn't worked in music, and doesn't look like it's working in software testing either. There are more disciplines to learn from than engineering and manufacturing, and the musical field is full of ideas we can explore as we learn more about software testing.

Notes

[BachRST] Bach, James, and Bolton, Michael, *Rapid Software Testing*. http://www.satisfice.com/rst.pdf

[Isacoff2001] Isacoff, Stuart. *Temperament: How Music Became a Battleground for the Great Minds of Western Civilization*. Random House, New York.

[Kohl2007] Kohl, Jonathan. 2007. *Exploratory Testing: Finding the Music of Software Investigation*. Self-published 2007 and Methods & Tools Magazine 2007

http://www.kohl.ca/articles/ExploratoryTesting_MusicofInvestigation.pdf

http://www.methodsandtools.com/archive/archive.php?id=65

[KohlM&M2007] Kohl, Jonathan. 2007. *Man and Machine: Combining the Power of the Human Mind With Automation Tools*. Better Software Magazine

http://www.kohl.ca/articles/ManandMachine_BetterSoftware_Dec2007.pdf

[Levitin2006] Levitin, Daniel J. 2006 *This Is Your Brain on Music: The Science of a Human Obsession*. Penguin Group, New York

[Pettichord2003] Pettichord, Bret.(2003-2007.) *Four Schools of Software Testing*. http://www.io.com/~wazmo/papers/four_schools.pdf

Session

The Challenge of Testing Scientific Software

Diane Kelly and Rebecca Sanders

Tuesday, July 15, 2008
10:45am - 11:45am

Room: Saint Patrick 3rd Floor

Overview

Though scientific software is an engine for scientific progress and provides data for critical decisions, the testing of scientific software is often anything but scientific. Our research examines the factors that complicate the testing of scientific software, including the complexity of the subject matter, inadequate validation criteria, a high demand for correctness, and the lack of testing expertise among scientists.

The Challenge of Testing Scientific Software

Rebecca Sanders
Queen's University
Kingston, Ontario
Canada
sanders@cs.queensu.ca

Diane Kelly
Royal Military College of Canada
Kingston, Ontario
Canada
kelly-d@rmc.ca

ABSTRACT

Scientific software, application software with a large computational component, is often used to provide data for critical decisions across broad areas of research in nearly every science and engineering discipline. Testing is key to assess the quality of this software. Yet there are several challenges in doing so that are not always adequately addressed. The definitions of verification and validation in the context of scientific software cannot even be agreed upon in the literature.

We carried out a series of interviews of scientists who write or use scientific software and found that there are three broad areas of risk in scientific software, related to theory, to code implementation, and to its usage. All the scientists we interviewed addressed the first risk by validation testing, generally did not test to address the second risk, and only tested to address the third risk if the users were not scientists in the same domain.

1. INTRODUCTION

Scientific software is software with a large computational component that models some physical phenomena and provides data for decision support [1]. Development of scientific software takes place in both industry and academia. The people who develop and test the software are often scientists and engineers, many of whom fit under Segal's definition of "professional end-user developers" [2]. Scientists and engineers developing software tend to have strong backgrounds in the theoretical models implemented in the software, but they do not usually have a strong background in computer science or software engineering. In other cases, scientific software is developed by people who are not domain experts (such as software engineers) in close consultation with domain expert scientists or engineers.

We conducted a qualitative study, the focus of which was the development and usage of scientific software. Carried out at two Canadian universities, this study was broad and hence not solely focused on testing issues. But many of the concerns we heard affect how scientific software is tested.

We found that there are three broad areas of risk in play. These are theory risk, which relates to the complexity of the theory and the difficulty of validating it; implementation risk, involving the fidelity with which the problem was coded and documented; and usage risk, concerning the use of the software by the target user groups. These risks may not be unique, but they are all in some way exacerbated due to the nature of software development in the science community, as well as the complex nature of the science that lies behind the software.

The ways in which these risks overlap and interact pose challenges for testing. Some of these challenges can be met by existing methods, while others may require an injection of ingenuity.

Section 2 gives a short description of our research method. Section 3 describes the contributions of risks related to scientific theory to the difficulty of testing scientific software. Section 4 details several risks related to the code, section 5 discusses usage risks, and section 6 concludes the paper.

2. RESEARCH METHOD

Qualitative data was collected through interviews with scientific software developers and users. These interviews were thirty to ninety minutes in length.

We interviewed sixteen scientists and engineers. Two are civil engineers, one is a chemist, two are electrical engineers, one is a geographer, one is a computer scientist involved in medical computing, five are nuclear engineers, three are in physics, and one is a theoretical computer scientist. There is a large degree of variation in their software development experience; some are not comfortable coding, while some others have worked on industrial software engineering projects and are very knowledgeable about requirements, design, testing, and version control options available to them. Their software varied greatly in size from modules of less than 1,000 lines of code to programs of over 100,000 lines of code. We interviewed scientific software developers who had delivered software, in their opinions successfully, as well as interviewing developers who were encountering severe problems in their development efforts and were asking for help. These problems include users finding it difficult or impossible to install the scientific software, poor code documentation that makes it hard to understand what the software is supposed to do, and an inability to find important bugs with current testing techniques.

At the beginning of each interview, each scientist gave a brief description of his or her work, the purpose of the software as related to his work, and his or her role in the development group. Then we asked questions. Though there were no pre-set questions, we always made sure we covered the purpose of the software, requirements documentation, design, development languages, code documentation, version control, and testing when we discussed development of scientific software.

We made a digital voice recording of each interview. After each interview, the digital voice recording was translated into written notes. As interviews continued to be conducted and notes were taken from the digital voice recordings, we began the process of focusing and bounding our collection of qualitative data. This was done through a well-defined qualitative analysis technique called coding [3]. Codes are a summarizing notation; they make analysis

more efficient and effective by both grouping notes on similar subjects so they can be compared during the later phase of pattern-finding and demonstrating where notes are incomplete. For example, notes related to testing were coded as TEST. The interviewees names were removed from the notes; in this paper, they will be referred to by number (S1, S2, S3, etc.) See Table 1 for a list of the research area for each interviewee.

Table 1: Interviewee Characterization

Subject #	Field	Formal training in soft. eng.?	Experience in the software industry?
S1	Nuclear Engineering	No	No
S2	Nuclear Engineering	No	No
S3	Optics/Image Processing	Yes	Yes
S4	Geography	No	No
S5	Chemistry	No	No
S6	Physics	No	No
S7	Theoretical Computer Science	Yes	No
S8	Electrical Engineering	Yes	No
S9	Nuclear Engineering	No	No
S10	Nuclear Engineering	No	No
S11	Nuclear Engineering	No	No
S12	Medical Computing	Yes	Yes
S13	Civil Engineering	No	No
S14	Electrical Engineering	No	No
S15	Civil Engineering	No	No
S16	Physics	No	Yes

Nine of the thirteen developers we interviewed (the other three interviewees were primarily interviewed on their use of scientific software and their validation testing) developed software to demonstrate that their theory works, to test their models, or to provide evidence for research publications. One interviewee developed software for training highly qualified personnel, and seven developed software for external decision support. These purposes are not mutually exclusive.

We used cross-case displays [3], a matrix with columns comparing the content of a code between different interviewees, as our primary systematic analysis method to compare and contrast data gathered from all, or a subset of, the interviewees. These displays allowed us to find patterns and themes among the data.

Most testing of scientific software among our interviewees can be described as unsystematic or ad-hoc. One interviewee said he had tested as much as he could, and he knew there were bugs, but he didn't know how to go about finding them. Another interviewee characterized the testing in his group as ad-hoc and disorganized. His group is continuously in what he terms a "run and fix" mode of operation. However, given information they provided on specific types of testing done in their group, their testing is probably more thorough than most.

After coming to several conclusions through systematic analysis, we decided to take our results back to the community of developers we had interviewed to gather their impressions. This also served as a check on our conclusions to ensure that they made sense to the practitioners themselves. Four of the interviewees attended our focus group: S3, S4, S9, and S14. The additional comments were added to our analysis.

3. THEORY RISKS

The primary goal of developing scientific software is very often to get the results of calculations from a scientific or engineering model as opposed to developing quality software for its own sake [2][4][5]. These results may be used for decision support or to validate a scientific model. Therefore the theory often serves as the functional requirements of the scientific software. In this context, theory risk is similar to requirements risk.

3.1 Risk of Cognitive Complexity

Cognitive complexity is the difficulty in understanding a concept, thought, or system [5]. Scientific software is being created to model increasingly complex phenomena. One interviewee in chemistry who has been developing scientific software for decades discussed how the complexity of what could be represented by software used to be limited by hardware. Before, it was challenging to model liquid water – but now, with processing power far greater than in the past, scientists in his domain are able to model protein folding.

As scientific software expands to encompass more complex science, the amount one needs to understand about the domain in order to understand the software is increasing. Some of the models being translated into software are cutting edge research. Developers may implement their science models in software to demonstrate that they work as part of their own research. Sometimes the software is intended to take advantage of a new market opportunity.

Whether the intent of the software is to demonstrate a new model, or to fill a new use in the market place, the cognitive complexity is high. This presents a problem for testing because most testing requires the tester to understand what the software is supposed to do. For example, most software testers do not have PhDs in chemistry and don't understand protein folding. The lack of scientific knowledge amongst software testers, or the lack of even a common cognitive ground with the scientists, had proven frustrating to several of our interviewees when they attempted to collaborate with them. The frustration was enough that most scientists stated that they wouldn't bother with such collaboration. S07 hired software developers for developing and testing of the user interface, but found that they could not do much to develop or test the scientific computational part of her scientific software due to their lack of knowledge.

There were some examples of successful collaboration between scientists and software developers among our interviewees, but these required the software developers to spend hundreds of hours learning the necessary background knowledge from the scientist. S14 and S15 worked together to develop civil engineering

software for the military to calculate the load on bridges, with S14 serving as the software expert. S14 did not have a formal background in software development but was self-taught in design and development. They spent many hours developing a shared vocabulary, with S15 teaching S14 so much about bridges that S15 stated "[S14] knows more about bridges now than most civil engineers". With this knowledge, S14 developed and tested the software.

S12 is a trained software developer who created medical imaging software for use by doctors in the operating room. In order to gain the understanding necessary to create software for use in the medical field, he spent hundreds of hours in the operating room watching doctors at work.

The lack of scientific knowledge among software developers is one reason why most of the scientific software development efforts we encountered were entirely composed of domain experts doing their own development and testing. But the scientists and engineers often have no background or education in testing. Unlike testers, they understand their domains, but they usually don't know how to effectively test their software. Those who did the best testing among our interviewees were, understandably, those who had made the most effort to gather software testing knowledge.

3.2 Risks to Validation Testing

Kendall et al [4] define validation as "determining whether the mathematical model instantiated in the code faithfully mimics the intended physical behavior". Post and Votta's definition of validation is "the determination that the model itself captures the essential physical phenomena with adequate fidelity", yet they go on to include several methods of validation testing that would fit the software engineering definition of verification testing [6]. Roache uses a mutation of the standard software definition – he says that validation is ensuring that software is "solving the right equations" [7]. Stevenson states that "Validation answers the question 'How well does the model reflect objective observations?'" [5].

Often validation testing (using Post and Votta's definition) is used by the scientists to the exclusion of all other forms of testing. As with other quality management methods, the effectiveness in implementation is often compromised [4][5][8].

Fifteen of our sixteen interviewees discussed their validation testing practices. From our notes, we identified three types of oracles used to validate scientific software. These oracles are not mutually exclusive; several interviewees used oracles from more than one of these sources.

3.2.1 Professional Judgment

Most common is an oracle based on *professional judgment*. This relies on an expert or group of experts in the domain to evaluate the results based on their experience and knowledge. One way of validating software in S3's domain of astronomy is what he calls "eyeball analysis" – a visual comparison of the image processed with their algorithm versus others. As S3 characterized it, this is as simple as stating, "look at image A, look at image B; image B is better." S11's model is related to nuclear meltdowns. This falls in the category of an undesirable event for which there is little real data. Due to his lack of validation data, his only oracle is his own expert judgment.

3.2.2 Data-based Oracles

Data-based oracles are those where data can be generated in some way as comparisons for the software being tested. This can include data from measuring real-world events, data from instrumented physical experiments, and data from hand calculations. S1 and S2's software was validated by using data gathered from several sources in the nuclear industry. S6 initially tests by using professional judgment. This serves as a preliminary test before allocating computing resources to input real weather data and check the forecast against the weather that actually occurs. S9 compares the output of his program to data on standard nuclear element performance. If his model produces a result within an error range, which could be "large" by his description, then it is considered correct. Some interviewees use hand calculations to check the results of their software; these include S13, S14 and S15 – the two examples of civil engineering software development among our interviewees.

3.2.3 Benchmarks

Benchmarks are measures by which the output of the model implemented in the scientific software is compared relative to the outputs of other models. S3's group uses an industry standard sharpness metric as one way to judge their software's output. S5 stated that in his domain there are applications modeling behavior that one does not ever want to encounter in real life, so there are benchmarks that can be used to determine whether the results are better or worse than those produced by other methods. S10's model was meant to be a simplification of another model, so if his model produces results that are acceptably close to those of the more complex model, he considers them correct.

3.2.4 Oracle Risk

All of these types of oracles carry the risk that they could be incorrect. The data may have been collected or recorded improperly, or a hand calculation may be incorrect due to human error. Even with industry data, S2 stated that his group has had trouble validating their models. He described a give and take process between his development group and the industry data sources: if the answer given by the output of the scientific software does not match the expected output from the industry data being used as the oracle, it is possible that the industry data is at fault instead of the scientific software. S2 has encountered instances in which the industry data was incorrect as well as times when his model required adjustment. Since the industry data is not entirely trustworthy, some degree of professional judgment must be used to determine whether the oracle data itself is incorrect.

With benchmarks, a risk is that the benchmark does not provide a consistent comparison. There may be some way to improve the performance of a model in relation to the benchmark without improving the model. S3 expressed his dissatisfaction with the industry standard sharpness metric used in his field, stating that the results can vary by 80% or more depending on the sampling of the image used by the benchmark. As with data-based oracles, there remains a necessary degree of expert judgment with regards to how much to rely upon the results of a benchmark.

There is also an inherent risk when the scientist is not objective when validating his own scientific software. Given that one of the most important goals of some scientific software developers is to publish papers based on the results of their models, there can be a

psychological barrier to finding problems with their software. And as argued in [6], the peer review process for papers is largely ineffective at sniffing out poor validation testing.

In addition to the challenge posed by obtaining a reliable oracle, it is challenging to determine what constitutes sufficient validation. Sometimes the oracle used is too simplistic or not in the range in which the scientific software will actually be used. S4, who created her own oracle data, describes her dataset as "simple" and not representative of the data that it will eventually be used to process. Whether this simple simulated data validates the software adequately enough to justifiably increase her confidence in the model's applicability on complex inputs is questionable.

3.3 Risk from Approximations to Continuous Models

The difficulty of deciding whether validation testing is adequate or not is compounded by the approximations needed to render the continuous models into computational models. It can be very hard, if not impossible, to determine where boundaries or singularities lie. Having two tests yield acceptable answers does not necessarily guarantee that points between or close to the test data will also yield acceptable answers. Therefore validating the computational model even within the accepted range of applicability can be a highly challenging task. As with other domains, validation of scientific software can only influence confidence that the software will provide a correct answer, but its limitations are even more pronounced when faced with the mathematics involved in scientific software computations.

4. CODE RISKS

4.1 Risk to Correctness

Correctness, or the accuracy of a calculation, is a critical quality factor for most scientific software applications. Scientific software often requires a high degree of correctness to fulfill its requirements. The importance of correctness to scientific software is underlined by Hatton and Roberts [8] and Hatton's follow-up [9]. In Hatton and Roberts' research, over a dozen programs for seismic data processing in the oil industry, all implementing a similar algorithm, were found to deliver drastically different results, with answers becoming more deviant as the amount of computation in a process increased. The accuracy errors in these programs reduced the accuracy of the output from six significant figures to two. For the interpretation of the data to be useful, the data needed an accuracy of three significant figures. The programs were unfit to address the tasks they were intended for.

Hatton and Roberts believed that poor testing of numerical computations was likely to contribute to accuracy problems in software in other scientific domains as well. Correctness was a primary quality goal for many of our interviewees. The purpose of scientific software is to produce an answer that is as accurate a model of reality as possible.

Testing is critical to the realization of this goal. Stevenson's article on quality in simulations [5] is supportive of measuring the correctness of calculations. He proposes that testing methods for scientific software should focus more on numerical analysis, numerical methods, and floating-point computations. Feedback from our interviewees was in agreement with this.

Another way that the correctness risk is mitigated before testing is through a conservative choice of development language. Decyk et al's preference for Fortran [10], which has mature compilers and math libraries, was also seen amongst our interviewees; Fortran was the most common language used to develop their software. S1 and S2, who went against this trend and developed their scientific software in Visual C++, were plagued by different numerical results on different platforms.

4.2 Risk from Poor Code Documentation

Poor code documentation is a very common vice in scientific software [2]. Some interviewees agreed that documentation of code was so bad that they wouldn't expect themselves or their students to be able to understand someone else's code. S4 encourages students to write code from scratch instead of using other people's code because she thinks it's easier than trying to determine how someone else's code works. S16 stated that, due to a combination of laziness and protectiveness of their code, code authors in his development group created cryptic, inadequate code documentation that made the code nearly impossible to interpret. He compared such code documentation to a Russian textbook – concise and hard to understand unless one already knows the material.

Given that the material represented by scientific software already exhibits high cognitive complexity, the lack of proper code documentation poses an even higher risk to testing in this domain. Legacy code has a lifespan of decades [12]; it is common for scientists to add modules or extensions to existing code bases. Yet it can be very difficult to understand what the legacy code is supposed to do, which impedes testing. This is made even worse by the unavoidably high turnover in research development groups. Once the code author is long gone, figuring out what old code is supposed to do can take heroic effort.

4.3 Risk to Verification Testing

As with validation, there are several definitions of verification testing in the context of scientific software. Kendall et al [4] define verification as "ensuring that the code solves the equations of the models correctly". Post and Votta describe verification as "the determination that the code solves the chosen model correctly" [6]. These two definitions are similar to Roache's statement that verification ensures that the software "solves the equations right" [7]. Stevenson's definition of verification as answering the question "Does the algorithm and code work as required?" is broader [5]. The fact that the first three definitions ignore most aspects of the quality of the code aside from its conformance to the model shows how little attention is given to quality factors other than correctness.

The main risk to verification is that scientists and engineers developing software are often unaware of the need for it and unsure of how to apply it. There is a tendency for scientists to focus on validation testing to the exclusion of all other testing. Validation testing deals with their models, which they understand and which they are most interested in testing. Our interviewees consistently validated their software in some way, but the use of other types of testing or testing for other goals was patchy. This is partially due to a lack of knowledge of software testing, but it is also a problem with what Kendall et al refer to as a "quality attitude" – scientific software developers are often unconcerned with "customary notions of IT software quality" [4].

5. USAGE RISK

Surprisingly, if our interviewees did any other type of testing besides validation testing, it was usability testing. Some of our interviewees were doing batch processing that required no user interface. Others were developing software with complex user interfaces.

In these cases, usage of their software was viewed as high risk. Usage risk was addressed through both usability testing and user documentation. Some leaned more strongly toward either testing or documentation while others used an even mix of both. Many other interviewees did not address usability risks at all. The main factor that correlated strongly with a concern about usability risk was the degree to which the backgrounds of the software developers differed from the software's intended users.

In most cases, the developers and the intended users of scientific software have very similar domain backgrounds. There are cases in which the intended users are the developers themselves, people in the same research group, or people in the same domain specialty. Of those interviewees with user characteristics fitting any of those cases, none created user documentation or did usability testing. Scientific software developers view usability as a risk only when the users' background knowledge is markedly unlike their own.

Our interviewees used several approaches to address usability. S7's students were one of the intended user groups, so her usability testing consisted of having students in her course try her software and give feedback. S14 and S15 initially had graduate students test their bridge classification software and its documentation by experimenting with the program and consulting the help files, but these students were not representative of the eventual user group. S14 and S15 later had the opportunity to have engineers typical of the target users test the software and help files. S13 took a different path by focusing completely on user documentation, such as help files, instead of usability testing, and felt that this was a successful approach.

In one case (S12's software) the users were operating room doctors. S12 first had doctors test his imaging software on plastic models. Later, S12 made multiple observations of the software in use in the operating room. He described several surprises he had in the software's use. The in-use observation of his software was critical to improving its quality.

In another case (S1 and S2's software) the intended users of the scientific software are not scientists. In this instance, the software was being developed for academic users and non-specialist industrial users. From our interviews, developing a user interface and documentation for such diverse groups was a challenge that they were having difficulty meeting. They had no direct contact with their industrial users and activities such as installation testing was faltering. Their approach contrasts with S12's early engagement with real users from the requirements stage onward.

6. CONCLUSION

Risks to scientific software come from theory, implementation, and usage. From what we have observed, the scientific software developer, being most often the scientist with no supporting software expertise, tests almost entirely to resolve the theory risk. The scientist/developer is in a position in which developing software is necessary to accomplish scientific goals. They are not software engineers, and they do not want to be. Their overriding mindset is to demonstrate that the theory embedded in the code is correct. They do validation testing to demonstrate their theory. There are many risks in accomplishing this. The scientists have developed different strategies to address this risk.

If the user of the scientific software is a scientist in the same domain, as often it is, the risk due to usage is lowered. Some of our interviewees recognized the risk due to users outside their domain and addressed this through usability testing, being opportunistic and inventive in their choices of methods.

The scientists' lack of resolution of the risk due to code implementation comes from a number of factors. First, they see the code and the theory inextricably entwined. They cannot see the code as a separate entity that needs attention. The code is not tested for its own sake. Certainly, the code is not usually tested to show where it is wrong. Second, the scientists' interest and deliverable is science, not software. The code is a means to an end, and often a frustrating one. Third, scientists generally do not possess software testing knowledge.

The questions of what constitutes an adequate testing oracle for scientific software and what constitutes adequate test data are left open. Some of the validation testing we researched was running up against the practical limits; they were using whatever oracle they could scrape together from industry data, benchmarks, and expert judgment.

Software engineers often lack knowledge of and interest in scientific domains. There is a general lack of knowledge transfer between these two broad sets of disciplines. Lastly, effective and efficient testing methods and techniques specifically developed for scientists have not been put into the scientists' hands. The interlocking risks influencing the testing of scientific software means that testing strategies cannot be directly imported from other domains. The building blocks of effective testing strategies likely exist, but how to put them together in a way that meets the goals of the scientists poses a unique challenge. There is some important and critical work that could be done here.

REFERENCES

[1] Kreyman, K., Parnas, D.L., and Qiao, S. 1999. "Inspection Procedures for Çritical Programs that Model Physical Phenomena". CRL Report No. 368. McMaster University.

[2] Segal, J. 2004. Professional end user developers and software development knowledge. Technical Report. Open University.

[3] Miles, M.B. and Huberman, A.M. 1994. Qualitative Data Analysis: An Expanded Sourcebook, 2nd Ed. Sage Publications, California.

[4] Kendall, R.P., Post, D.E., Carver, J.C., Henderson, D.B., and Fisher, D.A. 2007. A Proposed Taxonomy for Software Development Risks for High-Performance Computing (HPC) Scientific/Engineering Applications. Technical Notes. Carnegie Mellon University.

[5] Stevenson, D.E. 1999. A Critical Look at Quality in Large-Scale Simulations. Computing in Science and Engineering, (May-Jun. 1999), 53-63.

[6] Post, D.E. and Votta, L.G. 2005. Computational Science Demands a New Paradigm. Physics Today (Jan. 2005), 35-41.

[7] Roache, P.J. 1998. Verification and Validation in Computational Science and Engineering. Hermosa Publishers, New Mexico, USA.

[8] Hatton, L. and Roberts, A. 1994. How Accurate is Scientific Software? IEEE Transactions on Software Engineering 20 (Oct. 1994), 785-797.

[9] Hatton, L. 2007. The Chimera of Software Quality. The Profession (Aug. 2007) 102-104.

[10] Decyk, V.K., Norton, C.D., and Gardner, H.J. 2007. Why Fortran? Computing in Science and Engineering 9, 4 (Jul.-Aug. 2007), 68-71.

[11] Boisvert, R.F., Tang, P.T.P., ed. 2001. The Architecture of Scientific Software. Kluwer Academic Publishers

Association for Software Testing

CAST 2008: Beyond the Boundaries

July 14-16, 2008 – Toronto, Canada

http://www.cast2008.org

Session

Lessons for Testing from Financial Accounting:
Consistency in a Self-Regulated Profession

Doug Hoffman

Tuesday, July 15, 2008
2:45pm - 3:45pm

Room: Colony Ballroom 2nd Floor

Overview

Comparing software testing with accounting is a study in stark contrasts. There are valuable lessons for software testing from several millennia of evolution and refinements in accounting, ranging from underlying accounting principles to professional self-regulation techniques. This session draws parallels from accounting and points out some valuable lessons that might be applied in the software testing profession.

Lessons for Testing From Financial Accounting: Consistency in a Self-Regulated Profession

Douglas Hoffman
doug.hoffman@acm.org

"Accounting is the discipline of measuring, communicating and interpreting financial activity."[1]

"The purpose of accounting is to provide the information that is needed for sound economic decision making. The main purpose of financial accounting is to prepare financial reports that provide information about a firm's performance to external parties such as investors, creditors, and tax authorities. Managerial accounting contrasts with financial accounting in that managerial accounting is for internal decision making and does not have to follow any rules issued by standard-setting bodies. Financial accounting, on the other hand, is performed according to Generally Accepted Accounting Principles (GAAP) guidelines."[2]

"Software testing is the process of measuring, interpreting, and communicating important qualitative aspects of computer programs. The purpose of software testing is to provide the information about software quality that is needed for to improve quality and make sound business decisions." – Douglas Hoffman

Summary

To say that accounting concepts and methods have been around for a long time is a gross understatement. Accounting records date back over 7,000 years, recording business transactions and inventory. Over the centuries accounting has evolved (and continues to evolve) to become an effective set of practices based on principles and standards. Financial accounting is that part of accounting focused on consistent and complete reporting of financial information for outside stakeholders. Managerial accounting focuses on the reporting of financial information as input for making management decisions. At first look, the accounting field appears to be defined by strict rules using arcane language. Financial accounting standards are applied across industries to make financial statements consistent and thereby more understandable and comparable. A closer look at accounting, and particularly financial accounting, reveals that the standards are adjusted to accommodate real-world constraints and practical methods – context-driven standards.

By contrast, the field of software testing is young, having been around for about 50 years. Before then, programmers simply did whatever testing they felt appropriate during software development. As the software development field grew, roles and responsibilities became more clearly differentiated, During the 1960s the software creation process evolved to become logically separated from software validation. Many people have tried to describe a universal software testing methodology[3], but the usefulness and applicability of these different approaches are hotly debated today.

Stakeholders cannot generally compare (or understand) test reports without well defined, stable standards. Software testing methods used within industries vary substantially; terminology is inconsistent throughout industries, test reports include many different measurements, use various formats, and have different contents. On many projects the format and content of reports evolve within the course of a single release. Factors labeled with the same terms are often counted differently or embody altogether different concepts. The methods and vocabularies between industries are so different that software testers from different industry segments often cannot even discuss testing. (This sometimes happens between projects within a single organization.) The software testing field lacks a common vocabulary and shared definitions of terms. It lacks a set of assumptions, principles, or methods that apply to software testing across diverse parts of the software development industry.

As different and the fields of Accounting and Software Testing may seem, there are valuable lessons to be learned from accountancy and accounting [4] that are applicable to software testing. Where accounting's unit of measure is dollars, software testing has many measures for defects, tests, test outcomes, and other qualitative factors. There is currently no single element in software testing that could represent a common characteristic for everything we might measure, so it isn't possible to summarize software quality

[1] Wikipedia, 13 March 2008; http://en.wikipedia.org/wiki/Accountancy

[2] QuickMBA/Accounting/Financial Accounting; Copyright © 1999-2007 QuickMBA.com http://www.quickmba.com/accounting/fin/

[3] Some examples of prescribed test methodologies are: IEEE Standard 829-1998; Craig and Jaskiel, "Systematic Software Testing;" and Buwalda, Janssen, and Pinkster, "Integrated Test Design and Automation."

[4] *Accountancy* is the profession of *accounting*, which is the methodology. Wikipedia, Op. cit.

in a single equation as accounting does[5]. But, both accounting and software testing must present information in an understandable form for it to be useful. Software testing would benefit from using a common language and common methods to consistently identify and count items. Consistency is needed within organizations and across industries so stakeholders can understand and compare qualitative information. This paper re-casts several processes, rules, and measures borrowed from financial accounting to apply them in the context of software testing. The lessons help us understand the value of processes, keeping measurements simple, test strategies, documentation, and more.

Financial Accounting and Software Testing

In much the same way that software quality assurance is much broader than software testing, accounting is much broader in scope than the financial and managerial segments described here. Most implications drawn from accounting for software testing are related to measures, metrics, and processes because financial accounting is concerned about one metric: dollars. This paper focuses on some of the aspects of financial accounting that provide lessons that can be applied to software testing. The specific lessons drawn upon are outlined in Table 1.

◊[6] The ***foundation for accounting*** is based on three types of tenets: assumptions, principles, and guidelines. ***Assumptions*** define and limit the scope of accounting. ***Principles*** lay out the basic axioms for accounting. ***Guidelines*** describe exceptional circumstances and reasoning where assumptions or principles may not be strictly applied. Some of the tenets have parallels in software testing. In Appendix 1 I provide a detailed list of the basic tenets of accounting.

Software Testing Principles can be described using a similar three-tiered approach. The rules and responsibilities for software testing could define rules such as a definition for what constitutes a defect, possible test outcomes, and test report requirements. Where the ***Assumptions*** describe the boundaries of software testing, Assumptions for software testing might include ideas like: 'the purpose of testing is to provide information of interest to some stakeholders,' 'software testing is always incomplete since there is always a potential for undiscovered errors,' 'software tests should be designed and created to provide the most valuable information for important stakeholders,' and 'test measures and metrics represent information from specified products/projects during specific time periods.' These assumptions describe the common scope for software testing and should apply across test groups and industries.

The ***Principles*** describe what needs to be done, how things are defined, and fundamental approaches to be taken in software testing. Software testing principles might include things like: a software defect is defined as a discrepancy between expected (specified) behavior and actual behavior, and defect counts should not include cosmetic errors that do not lead to inappropriate user actions.

The ***Guidelines*** provide a set of rules that allow flexibility to accommodate industry differences and operational reality. Possible guidelines might include: when the verdict from a test should be recorded as a failure until it is known to have passed, and accepted industry practices should be followed even if they differ from the assumptions and principles.

◊ The ***Monetary Unit Assumption*** says that all values are translated into Dollars (or Kroners, Rupees, Yuan, etc.). Accounting concerns itself with items that can be valued or measured in money. Things like customer loyalty (goodwill) may be quantified, but only the financial value which is quantified in dollars will be included in financial records.

For software testing, this could translate to using one unit of measure for all values. (We may use several different factors, such as defect counts and lines of code. Each factor should be measured the same way, converted to, or normalized so that we don't combine function points and lines of code.) Each factor should have the same meaning across an industry segment. If a value cannot be ascertained (e.g., lines of code for a purchased dll library), then that factor is not included in any measures or metrics.

---

[5] Financial accounting has used a single summarizing model since the 18th century:

Assets = Liabilities + Equity

The equation basically states that the financial value of an entity (the Assets) is equal to what the entity has been given or earned (the Equity) and what it has been loaned (the Liabilities). Equity changes through operations (revenues and expenses), gains and losses, and owners' contributions and withdrawals but the equation always balances.

[6] The symbol ◊ is used to denote the beginning of the lessons

Accounting Concept	Software Testing Implications
Accounting Tenets – Assumptions, Principles, and Guidelines.	The rules and responsibilities for software testing could be described using a similar three-tiered approach.
Monetary Unit Assumption – All values are counted in dollars. [When a dollar value cannot be assigned, it is not recorded.]	Values are all stated in (or converted to) one measure. If the value cannot be ascertained it does not become part of any measures or metrics.
Basic Accounting Cycle	Software testing and reporting may follow a similar cycle.
Generally Accepted Accounting Principles ("GAAP") – A combination of basic accounting tenets, rules and standards from FASB, and the generally accepted industry practices.	The tenets are only a foundation for rules, standards, and methods. Rules, standards, and methods may vary by industry.
Materiality Guideline relaxes GAAP requirements when the impact is small.	Some discretion is allowed in adhering to conventions when doing so costs more than it's worth.
Full Disclosure – Reports disclose all information thought to be important to an investor or lender within the report or in the notes to the report.	Report all relevant information without regard to its implications. Then clarify or qualify the information as necessary.
Conservatism Guideline – report minimum income values and maximum expenses	Report all potential problems and unknown outcomes in the data, with footnotes (if deemed necessary)
Balance Sheet (Current balances)	A report describing the state of testing at a specific point in time.
Income Statement (Profit & Loss)	A report showing the net effects of activities during a specified time period.

Table 1: Summary of Implications

◊ The ***Basic Accounting Cycle*** has five parts:

1. *Identifying and recording* business transactions
2. *Posting* to journals and the general ledger
3. *Adjusting* the general ledger to reflect actual business operations (e.g., bad debts, accrued interest, taxes, etc.)
4. *Preparing financial statements* for the time periods
5. *Closing the books* (resetting accounts to zero for the next accounting cycle)

A ***Basic Software Test Execution Cycle*** might consist of:

1. *Performing tests and recording results*
2. *Updating quality data* to reflect the new results
3. *Adjusting quality data* to reflect actual business operations (e.g., removing the results of obsolete tests, reflecting reclassified defects, updating code size measures, etc.)
4. *Preparing quality reports* for the time period(s)
5. *Re-Baselining the records* (resetting counts to zero for the next testing cycle)

The testing cycle described here covers only running and reporting test results, thus excluding test planning, logistics, test case creation, maintenance, etc. Even so, quality data (in part 3) is seldom adjusted. This causes distortion of the quality picture from the data. For example, rerun tests are counted the same way as running new tests, and tests results from older versions of software are accumulated with test results from the current version. Some of the distortions might be removed (in part 5) since the counts for measures and metrics would restart each test cycle.

◊ Accounting follows a set of ***Generally Accepted Accounting Principles*** ("GAAP"). GAAP is extremely useful for standardizing and regulating accounting definitions, assumptions, and methods. It legitimizes industry differences and codifies how standards for accounting must be defined and justified. Although variations exist between industries, using GAAP results in consistent reports over time and across industries.

Consistent reporting over time allows fair assessment of the progressive performance of an organization. This context enables stakeholders to understand, evaluate, and compare financial reports at different points in time, understand financial health of organizations and industries, and provides a basis for ascertaining the validity of financial statements. It isn't useful to compare a financial statement for one time period where revenue is counted when money is deposited in the bank (cash accounting) with a financial statement from another time period where revenue is counted when goods are shipped (accrual accounting)[7]. Adherence to GAAP assures that standard accounting methods are consistently applied within an organization and if methods are changed the impacts of the changes are fully explained and separated.

Adherence to GAAP results in comparable statements by organizations within an industry segment (to the extent they interpret and implement the common industry practices the same way). Although financial statements from banks are very different from manufacturing

[7] Cash accounting recognizes revenue based on how soon customers pay, while accrual accounting recognizes revenue based on how fast goods are shipped. Revenue is usually lower under cash accounting.

organizations, statements from different banks are largely comparable, as are the statements from manufacturing organizations. Financial statements for all firms are superficially very similar in terminology and format because industries share the same foundational principles (GAAP), and the differences are understandable and justifiable within each industry segment.

There are three primary parts in ***GAAP***:

1. The basic accounting tenets (assumptions, principles, and guidelines)
2. Detailed rules and standards issued by the Financial Accounting Standards Board (FASB), a group of accounting experts independent of all other business and professional organizations [8].
3. The generally accepted industry practices for each industry segment

The software testing profession would benefit from identifying and standardizing generally accepted software testing principles (GASP) and identifying some of the more and less applicable contexts where the principles apply. A structure similar to GAAP might be created for GASP. A Software Testing Standards Board (STSB)9 would need to be created to identify, articulate, oversee, and adjudicate software testing issues. The STSB would represent the many constituent private sector stakeholders and use an open decision-making process to establish GASP standards. STSB members would be expert software testing stakeholders and need to severe all connections with the firms and institutions they served prior to joining the Board. The members of the board would be selected based upon their knowledge of software quality, software testing, business, and a concern for the public interest in matters of software testing and reporting.

Under GAAP, many accounting rules require accountants to apply expert judgment in deciding key assumptions that have "material" impact on the reported results (e.g., whether to use cash or accrual accounting methods, or accounting for software development costs as a period expense or an investment to be recouped by future sales). Similarly, software testing is context-specific and requires expert judgment to decide on key factors in testing and reporting. GASP must account for the relatively young state of the science and large variation in contexts for software testing. Any "best practice" for software testing must be applicable without exceptions or qualified so that following it is not mandatory (which makes it a "good practice" in some contexts and not applicable in some others).

◊ The ***Materiality Guideline*** relaxes certain GAAP requirements if the impact is not large enough to influence decisions so that users of the information should not be overburdened with information overload. The general rule for assets depreciates (uses up) its value a little bit at a time over the useful life. To a large corporation it may cost more to compute and track depreciation for assets costing a few hundred dollars than the item itself costs. The materiality guideline allows accounting for those assets as immediate expenses because the numbers on the financial statements will not change.

An example parallel guideline for software testing would be removing low priority/low impact defects from defect counts. The metrics might give a false impression if it includes low priority defects not likely to be fixed. The biased metrics could lead stakeholders to poor conclusions about what to test or what to fix. Another example of materiality might be elimination of report items that do not provide any information that could change stakeholder behavior. The cost of gathering and reporting the information is wasted when no stakeholder will make decisions based on it.

◊ The ***Full Disclosure Principle*** and ***Conservatism Guideline*** for accounting state that all information thought to be important to an investor or lender should be disclosed within the statements or in the notes to the statements and it must include all known information that could negatively affect the financial statements. An example would be the possibility of losing a pending lawsuit that could force the company to pay out a large amount of money. However, speculative information that might positively impact the organizations financial is not allowed to be included in the body of financial statements (except as footnotes).

The parallel for this in software testing is for reports to include all information known at the time of report generation. For example, non-reproducible errors would need to be documented (at least in footnotes). The preliminary outcomes from the most recent test runs would be included in all measures and metrics. Footnotes for the measures and metrics can explain and clarify if pending analysis of some test failures might change their result to passes. Another example would be where a defect report is logged for a potential problem immediately before measures or metrics are reported. Even though analysis might lead to closing the report (e.g., as 'Not a Problem' or 'Duplicate), the report should be counted as an error in measures and metrics, using footnotes to explain and clarify.

◊ Two separate financial reports are used to describe the current state of the organization. The ***Balance Sheet*** provides a snapshot of the financial state at one point in time. This often appears in financial reports with the corresponding Balance Sheet values from other time periods for comparison. The ***Income Statement*** (a.k.a.,

[8] See http://www.fasb.org/facts/index.shtml#mission for facts about the FASB

[9] Bryan Kocher published an article "A Model for Software Practices from the Accounting Profession" (IEEE Software, Volume 17 Number 1, January/February 2000) calling for the establishment of an Information Systems Standards Board (ISSB) to create a set of standards for software system design and construction, modeled on the FASB and accounting professions. I do not share his confidence that Generally Accepted Programming Practices exist and can be simply articulated, but I do agree that establishing a system based on *personal responsibility* is preferable to *endless regulation*.

Profit and Loss Statement) describes the recent financial changes in accounts due to business activities during a period of time (e.g., monthly, quarterly, or yearly). Together, the two reports show where the organization stands and how the organization has been doing recently.

Software test reports can be logically separated in a similar way. One component in the report covers the state of the software being tested. This quality indication is a statement of current software behavior, completeness, or readiness. The other component covers the recent history of changes in behavior, completeness, or readiness. The first part is about the product and includes information needed to understand the quality of the software at that one point in time. The second part is about progress and it describes recent changes in the product quality.

Conclusions

The rules for accounting have been established and refined for centuries, where the computer software testing profession operates without established rules and has existed for roughly half a century. Although software testing is in its relative infancy and we are still establishing a vocabulary to describe what we do and I think that valuable lessons for software testing are available from accounting. These lessons will help software testing evolve and mature more quickly.

The structure and concepts for the rules of accounting can be used to provide a useful conceptual framework for a set of rules that could govern software testing. A major lesson from accounting in this regard is that rules differ from one segment of the industry to another because of the widely differing requirements for each. Another lesson is that it is possible for a profession to standardize and regulate itself through personal responsibility rather than by externally imposed regulations.

The software testing profession will be able to approach standardizing the rules when the information we measure can be unambiguously assigned values on a single scale (or a few distinct scales). Monetary value is universally applied in accounting in spite of the scores of currencies in use and accounting methods work the same way for all of them. Fundamental measures in software testing are not well defined yet: “test case,” “defect,” “defect report,” “code complexity,” and “program size” are examples of commonly used terms that have several, often conflicting definitions. The differences in the terms are not only due to different industries, but many are hotly debated between industries and sometimes within a single company.

Generally Accepted Accounting Principles validate and institutionalize differences in rules across industries. Different rules, formats, or terms are due to justifiable context specific differences and the same accounting principles are applied across the industry segment. This results in accounting reports across each industry that can be easily read, interpreted, and compared. Similarly, context differences in different industries that test software result in different testing and information requirements. Similarly defined tenets in software testing could facilitate acceptance of common foundational rules and approaches while allowing for context specific solutions. Although I have not encountered any universal “best practices” in software testing that I could professionally support, they may exist, and context-specific “good practices” have been known for decades. The software testing profession would benefit from standardizing them and identifying some of the more and less applicable contexts.

A materiality guideline allows for some discretion in measurement and reporting of useful information to an appropriate level of detail. This flexibility is critical since the requirements and practices for software testing varies so much today. Reporting standards that require conservatism and full disclosure could also benefit software testing. An example of conservatism in testing would be a requirement for test reports to include an explicit list of tests that were not performed (including ones considered but not created) and the rationale behind their not being done. Full disclosure would require that any known information that might influence stakeholders’ decisions be included in reports (e.g., non-reproducible errors). Such standards would remove ambiguities that may be exploited to bias reports or justify sub-optimal testing by providing all of the potentially negative information that might influence stakeholders’ decisions.

Software testing might also benefit from considering two distinct types of reports: one to display the current state and another to describe recent activities. A report of the current state provides a snapshot of status, while the activity report shows changes occurring during specified time periods. One shows where we are and the other how we got here.

Because the state of the science in computer science is still young, coming up with generally accepted software testing principles may not be within our grasp anytime soon. However, endeavoring to do so will improve the state of the science by exploring the serious question of why there are different definitions and tenets. Applying some of these lessons from financial accounting could improve the software testing profession. Standardizing terms, definitions, and report formats could reduce some of the redundant work done by nearly all software test organizations, possibly freeing up resources to delve more deeply into the invaluable job of more and better software testing.

Appendix 1

Twelve basic accounting assumptions, principles, and guidelines:

1. Economic Entity Assumption

   Accounting keeps track of business transactions (as separated from personal transactions), so that the finances of the firm are not co-mingled with the finances of the owners.

2. Monetary Unit Assumption

   Economic activity is measured in dollars, and only transactions that can be expressed in dollars are recorded. [This means, for example, that the effects of inflation are ignored in accounting.]

3. Time Period Assumption

   This principle assumes that it is possible to report the complex and ongoing financial activities in relatively short, distinct time intervals. The time interval must be included in P&L and Cash Flow reports, and the specific date on a Balance Sheet.

4. Cost Principle

   "Cost" refers to the amount spent (cash or cash equivalent) when the transaction takes place. This means historical values are not adjusted to reflect increases in value. [Hence, accounting values other than stocks and bonds do not reflect the amount of money a company would receive if it were to sell the asset at today's market value.]

5. Full Disclosure Principle

   All information thought to be important to an investor or lender should be disclosed within the statement or in the notes to the statement.

6. Going Concern Principle

   Accounting assumes that an organization will continue to operate and will not liquidate in the foreseeable future. This allows deferring of prepaid expenses until future accounting periods.

7. Matching Principle

   This principle requires that expenses be matched with revenues in the same time period [using an accrual basis of accounting], even when paid at different times. Where the expenses cannot be matched with particular revenues (such as advertising costs that increase future sales), the expense is charged in the period it is incurred (when the ad is run) and the revenue recorded when the sale is made.

8. Revenue Recognition Principle

   Revenue is recognized as soon as everything that is necessary to earn the revenue has been completed for a product or service (but not before that), regardless of when money is actually received.

9. Materiality Guideline

   A modifying convention that relaxes certain GAAP requirements if the impact is not large enough to influence decisions so that users of the information should not be overburdened with information overload.

10. Conservatism Guideline

    If there are two acceptable alternatives for reporting an item, the alternative that will result in less net income and/or less asset amount is chosen. [For example, an accountant may write inventory down to an amount that is lower than the original cost, but will not write inventory up to an amount higher than the original cost.]

11. Cost-benefit Guideline

    A convention that relaxes GAAP requirements if the expected cost of reporting something exceeds the benefits of reporting it.

12. Industry Practices Guideline

    Accepted industry practices should be followed even if they differ from GAAP

Session

Sleight-of-Quality:
A Magical Approach to Testing

Jerry Kominar

Tuesday, July 15, 2008
2:45pm - 3:45pm

Room: Saint David 3rd Floor

Overview

Quality assurance requires a diverse set of skills, demonstrated in complex testing environments. The study of traditional magic principles can help software testers raise their awareness of discrepancies that can be found in these environments, leading to improved quality assurance. Software likened to a magical "trick" offers an interdisciplinary approach to the study of method and effect, enabling us as professionals to approach testing from new angles. Magic in this context will not only educate but will also entertain!

Sleight-of-Quality: A Magical Approach to Testing

Jeremy Kominar
Research in Motion
195 Phillip Street
Waterloo ON

jkominar@rim.com

ABSTRACT

Software Testing is a field that requires a diverse set of skills, demonstrated in a myriad of complex testing environments. The study of magic and deception can help software testers raise their awareness of the discrepancies found in these complex environments, leading to improved software testing. Software, likened to a magical "trick", offers an interdisciplinary approach to the study of effect, through which we can gain awareness of test methods. This paper will draw parallels between the complementary roles of the developer and the tester, and the magician and their audience. Both can benefit from understanding one another's perspectives and from seeing things from a different point of view. Magicians use psychology, philosophy, mechanics, manipulation and theatrics, the five pillars of magic, to deceive their audiences. These pillars of magic, combined with misdirection of attention, provide a breeding ground for distorted perceptions. Similarly, in software testing, these elements of deception can allow bugs to go unnoticed. Without awareness of the realities in which we, the observers, conduct our observations, we as the tester are no better than laymen naively watching a magic trick for the first time. This paper will discuss how the tester can exploit each pillar of magic for the purposes of software testing. By exploiting the knowledge of magicians, we gain a heightened awareness of the testing environment, and strengthen our deductive and reasoning skills.

Categories and Subject Descriptors

D.2.5 [Software Engineering] Testing and Debugging - *Testing*

General Terms

Performance, Experimentation, Human Factors, Standardization, Languages, Theory, Verification.

Keywords

Software Quality Assurance, Pillars of Magic, Psychology, Deception, Misdirection, Observation, Inference, Agile, Exploratory Testing, Personae Testing, Refactoring, Change Blindness.

1. INTRODUCTION

For centuries, magicians have entertained audiences through use of guile, ruse, and clever misdirection. Their methods, shrouded in mystery and known only by a select few, have produced effects that are both unbelievable and unfathomable. This paper examines the five pillars of magic used by magicians to deceive their audiences and how software testers (hereinafter referred to as testers) can use similar knowledge in a testing context. Readers may be perturbed by what they have read, and may wonder how deception could possibly have any relevant place in a serious discussion about testing. However, skeptics are encouraged to read further and to temporarily suspend their disbelief.

In their book *Rapid Software Testing*, James Bach and Michael Bolton state that "testing magic is indistinguishable from testing sufficiently advanced software" [1], and it is this belief that warrants further exploration. Parallels exist between the roles of 'The Tester' and the 'The Magician', and 'The Tester' and 'The Magician's Audience'. This paper will enhance the tester's ability to comment

on the quality of software by exposure to the five pillars of magic: psychology, philosophy, mechanics, manipulation, theatrics and the parallel roles in magic and testing. This exposure will allow the tester to be able to approach testing from a new perspective, which will provide a heightened sense of awareness of the testing environment, and result in strengthened deductive and reasoning skills.

2. MECHANICS

The first pillar of magic discussed is 'mechanics' – the basic methods and procedural workings of a system. With respect to magic, the mechanics define how a trick comes to fruition from start to finish. Most tricks are composed of two elements, method and effect, or input and response. In software testing, test cases quite often require the same elements: user input and observation of a system response.

2.1 Where Do Magic Tricks Come From?

Similar to the software development life cycle (SDLC), magicians have their own processes for creating tricks. In fact, many of the process models of the SDLC are similar in nature to those in the magic development life cycle (MDLC). The similarity of these models can be seen in a discussion of the most traditional model in each disciple, the waterfall model.

The tester's traditional role in the waterfall model involves testing software created by development at the end of the development cycle. However, testers can extend their sphere of influence by being involved throughout the entire process of the SDLC through review and analysis of requirements, or in helping to create document specifications. Involvement in each phase of development is what gives magicians the opportunity to minimize bugs and maximize effect. To have this same impact, testers should engage in the software development process at the earliest possible opportunity.

Now consider the waterfall model in the MDLC. Before the magician's audience is privy to any sleight-of-hand, or grand illusion, the magician first considers the requirements of the effect her or she wishes to achieve. Will the trick be with cards or coins or some other prop? Will the setting be one-on-one and in close quarters, in a small group, or on stage at a distance? Will the audience be actively involved in the trick or will they simply watch? The answers to all of these questions manifest themselves in the requirements of the effect. At this point the trick is conceptualized, keeping in mind that a trick that meets all the requirements may already exist. During the development phase, the magician learns of, or possibly develops, methods to achieve the effect that the trick requires. After practice and demonstration to people 'in the know' or those who are very tolerant, the magician performs for their audience, their end-users. The presentation phase is where the audience suspends their disbelief, the magician distorts reality, and nothing is what it seems. One cannot help but notice the similarities of the models discussed, so it is natural to believe that interdisciplinary approaches to the process may prove both beneficial and insightful. Another similarity is that in both cases, it is important to practice and learn in different settings so that one is prepared for the moment when testing or performance is required and no routine or test case has been created.

Software development has its own developmental models and subsequent mechanics that govern its testing activities. By following a repeatable process and through diligent practice similar to that of magicians, testers can have the opportunity to refine the mechanics of their model, reflect on historical trends, and improve upon quality.

2.2 How Does This Thing Work Anyways?

Testers need to know the mechanics of the tests that they are running. By constantly questioning the rationale of the test cases one can not only learn more about the software and test case coverage of a particular test suite, but also, they can hone their testing prowess by thinking critically. The following questions are great candidates to promote critical thinking while testing:

"What is the usage scenario for this test case?"

"Why is this test case important?"

"What part of the code does this test case cover?"

"Can this test case be tested differently? How?"

Simply following test case method steps without knowing why the steps are to be followed results in less effective testing due to missed opportunity. Each time a tester executes a test, the opportunity

exists to question its validity. If testers do not question the test cases or subsequently challenge the system, the same bugs will be found and new ones will go unnoticed. Magicians will often start with an effect and then construct methods that can achieve it. The methods to their tricks and the audience responses are evaluated throughout the life of the trick resulting in continuous improvement. Through the study of system response and effect, testers and magicians alike can gain awareness of methods used stimulate desired responses. These methods also help to identify ways that users interact with the system under test.

2.3 How Can It Be Made Better?

Magic happens in the minds of the spectators. A good magic trick will only reveal effect and not method. Therefore, does the method used matter if the effect remains the same? The answer is, YES! While performing, magicians aim to minimize their effort while maximizing the impact. This efficiency results in a reduction of the chance of failure and of potential exposure.

Testers should also ascribe to this efficiency equation. Bret Pettichord defines 'refactoring' as [2], "a process in agile development that improves the way in which existing code is designed." The technique involves "changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure." Refactoring is also applicable to testing. To testers, method is paramount. It is ideal to reduce the number of superfluous test cases or test case steps in a particular test suite to free resources with the provision that test coverage is not lost. However, the tester must also be aware that multiple paths may lead to the same goal – in which case each new path could potentially be as important as the previous. These paths will represent different usage cases of the end user. By refactoring test cases, testers can work towards creating less bloated tests that still have the same test coverage but take less time to run.

2.4 Can I See Those Cards Please?

Magicians have used props to assist in achieving illusion for quite some time. Take the magician's wand, for instance; with a point and a simple waving motion, the magician can create instant misdirection of attention. However, props like the wand are not the only deceptive objects utilized in performances.

Magicians can also employ gaffs in their effects. Gaffs are objects or tools that magicians have modified in some way to serve a specific purpose – such as a coin that has both sides depicting heads for instance! Most laymen fail to consider the wide variety of possible options that are at the magician's disposal. For instance, when thinking about the objects that magicians use in their tricks, laymen are deceived in to thinking that the objects they observe are in fact the same objects that they are familiar with and have handled other situations. Coin magic is a very effective form of magic as coins and money are something that people can easily relate to, and magicians will exploit this familiarity to lull the audience into a false sense of security. You need to think outside of the box to be a magician or to discover a magician's methods.

If tools exist that can help magicians achieve the impossible, it is appropriate to wonder if similar tools exist for testers. As it turns out this is the case, and one of the available tools is more common than one may think. The tool in question is, 'other testers!' In the same way that a spectator develops a clearer mental model of the method of a trick once they have been exposed to how it works, a tester's mental model of the product becomes clearer through the course of testing it. However, just as not all spectators believe a trick works the same way, not all testers share the same mental model. Testers can leverage other testers' mental models by observing how they test, learning their test methods, and using their own as well as these newly learned methods in their testing. Similar to how [3] "magicians become experts by listening to and watching other successful magicians," testers too can benefit from observing testing methods and patterns of their peers and contemporaries in an attempt to replicate them. By gaining exposure to testing subcultures, the tester's body of knowledge will increase. As knowledge increases, mental models develop and become engrained. Active mental model development will afford the tester the ability to reason more soundly and as such comment more comprehensively on what they are testing. In fact, as discussed in their book *Lessons Learned in Software Testing: A Context-Driven Approach* Cem Kaner, James Bach, and Bret Pettichord [4] "testers

don't like breaking things; they like dispelling the illusion that things work."

3. PHILOSOPHY

The second pillar of discussion is philosophy - the critical study of the ideas and basic principles that govern a body of knowledge, especially with an aim to improve or reconstitute those principles.

3.1 What Does the Future Hold?

Predictive models of software development focus on planning the future in detail. Teams that use these models can often report in detail on the features and tasks that are planned for a project, assuming little variance. However, they tend to have difficulty changing direction when the need inevitably arises.

Earlier, the MDLC waterfall model was described, but what was not discussed about this model is that it is extremely idealistic and not practical for most magicians (sorry for the deception). For the most part, magicians are quite agile – not necessarily in a literal sense, but a developmental sense. The MDLC is more commonly viewed as an iterative process. It would be extremely naive to believe that one could read the method of a magic trick and then shortly after, present the trick to an audience and achieve the full effect. Even if the trick could be executed under these circumstances, what would the quality of that trick be? Would it have been deceptive? Could the performance have been enhanced? Will the trick work the next time? Predictive models can be used to anticipate magical effects, but magic is created both in the environment in which it is presented , as well as in the mind of the spectator. Both are subject to change and variability. These dynamic factors call for a more adaptive and iterative approach to achieving consistent and deceptive magical acts.

Many of the tricks which magicians perform involve prestidigitation (skilled sleight of hand). Before the presentation phase, the experienced magician will rehearse for their confidants not only the effect, but also the sleights that comprise it. Since many effects are experienced in the spectator's minds it is essential that the magician collect frequent and early feedback in the MDLC, and an iterative approach affords them this ongoing opportunity. Magicians are proponents of this iterative approach because unlike the SDLC, the MDLC does not have a maintenance phase. As a result, if the magician accidentally exposes something that was not meant to be seen by the audience, there are no 'hot fixes' or 'updates' that the audience can download to correct the problem. The audience is now less susceptible to deception as they have learned something new about magic. It is also essential that a magician understand as much as possible about their trick, their tools, their audience, and the environment. Understanding the relationships between these multiple factors allows for better adaptation and agility. Without question, testers can benefit from possessing the same awareness as they are afforded the same benefits, however, in a testing context.

A common assumption is that the earlier a defect is found the cheaper it is to fix. By using a more iterative approach during the development life cycle, defects can be discovered and potentially resolved more regularly – this principle applies to both magicians and testers. However, it must be remembered that context plays a pivotal role in any testing model, and must be appropriately incorporated. In considering context, it is not always the case that one method should be favored above all others. With the dynamic nature of environment, stakeholder needs and requirements [2] "there are good practices in [different] context[s], but there are no best practices." Proponents of the context-driven testing school of thought believe in the importance of context, and rightfully so.

4. PSYCHOLOGY

The third pillar of magic deals with the psychology and the role which it plays in deception. The nature of psychology is such that it is applicable to any field or discipline that involves human factors. Software testing certainly falls into this category. Through exposure to the nuances of how the mind works and how magicians exploit these nuances, testers can learn to become better observers.

4.1 Why Are We So Easily Deceived?

Cem Kaner et el, in their book *Lessons Learned in Software Testing* state that [4] "testing is grounded in cognitive psychology." Just as audiences are fooled by the psychological principles applied by magicians, testers can be fooled by psychological principles that deceive them as to the quality of the

software. By the exploitation of human limitations in reasoning and perception, both the magician's audience and testers can have their attention misled, allowing discrepancies to go unnoticed. However, there is hope, and it can be found in better understanding our cognitive limitations. According to Kaner et el [4], "if you want to be better than good, studying cognitive psychology will help you understand the factors that affect your performance as a tester, as well as the factors that affect how people interpret your work."

Cognitive psychology can help us understand these limitations. According to Bach and Bolton [1] we "misunderstand probabilities, we use the wrong heuristics, we lack specialized knowledge, we forget details and we don't pay attention to the right things." Magicians commonly exploit these limitations, and recognizing these limitations will enable us to become better testers. If, for instance, we appreciate that there are holes in our vision that we unconsciously fill, or that sounds and lights can divert our attention, we will be able to [5] "design better tools, and create better interfaces that work with the grain of our mental architecture and not against it."

It has been stated that [6], "you're harder to fool if you know you're a fool." By knowing one's susceptibility to being fooled, the tester is also able to conduct risk analysis on each of these susceptibilities. Not all of the idiosyncrasies of the mind may be relevant to the testing context, but the savvy tester will want to learn more about the cognitive factors that may affect their testing. If someone believes that they are immune to the psychological factors that affect their testing, they clearly should think again [6], "con artists say that the person easiest to con is one who is absolutely convinced he cannot be fooled. You can put that principle to work for you as a tester. Convince yourself that you are *easy* to fool. It's not hard, just watch carefully for your own mistakes while testing. Notice whenever another tester finds a problem that you could have found, but didn't."

4.2 May I Have Your Attention Please?

Human beings interact constantly with their environment through observation. The environment constantly bombards them with stimuli. In fact, they are subjected to far more information than their brains can even process. It is the brain's job to filter this information according to where they choose to direct their attention. The brain [5] "is not a clear mechanical system such as clockwork or like a computer program; giving the same input won't always give the same output. Automatic and voluntary actions are highly meshed, often inextricable. Parts of vision that appear fully isolated from conscious experience suddenly report different results if conscious expectations change." Magicians are constantly diverting their audience's attention away from one thing in order to focus it on something else. By learning how magicians divert attention, testers can learn to avoid falling prey to similar tactics during their testing. Bringing subconscious processes to the conscious mind allows the tester to become more aware of their actions and their environment, resulting in increased observational skills and attention to detail.

4.3 How Would You Like That Framed?

The ability to frame actions or events strategically, given the audiences' particular perspective, enables the magician to focus the audience's attention wherever they so desire. While attention is focused in one place, subterfuge can take place in another. In 'close-up' or 'parlor' magic, the audience is in close proximity to the magician. While this may seem advantageous to one who wants to discover the magician's methods, it is in fact just as easy, if not easier, for the spectator to be misdirected. As proximity to the magician increases, tunnel vision overtakes the audience's perception. The magician can use not only his eyes and hands for misdirection, but he can also make use of his voice, body language, and other props to deceive. However, if the audience were to move their vantage point further away from the magician, their field of vision would increase, making them less, not more, susceptible to deception. By changing the framing of a magic trick, the audience can increase the breadth of their model, and be more likely to discover how the effect works.

The principle of framing is also used in the field of quality assurance. Framing can affect many of the tester's activities in the SDLC. These tasks include but are not limited to testing, test design, and requirements review. By reducing the scope of focus for a particular task which is currently in

frame, the tester can give undivided attention and resources to the task at hand and increase the depth of investigation, but this focus comes at a cost. While resources are devoted to one activity or task in such a focused manner, other areas will tend to be neglected. To overcome this issue, Kaner, Bach and Pettichord [6] suggest the "plunge in and quit" approach, which continually adjusts the depth of focus. "The great thing about this method is that it requires absolutely no plan other than to select a part of the product and work with it. After a few cycles of the plunge in and quit, you will begin to see patterns and outlines of the product. Soon, more organized and specific testing and studying strategies will come to mind. It works like magic."

The plunge in and quit method has certain advantages, but it does not provide breadth of focus, and so it is best used in conjunction with framing / reframing. Focusing / defocusing, demonstrated in the plunge in and quit technique, along with framing / reframing, which constantly redefines the testing charter throughout the course of testing, comprise the basic elements of exploratory testing. Testers design their tests by using their mental model of the system they are testing. Exploratory testing gives the tester the freedom to define and or redefine their models of the software by exploring any areas that may need further clarity. As Kaner et al. state [7], "because testing is sampling and your sample can never be complete, exploratory thinking has a role throughout the test project as you seek to maximize the value of testing." Both depth and breadth of mental models can be achieved through exploratory testing. This approach quickly builds richer models of the product in the tester's mind through curious yet focused exploration and observation of the software. Once a preliminary model that reflects their current understanding of the system is defined in the tester's mind, the model can be probed for areas of 'opportunity' or uncertainty. Areas of uncertainty in a model are often considered 'off the happy path testing', a prime breeding ground for bugs.

4.4 Did I Observe or Infer That?

Bach and Bolton comment that observation and inference have the ability to be easily confused. They state [1], "Heuristics provide the tester and magician's audience with a fallible means to solving problems or making decisions. When we are observing something, our minds are not passive, but rather, they ask questions about the sensory data from the environment." Some everyday heuristics are as follows:

> *Where there is smoke there is fire.*
>
> *By finishing all testing and finding no bugs there are no bugs to be found.*
>
> *Regression test cases that did not fail before should not fail now.*
>
> *People cannot saw other people in half without critical injury.*

Heuristics are used by the observer to make inferences about what they have observed. It is stated in [8] that "Because they are *reasonable, low-cost* shortcuts, heuristics can present *more valuable* solutions for the present circumstances." In short, heuristics enable us to make sense of our observations without using excessive observational resources. However, issues arise when a heuristic is used in the wrong context or the heuristic is no longer valid. In the case of our regression test case heuristic, take into consideration that code changes could have been made since the last execution of the test. While this may be unlikely, if it happened, the heuristic would break down. This idea is summarized in [1] by stating, "It's easy to miss bugs that occur right in front of your eyes. It's [also] easy to think you 'saw' a thing when in fact you merely inferred that you must have seen it."

Taking inference a step further can lead to the phenomenon of change blindness. Change blindness refers to our inability to notice information that our heuristics tell us is extraneous. Tom Stafford and Matt Webb in their book *Mind Hacks : Tips and Tools for Using Your Brain* [9] claim, "We don't memorize every detail of a visual scene. Instead, we use the world as its own best representation – continually revisiting any bits we want to think about. This saves the brain time and resources, but can make us blind to change." Despite this efficiency of viewing the world in this way, it is sometimes important to revisit specific sections of the visual scene that do not have our attention. With all said and done is there anything that can be done to combat this affect? Again, the answer is, YES!

Bach and Bolton [10] suggest to, "[pay] special attention to incidents where other people notice things that you could have noticed, but did not." This will help you to understand where you tend to be susceptible to change blindness. They also suggest probing for further evidence that something has gone awry before basing a statement or a belief on one observation alone. Finally, look at the software from multiple vantage points and gather different types of information, including but not limited to multiple execution results of test cases, review of requirements documents, and as mentioned earlier, the opinions of fellow testers.

5. MANIPULATION

Manipulation is the set of methods by which magicians are able to deceive their audience. Discussion will follow surrounding how these methods may work against the tester, as well as suggesting ways that the risk of deception can be mitigate.

5.1 How Can We See What Is Not There?

Even when anticipated, deception has the unique property of remaining effective. It is stated [11] that, "Despite the audience's knowledge of the deception and its ardent efforts to detect magicians' methods, magicians are consistently effective in deceiving the audience." Similarly, the tester knows that the software they are testing has bugs, and yet bugs still go undetected. Why does this happen despite conscious awareness of this principle? Is the reality is that conscious awareness does not exist? With this awareness can these risk be mitigated? The answer to the first question lies in misdirection and its effects on the attention of the observer. The answer to the second is reader specific, and finally the answer to the third question is, YES!

5.2 How Many Methods of Misdirection Make a Magician Magical?

Magicians have many ways of misdirecting the audience's attention; however not all methods are required to execute any one trick. The choice of misdirection depends primarily on the context. Below, the reader will find a description of the context in which some key types of misdirection are used by magicians. Following each magical context, a software testing context will be given where the respective forms of misdirection could potentially exist and cause bugs to go unnoticed.

5.2.1 Anticipation

John W. Cooley [12] speaking about the great magician Harry Blackstone states that anticipation is "a type of misdirection in which a magician first anticipates that the spectator's attention will be fixed on a critical thing." At this point, for instance, the magician has the opportunity to 'get one ahead' of the audience by placing a rabbit into their hat before walking out on stage and producing one from the same location. Once on stage, the audience will try to anticipate what the magician will do to deceive them, however, at this point the dirty work has already been done.

Anticipation in software testing lures testers into a similar false frame of reference. It may, for instance, cause the tester to believe that the code for a new feature has detrimentally affected an old feature when in fact has not, or worse yet, to assume that it has not affected the old feature when in fact it has. To mitigate the risk of this type of misdirection affecting the tester, he or she can communicate with the development team before testing, ensuring that any modifications and ramifications have been explained.

5.2.2 Premature Consummation

Cooley [12] continues to describe Blackstone's thoughts on premature consummation. "Using premature consummation, [the] magician gets spectators to relax attention prior to the magician making a necessary move." By misleading the spectator's attention into prematurely believing a deceptive act has already occurred, the magician can then cause the spectator's attention to relax. At this moment, since vigilant attention is no longer required, the magician can make the move necessary to achieve their goal.

This form of misdirection can take place in the SDLC in the following common scenario. The tester has completed testing, bugs have been logged, fixed, and the fixes have been verified – therefore technically the problems should no longer exist. At this point, the tester may be inclined to 'relax' having done their due diligence in verifying the fixes. However, as it turns out the bug was fixed but core functionality was broken. While fixing one bug another bug can potentially be introduced into

the code. The tester's familiarity with the software can be utilized in a similar situation while verifying bug fixes. By drawing upon their model of how the software works, they can infer what areas of the software may have been changed because of the fix, if this information is not explicitly or readily available. Again, development is a tremendous resource in this situation. If the option is available, testers can communicate with development in order to find out the scope of the changes that went into the fix and test accordingly.

5.2.3 Monotony

A third type of misdirection is described as monotony. Cooley [12] states that, "Monotony, is effective because of its simple premise that the audience's attention becomes dulled after vigilance of a repetitive act of some duration. Monotony is the misdirection sometimes used to produce a rabbit from a top hat. Magicians take several silk scarves from a top hat and deliberately allow them, repetitiously, to fall to the floor. Consequently, the audience's attention wanders. Magicians take advantage of the attention loss by sneaking a rabbit into the pile of scarves. Then, they produce many more scarves in a sweeping gesture again and again, as if they are multiplying uncontrollably, and then scoop the whole pile, rabbit and all, into the hat."

Testers can be very susceptible to this form of misdirection. Take for instance regression testing. The novelty of regression wanes rather quickly in comparison to test cases that test new features and new code – after all, statistically there should be a higher probability of finding bugs in new code. Hence, the possibility of the tester letting their guard down may become a reality while running test cases they find less interesting. This situation can be avoided by introducing novelty into regression in the form of exploratory testing using the plunge in and quit approach. By conducting careful risk analysis the tester may temper regression with exploratory testing of legacy features by using rotating charters to discover new bugs.

5.2.4 Confusion

Cooley [12] describes Blackstone's fourth type of misdirection as confusion. "By using confusion, the magicians present so many varied individual interests for the spectators' observation that it is impossible for the spectators, in the limited time available, to differentiate the significant from the insignificant. Spectators must make a "desperate and hurried attempt to inspect and weigh the multiple interests presented, [thus they are] able to give only superficial, hasty attention to the individual things before [them]." Thus, the spectators' attentions become scattered … Confusion is different from the monotony stratagem in that in using monotony, all the details are identical and success depends on tiresome sameness, whereas in using confusion, the individual details need not be the same and success depends on "disarray, turmoil and disorder."

Kaner, Bach and Pettichord [6] speak to confusion in the SDLC accordingly: "Confusion should be used as a test tool." Testers need to realize that confusion that they find in the product may be tied to confusion in specification documents, confusion in implementation, or the product may simply be broken. Kaner et el [6] go further to state that "the more knowledge the tester possesses about the product and testing in general, the more powerful a compass their confusion becomes, showing them where important problems lie."

5.2.5 Diversion

Blackstone's fifth type of deception is described by Cooley as diversion. Diversion plays on the audience's psychological requirement to shift attention to things that are deemed to be more critical by the brain. By substituting something critical for something of lesser importance, the magician can draw attention toward the more critical object. Cooley [12] states that, "Diversion occurs when magicians achieve a lack of spectator attention at the proper point in a routine by directing spectator attention from the proper course and toward a false course."

The effects of diversion can be seen in instances of the SDLC which suffer from resources constraints. What is viewed as critical can vary daily due to shifting priorities and changes in requirements and features. To mitigate the risk of this type if misdirection one approach is for the project to adopt a more agile development life cycle. By taking a more iterative and agile approach, testers and development can inform each other of updates during testing and feature development since they are in constant communication. The risk of

diversion can be minimized by dissemination of knowledge and project planning that revolves around short turnaround times and consistent change.

5.2.6 Distraction

Blackstone is quoted [12] as saying "with diversion, the key is a natural and inconspicuous approach, whereas with distraction, the key is surprise at a time when the audience is not suspicious … [distraction] implies [the] inability on the part of a spectator to think properly about anything."

Distraction is all around us and in many facets of our lives. By overwhelming the audience's attention the magician can execute actions that will go unnoticed while attention is diverted. One of the 'advantages' that this form of misdirection has over the others is that sources of distraction can generally be identified. Risk analysis will help identify 'distractions' in a project and mitigation strategies can be determined accordingly [3]. Some common distractions for testers are changing priorities, changing specifications, changes to environmental, as well as changes to resources. Once identified as risks, these distractions can be dealt with in the appropriate manner. Creating a risk catalogue that identifies each risk, its probability and its mitigation strategy can assure risks have proactively been considered.

5.2.7 Specific direction

Blackstone's comment on the final type of misdirection, 'specific direction' is also noted in Cooley's paper. Specific misdirection is, "a bold, undisguised act of definite direction. Specific direction can be an act, a verbal direction, or a gesture." [12] that lures the audience's attention towards a particular place.

Specific misdirection in testing has the ability to cause the tester to focus their attention on a particular task when their attention should be given to something else. In the case of specific misdirection the tester could know that their attention should be focused elsewhere, for example, if they believe they should be testing more important or error prone aspects of the product when they are busy testing or doing something else. An example to consider is when a tester has found a bug and continues to drill down and explore the issue while allowing other testing and areas of opportunity in the code to go unexplored.. However, the tester may be assigned testing tasks by someone else who does not share their same opinion. Avoiding this type of misdirection is easier when the tester knows more about the product they are testing. Through dialogue with management, the tester can convey the potential risks of not testing certain areas and focusing on others.

5.3 Is A Second Opinion Needed?

A valuable tool for both the magician's audience and the software tester is a second pair of eyes. In both contexts, multiple perspectives can add value. The more information that is available about a given model, especially from different perspectives, the more inferences can be made about how the model works. Another benefit to this approach is the fact that familiarity of subject matter is not necessarily an asset. Less familiarity with a subject will lead to different heuristics used by the observer to make inferences. The combination of inexperienced and experienced observation will result in focus and attention being given to different aspects of the product. Assumptions made by the experienced tester will be questioned by the inexperienced tester and vice versa – sharing tester experiences can help determine whether or not the product meets requirements. Just as in magic, where not knowing how tricks work can lead to great ideas, not having specific knowledge of how the software should work means that the sky is the limit for the observer's imagination – this in turn fosters creativity.

6. THEATRICS

The final pillar of magic is 'theatrics' and the presentation of the effect. Given that magic happens in the spectator's mind, it seems prudent to know what is important to the spectator and how they will perceive a particular effect. By knowing how the spectator will react, the magician maximums the impact of effect. The same principal extends to the field of software testing. Knowledge of how the software works and how the end-user will respond to system usage scenarios allows testers to comment on the usability of the system.

6.1 Do I Know You? Should I Know You?

In order to understand and learn from magicians one must first learn about their processes and operating

environment. Magicians know their environment and audience very well, as this is essential to their livelihood and ability to entertain. Testers too can benefit from gaining knowledge about their project stakeholders and product environment. For instance, by having developed a rapport with the development team through dialogue and interaction, a tester can gain more intimate knowledge of the software and its implementation. In a similar fashion, knowing the needs of the end user allows the tester to more accurately predict how the software will be used while testing accordingly. Since software generally has more than one end user, having knowledge of each user's motives and objectives allows the tester to engage in persona testing. In this form of testing, test cases are modeled specifically around use cases which are applicable to each type of user that interfaces with the system. For instance, an administrator may conduct different actions than a normal system user. It has been said that "knowledge is power," which is certainly true for magicians, however, in the context of software testing, a more apt phrase would be, "knowledge breeds quality." Testers provide information about quality, and that information can help lead to better quality. By possessing information about quality, testers are afforded the liberty to improvise while testing generally leading to novel subtitles of the software and of course, bugs.

6.2 How Good is Good Enough?

As the old adage goes, practice makes perfect. Even very skilled magicians must continuously sharpen their skills in order to maintain their prowess. Practice for the magician not only involves performing repetitive actions ad nauseam, but it also involves practicing tricks in different contexts. A trick may work in specific contexts: while seated or standing, in low light, for your pet, or while performing for friends and loved ones; however there will ALWAYS be new contexts for performing tricks that offer new learning opportunities.

Testing is no different. Despite all of the bugs that a tester may have uncovered, there will always be more bugs to find in different software, in a different context, and related to different implementations. It is for this reason that testers must continue to pursue testing and new test methodologies. As bugs are found and fixed in software, the overall quality is improved, which makes the tester's task of finding bugs more difficult. Testers must continue to pose new questions to the system and be willing to accept the results. Testing is all about asking the right questions, about the right things, and at the right time – if the same questions are uncritically asked in the same context, testing efforts will lose their novelty and their quality will stagnate. By changing focus and trying new software testing techniques from a different point of view, the savvy tester will generate new questions to be answered.

7. CONCLUSION

As mentioned earlier, knowledge breeds quality in a testing context. The arguments in this paper have shown, through parallels between the SDLC and the MDLC, that interdisciplinary approaches can add not only value but perspective to the tester's role in assuring quality. By gaining awareness of method and effect through the study of the five pillars of magic, the tester is given new ways to think both deductively and critically about what they observe and perceive. Testing is full of questions and as long as the tester remains both open minded and curious, as is the magician's ideal audience, the field of software testing will remain novel and fun. However, without awareness of the realities in which we the observers conduct our observations, we as the tester are no better than laymen naively watching magic tricks for the first time. That said, there is a rich body of testing knowledge available today as this field has been in existence for over twenty years. By engaging with this existing knowledge base and integrating experience from the field of magic, as well as constantly evolving our own practices, we can avoid being deceived by our own senses and thereby become "magicians of software testing."

8. REFERENCES

[1] J. Bach, M. Bolton "Rapid Software Testing V2.1.3," Satisfice Inc, pp. 47-48, 2007.

[2] B. Pettichord "Agile Testing: What is it? Can it Work?" http://www.io.com/~wazmo/papers/agile_testing_20021015.pdf, 2002.

[3] J. W. Cooley, "Mediation Magic: Its Use and Abuse," University of Chicago Law School Journal, pp. 34, 1997.

[4] C. Kaner, J. Bach, B. Pettichord, "*Lessons Learned in Software Testing: A Context-Driven Approach*." Wiley, 4. ISBN 0-471-08112-4. pp. 11-13, 2001.

[5] T. Stafford, M. Webb "Mind Hacks, Tips & Tricks for Using Your Brain" O'Reilly, preface, 2005.

[6] C. Kaner, J. Bach, B. Pettichord, "*Lessons Learned in Software Testing: A Context-Driven Approach*." Wiley, 4. ISBN 0-471-08112-4, pp. 24-28, 2001.

[7] C. Kaner, J. Bach, B. Pettichord, "*Lessons Learned in Software Testing: A Context-Driven Approach*." Wiley, 4. ISBN 0-471-08112-4, pp. 18, 2001.

[8] J. Bach, M. Bolton "Rapid Software Testing V2.1.3," Satisfice Inc, pp. 16, 2007.

[9] T. Stafford, M. Webb "Mind Hacks, Tips & Tricks for Using Your Brain" O'Reilly, pp. 134, 2005.

[10] J. Bach, M. Bolton "Rapid Software Testing V2.1.3," Satisfice Inc, pp. 45, 2007.

[11] J. W. Cooley, "Mediation Magic: Its Use and Abuse," University of Chicago Law School Journal, pp. 4, 1997.

[12] J. W. Cooley, "Mediation Magic: Its Use and Abuse," University of Chicago Law School Journal, pp. 28-30, 1997.

Session

Measuring File Systems

Morven Gentleman

Tuesday, July 15, 2008
2:45pm - 3:45pm

Room: Saint Patrick 3rd Floor

Overview

Performance testing almost always involves cross-disciplinary skills. These skills include domain knowledge to identify sensible questions, appropriate variables, instrumentation to measure them, and functional forms that they might follow; statistical knowledge to design experiments, choosing control variables and observations; and data visualization knowledge to recognize breakdowns in models, outliers, or censoring.

This talk will illustrate these ideas as they are relevant to measuring and understanding file systems, their design, and static and dynamic characteristics of the load. It will show how designers, system operators, and users can perform empirical studies to aid in implementation choices, purchasing decisions, system tuning, and operational practices.

Measuring File Systems

W. Morven Gentleman
Dalhousie University
Halifax, NS, Canada
Morven.Gentleman@dal.ca

Abstract

Performance testing almost always involves cross-disciplinary skills. Domain knowledge is required to understand what might be sensible questions to ask, what variables might be appropriate to observe, what instrumentation might be able to measure these variables, and what functional forms the relationships between variables might follow. Statistical knowledge is required to design the experiments, including what observations should be taken in terms of setting combinations of controlled variables, as well as how to analyze these observations. Data visualization knowledge is important for recognizing anomalies through breakdowns in models, such as outliers or censoring. All these disciplines play a role in interpreting the results.

This talk will illustrate these ideas as they apply to measuring file systems. There are many different file system designs and the relevant issues depend not just on the designs, but also on static and dynamic characteristics of the load. Empirical studies are called for by designers making implementation choices, but also by system operators making purchasing decisions, tuning their systems or settling on operational practices. Users too can take advantage of empirical study of the file systems they use.

Motivation

This paper examines performance measurement of file systems. It illustrates the cross-disciplinary skills useful in studying file system performance. Performance testing almost always involves cross-disciplinary skills. Domain knowledge is required to understand what questions might be sensible to ask, what variables might be appropriate to observe, what instrumentation might be able to measure these variables, and what functional forms the relationships between variables might plausibly follow. Statistical knowledge is required to design the experiments, including what observations should be taken (in terms of how many observations should be made, and more importantly what combinations of settings should be used for controlled variables), as well as how to analyze these observations. Data visualization knowledge is important for insightful display and manipulation that leads to detecting unexpected dependencies, to suggesting the functional forms of those dependencies, as well as to recognizing anomalies through breakdowns in models, such as outliers or censoring (values outside the instrumentation recording range). All these disciplines play a role in interpreting the results.

File system performance is usually of major importance in overall computer system performance. Performance measures for file systems include typical times to access a file, to read data from a file, and to write data to a file. They also include the related measures of typical rates for how many file access operations can occur per unit time, how much data can be read per unit time and how much data can be written per unit time. Rates are not simply reciprocals of time to perform an operation, because normally several operations are performed concurrently. Typical levels of concurrency are also important measures of performance. Beyond typical values for measures, the variability of these measures is also important, as well as the values of the measures in specific situations. Times for compound operations, such as a file system check, file system backup, or file system de-fragmentation are also of interest. Moreover, elapsed time is not the only resource consumed in file system operation, so file system performance also is concerned with overhead space required on disk, working space on disk for compound operations, lengths of queues, sizes in main store of critical data structures and buffers, processor cycles consumed and context switches required, jitter introduced to streaming data, and many other things.

Designers of file systems need file system performance measurements. Today there are many different variations in file system features and functionality, even for the same operating system or on the same hardware. For any of these variations there are many details of file system implementation that a designer must choose, often to optimize performance [Sun 2004][Apple 2007]. Optimal (or even just good) choices depend on load, and what is perhaps not obvious is that load cannot be well characterized by a simple scalar quantity. The type of usage engaged in at a particular site can subject its file system to very different kinds of demands [Iamnitchi 2002], and can result in files with very different characteristics such as file size, file lifetime, intensity of activity, or sequentially of access. Since file systems are rarely designed for only a single site, file system designers are interested in typical behavior over some representative category of systems, but they are also interested in the variation of performance across that category of systems, and in the difference between the performance as experienced on this category and that experienced on other seemingly similar categories. All of this means that file system designers have a vested interest in modeling different categories of load and using these load models to predict performance.

Operators of file systems also need file system measurements. Quantifying need and usage for a site can obviously assist in making choices among alternate file systems, as well as in making provisioning choices as to physical media such as SAN (Storage Area Network), RAID, or simple disk spindles. Quantified need and usage for the site can assist in choosing how much volume of storage is required, how much spare capacity will benefit performance, how much redundancy is warranted, how much bandwidth data transfer will consume and what benefit spare capacity might offer. Structural decisions such as file layout within and across devices can have significant effects on performance [Smith 1994]. Operational procedures, such as disk compaction or file system optimization, can be the

appropriate response to measured performance bottlenecks. Note, however, that operators of file systems are normally only interested in measurements of the file system sites that they operate. Measurements on other sites are of interest only for comparative purposes.

Users of file system services can also benefit by adapting their practice to what they learn from measurements on the file systems that they use. When is caching from a central server effective? How often should file caches be flushed and refreshed? How often should garbage collection and deletion of obsolete files be performed? When should older files be migrated or archived? When is processed data cheaper to regenerate than to retrieve? Is file compression a worthwhile technique? How much file sharing occurs in practice, and are the costs of supporting and managing sharing warranted? For this collection of files, is journalizing and restoring earlier versions of files something that is actually used? Are these files used atomically or do they morph? Are they accessed randomly or sequentially? Is it better to archive smaller files within a single compendium file or to represent them individually in directories? Again, such questions do not have answers that are universally true, so users of file system services care only about measurements taken on the current state of the file systems they use.

The foregoing paragraphs serve to indicate some of the challenges facing testers setting out to make and analyze measurements on file systems. The market is diverse in objectives, interests and sophistication. Although early measurements of file systems were historically made on centralized services such as university time-sharing systems [Satyanarayanan 1981] [Ousterhout 1985] [Irlam 1993] [Tanenbaum 2006], and later there have been measurements on workstations centralized by sharing a single LAN and file server [Baker 1991] [Muharemagic 1995] [Mummert 1996] [Vogels 1999] [Zhou 1999] [Douceur 1999], today measurements need to be taken for many different systems, independently owned and operated, which the tester may never personally see nor interact with. There is no single authority who can approve code changes for instrumentation or assuage concerns about security breaches.

Instrumentation

As in many situations for in progress measurement of evolving processes, there is a choice between exhaustive recording of individual events as they happen and statistical sampling over time of process state. Recording all events as they happen, often called an event trace [Ousterhout 1985][Mummert 1996][Vogels 1999] is simple to understand and ensures that rare states or rare state transitions are not missed. When events occur frequently and record nontrivial amounts of data, however, tracing can add substantial overhead that may even interfere with the activity being measured. Consider, for example, recording all I/O activity generated for a heavily used database as it is queried and updated. Recording all the read and write actions could cause even more trace I/O than the activity being traced! Moreover, when multiple processes are being traced concurrently, relating the different traces to each other may not be easy, especially when they occur on distinct computers networked in a cluster or distributed across a wide area network. Static assessments of snapshots taken as the systems evolve may be more practical and may even be easier to comprehend.

Whether tracing all events or taking snapshots of state at sampled times, choosing what to record requires not only an understanding of the performance questions to be answered, but also often a detailed understanding of the architecture and implementation of the system. It can involve interfacing to and possibly modifying code and runtime data structures, s well as persistent data structures of the file system representation on disk.

As one example, an design choice to be evaluated might be the relative merit of growing a file by allocating a contiguous or nearby disk block versus allocating a disk block arbitrarily found somewhere on the disk. (Older file system designs strenuously attempted to achieve the former, more modern design often advocates the latter because it is simpler.) The tradeoff appears straightforward – will an arbitrarily allocated disk block impose significantly more and bigger seeks? Excessive seeks not only can reduce overall throughput, they can degrade the quality of service for individual files containing time-based media such as music or video. The pattern of reads and writes within the newly grown file itself clearly matters, but is only part of the issue. There are other considerations. Instantaneous multiprogramming depth for the whole operating system also matters, because an intervening disk access request to the same disk drive serviced on behalf of another program may leave the arm remote from the carefully arranged contiguous blocks of this file. Also, how frequently are files grown anyway – published survey statistics show the vast predominance of files are only written once, when they are first created. Even if for a particular site there really is a problem with discontinuous files, is the file growth algorithm critical, or would the problem be coped with adequately by an operational procedure of running a defragmentation utility sufficiently frequently. Measurements adequate to resolve these questions involve not just file system primitives, but other parts of the OS such as the dispatcher, and possibly long-term scheduling of operational procedures as well.

As a different example, another design choice is what metadata is useful to record in the directory entry for each file. Satyanarayanan, in one of the original studies of file systems as used [Satyanarayanan 1981], points out that the decision in the TOPS-10 OS not to record the file creation date but only the date of last modification caused challenges for implanting an automatic migration scheme. Inconsistent interpretation of file creation and last file modification dates for copies created from existing files has also caused challenges. As bandwidth and device storage capacity have increased over the years, space taken for metadata is probably not usually serious, but the overhead to maintain metadata, and the question of interpretation of metadata for files exchanged with other systems that record different metadata, may be serious. The value of such metadata thus hinges on how it is used in practice, especially as we recognize how few of the files in most file system actually are used to store archival user data, compared to those used as working store, as configuration descriptions, as internal system information, etc. Unfortunately, experiments with different metadata generally imply unique customized versions of the file system and supporting tools, although some can be done by conventions for associating each file with its metadata stored in

another conventional file (an example of this was simulation of the resource fork of files in the original Mac OS when some different file system was used as a repository).

The point of these examples is to illustrate that instrumenting a system for measurements to resolve file system issues is often unique to the issue being investigated and to the file system being investigated, possibly even to the specific site. In the past, when system source code was available, system support personnel produced customized test versions of systems. Today that is generally impractical, especially when source is not available. Unique test harnesses and load monitors can also be constructed with programmable debuggers or by running the system within a virtual machine environment that can be stopped and inspected from the host. Even so, custom instrumentation is expensive and difficult to get operations staff to approve for use in production, especially for remote or independently owned and operated sites. A different approach is to attempt to use the logs generated by many (including Linux, Windows Vista and Macintosh Leopard but not all other) systems when running in production. Such logs, when they exist, may even be configurable (indicating data that could be recorded but may not always have been). However, predefined system logs may not record exactly what is needed, leading to creative searches for surrogate recorded data from which data of interest can be inferred.

Yet another approach to instrumentation is to use portable assessment tools that are universal in that they can work on a wide variety of systems. An excellent example of such a tool is the CMU and Panasas Inc. *fsstat* [Gibson 2008] utility for collecting size and age distribution statistics for a file system site.

Statistics and Data Visualization

Classical statistics can contribute in many ways to empirical performance modeling, from sequential sampling to experimental design to stepwise regression to signal processing. The perspective is rarely that of formal hypothesis testing, but rather exploratory data analysis (EDA) where observations are collected and analyzed to suggest possible relationships consistent with the observations as well as to eliminate relationships unlikely to be accepted by more formal analysis. Exploratory data analysis is the essence of what is called data mining by other communities. Exploratory data analysis frequently relies on data visualization to extract subtle properties from experimental observations, to detect breakdowns in model assumptions, and to identify anomalies that violate patterns established elsewhere in the data.

Analysis of file system measurements is much like other empirical performance modeling except for one significant aspect that has become recognized over the years. Whereas the natural scale used for measurements in many areas produces observations that are roughly distributed like the normal distribution, or can conservatively be treated as if they were, measurements with respect to a variety of properties o file systems produce observations with much greater weight in the tails. File sizes are one such property; others include file lifetimes (time since file creation, time since last modification, time since last access). This has been noted many times on many different systems in many different experimental situations [Gribble 1998][Douceur 1999] [Evans 2002][Traeger 2008], often with sample sizes so large that the non-normality could even be confirmed by the Kolmogorov-Smirnov test, which is notoriously insensitive to tail behavior. How much greater weight is in the tails is controversial: is a logarithmic transformation sufficient, i.e. were the observations in the natural scale roughly distributed according to the lognormal distribution, or is the even more heavy-tailed power law family of distributions (Pareto distribution, Zipf's law, etc.) [Crovella 2000], required? Heuristic arguments have been presented as to why lognormal should have been anticipated [Gong 2001][Downey 2001].

The consequence of these distributions being so heavy-tailed is profound affecting not just the details of appropriate statistical methods, but even the terminology that should be used to describe results. In everyday English, when we want to refer to a typical or representative value of a random variable, we usually use the terms average, mean, or expected value without necessarily implying the formal definition from statistics. There was a great deal of surprise nearly thirty years ago [Satyanarayanan 1981] when it was first learned that although large files were easy to notice, the average file size was only about 12 Kbytes. The effect was even more dramatic for the median, i.e. that value for which 50% of the observations are less and 50% of the observations are greater. The median is less affected by the large values in the tails; indeed it picks out the same observation whatever monotone transformation is applied to the measurement scale. Median file size was at that time only about 2.5 Kbytes. The lesson was clear: for robustness we should only talk about, only think about, and only do calculation in terms of medians. Just about the only sensible use of mean file size is for estimating total required capacity in the unusual circumstances where number of files can be predicted. Median file sizes have grown over time as backing store has become cheaper and more plentiful, and as new applications and datatypes have become more demanding – but the effect is surprisingly small [Tanenbaum 2006].

It would be nice if these distributional issues could be finessed simply by making measurements in a log scale, such as measuring file size as the logarithm of the number of bytes in the file, or measuring file lifetimes as the logarithm of the time interval. Various statistical tests or data visualizations have been proposed in order to check whether that transformation is sufficient for a particular set of observations. Histograms [Douceur], fitted probability density functions (PDF) [Evans 2002], empirical cumulative distribution functions (CDF) [Douceur], collective cumulative distribution functions (CCDF) [Douceur], log-log Complementary CDF [Crovella 1998], log-log Limit Distribution Test [Crovella 1998] are among the plots tried. One of the most convincing data visualizations is the probability plot, sometimes called the qq plot [Chambers 1983][NIST 2008]. This is a plot of the quantiles of the empirically observed cumulative distribution versus the quantiles of the theoretical distribution. (The qth quantile is defined as that value for which a fraction q of the data is less than the value and a fraction (1-q) is greater than it.) If the observed sample comes from that theoretical distribution, the plot should be a straight line. What is particularly attractive about this data visualization is that it highlights the tails without being dominated by a few extreme observations. Nevertheless it immediately provides insight into questions such as whether

power law distributions need be assumed instead of lognormal: for this to be the case, the extreme observations would have to curve upward.

Illustrative example of Statistics and Data Visualization

To illustrate the foregoing, we look at some load characterization measurements from a home personal computer (Macintosh iMac, running MacOS 10.4). The plots use $\log_2$.

Figure 1 is the qq plot for the sizes of the 601,146 files in the root directory on that computer. The average file size observed was 89.70 Kbytes.

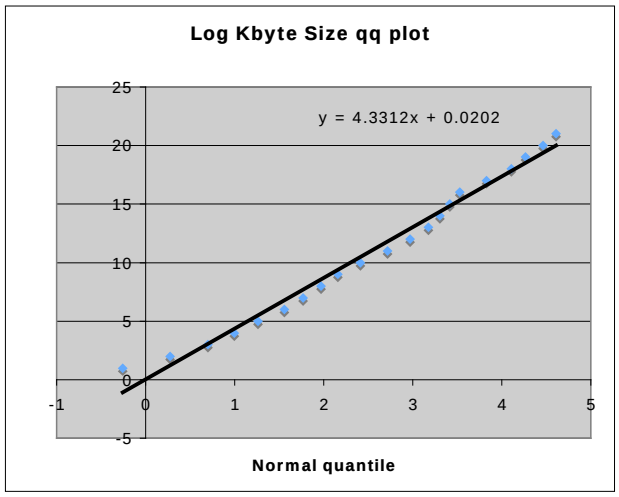


Figure 1. Probability plot for root file system of file size

We see that the qq plot is indeed linear, so there is no suggestion that a lognormal distribution for file size is not adequate. The y-intercept of the trend line, an estimator for the median, is at $2^{0.0202}$ Kbytes, that is 1.0141 Kbytes. Although the tails of log file size are normal, the center of the distribution is less so. The actual median file size is approximately 3 Kbytes. The slope of the trend line is a linear estimator for the scale factor *sigma* in the lognormal distribution, something like a Best Linear Unbiased Estimator (BLUE). The mean of the fitted lognormal, a better estimator than the observed sample mean for the population mean, is 91.88 Kbytes.

To study at the age of these files, we look at Figure 2, the qq plot for log days since file creation.

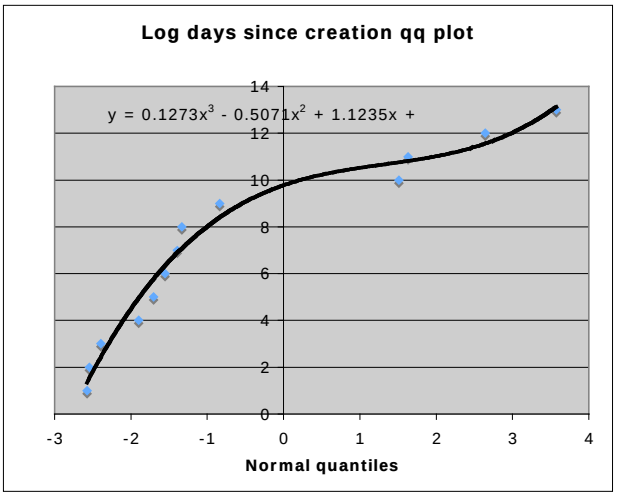


Figure 2. Probability plot for root file system of file age

In this case the qq plot is not so close to a straight line, although a piecewise linear curve might fit. We have instead used a cubic polynomial, which fits the data quite well. Again there is little reason to consider a Pareto distribution. The median age appears to be $2^{9.7783}$, that is 878.14 days.

In the system considered here, there is an additional consideration: the preponderance of files belong to, or are created by, the system itself, and are not user files. This system was initially installed 819 days before these statistics were taken, so it is not surprising that many files are of similar age.

It is interesting to consider the age of these files in terms of how recently they were changed. To do this, we look at Figure 3, the qq plot for log days since file modification.

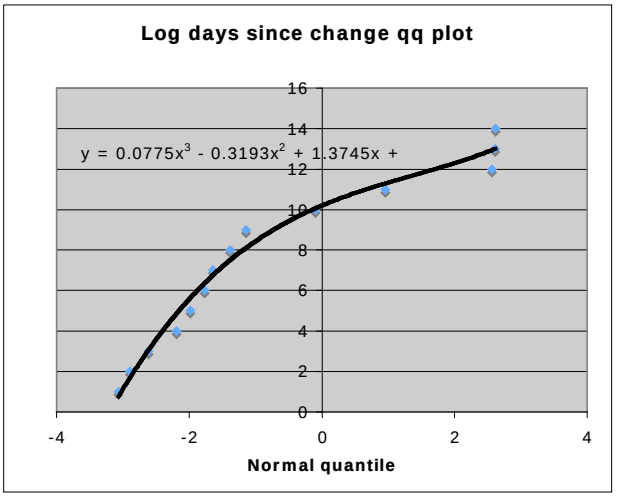


Figure 3. Probability plot for root file system of file age since last modification

As in the preceding plot, the qq plot is not so close to a straight line, although a piecewise linear curve might fit. We have instead again used a cubic polynomial, which fits the data quite well. Again there is little reason to consider a Pareto distribution. The median age appears to be $2^{10.209}$, that is 1183.63 days. It might seem odd that the median file age since

modification exceeds the median age since file creation, although not by much. The explanation apparently has to do with an inconsistent use of age fields when files are copied and not subsequently changed.

Finally, consider the age of these files in terms of how recently they were accessed. To do this, we look at Figure 4, the qq plot for log days since file access.

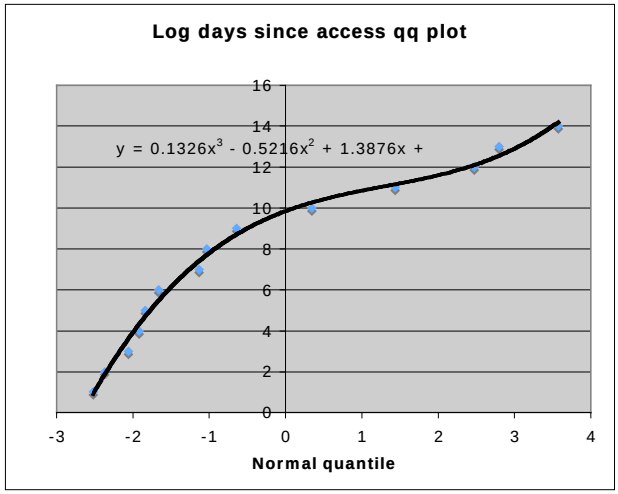


Figure 4. Probability plot for root file system of file age since last access

Once again the qq plot is not so close to a straight line, although a piecewise linear curve might fit. We have once more used a cubic polynomial, which fits the data quite well. Again there is little reason to consider a Pareto distribution. The median age appears to be $2^{9.8444}$, that is 919.31 days. This again is comparable to the age of the files since file creation.

The fact that so many files have not been written or read again since they were created has been noted on other systems. It would seem that most files are written only when they are created, and not even read after that. One corollary is that details of strategies for growing files are in practice of little importance.

In order to get some sense of how user file statistics might differ from system files, we repeat the same four data visualizations above for the 22,813 files in the home directory of the only user. (This directory contains no applications, which on other systems have been observed to be among the larger files.) The average file size observed was 620.67 Kbytes.

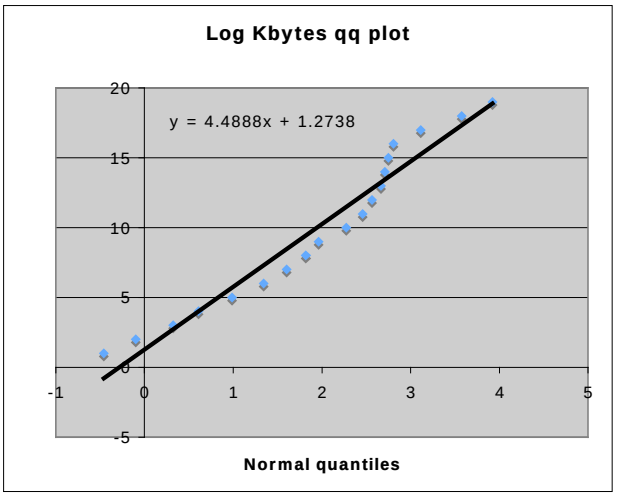


Figure 5. Probability plot for user home file system of file size

As for the root file system, Figure 5, the probability plot for the user's home directory alone, is nearly a straight line, confirming that log file size can be regarded as normally distributed. The y-intercept of $2^{1.2738}$, that is 2.4180 Kbytes, corresponds to the median predicted from the tails, although the actual median is 4.9328 Kbytes. Again, the slope of the trend line gives an estimate for the scale factor *sigma* in the lognormal distribution. The mean of the fitted lognormal, a better estimator than the observed sample mean for the population mean, is 305.92 Kbytes. This estimate being less than half the observed sample mean illustrates how the huge variance of the lognormal distribution can cause the observed sample mean to be a poor indicator of the population mean.

The median and mean file size for the home directory are somewhat larger than the corresponding file size for the root file system. Because of the large number of files in the sample, this difference is statistically significant.

Moving to the age of files in the user's home directory, we look at Figure 6, the qq plot of log days since file creation.

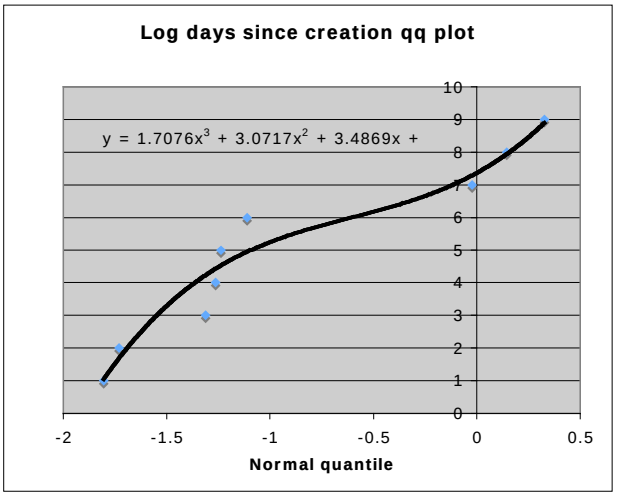


Figure 6. Probability plot for user home file system of file age

As before, the qq plot overall is not really linear although a piecewise linear curve might do. We have again fit a cubic polynomial to get the y-intercept, the fitted median. It is $2^{9.8444}$, that is 165.30 days. This is considerably younger than the median age of the files in the root file system

Moving to the age of the user's files in terms of how recently they were changed, we look at Figure 7, the qq plot for log days since last file modification.

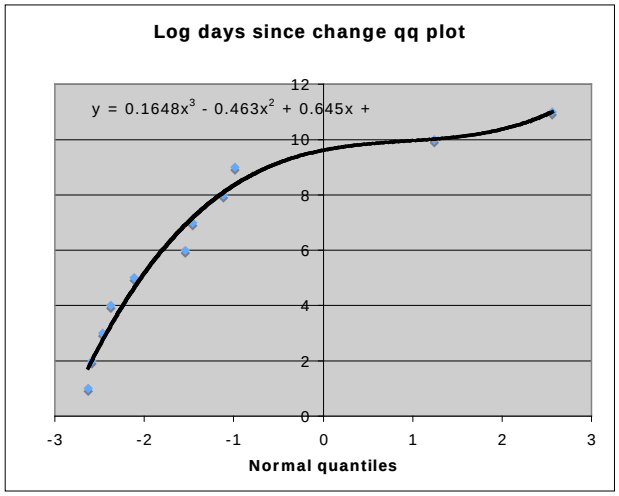


Figure 7. Probability plot for user home file system of file age since last modification

As before, the qq plot overall is not really linear although a piecewise linear curve might do. We have again fit a cubic polynomial to get the y-intercept, the fitted median. It is 29.6162, that is 784.81 days. This is considerably older than the median age since creation of the files themselves, although less than the corresponding median age since modification in the root file system. This is not entirely surprising, as many of the files were copied from earlier systems.

Finally we move to the age of the user's files since last access. To do this we look at Figure 8, the qq plot for log days since file access.

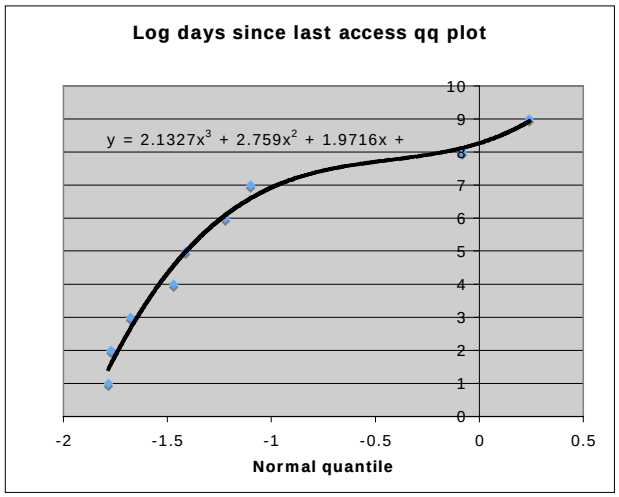


Figure 8. Probability plot for user home file system of file age since last access

As with the other file age qq plots, this plot is not really straight although a piecewise linear curve might do. Once more we fit a cubic polynomial, which fits the data quite well. The y-intercept, the fitted median, is $2^{8.2707}$, that is 308.84 days, This again is considerably older than the median age since creation of the files themselves, although less the median age since last modification of these files, and less than the corresponding median age since access in the root file system. Very odd!

Conclusions

Measurement, analysis, and interpretation of file systems is of interest to file system designers, operators and users. Testers setting out to fill this need will find challenges in instrumentation, whether custom built, culled from existing logs, or available from third parties as universal tools. Providing a valid and persuasive analysis will depend on statistical knowledge as well as data visualization. Effective visualizations often are based on deep theory. Successful interpretation often exposes things about the file system under study that you did not know or did not think mattered. Most importantly, it provides actionable advice to the client.

The illustrative example of the home computer file system has highlighted several apparent anomalies that have also been identified in previous studies.

1. The very large number of files in the system file system tree outside the user tree, that are not accessed after product installation, can be attributed in part to the propensity of operating system and application vendors to supply their products and documentation in a multitude of configurable variants, corresponding to release, processor model, and national language, along with other factors [Faas 2007]. This diversity is good, and the customer may well want to retain variants for future needs. However it is likely that on a home computer only one, or at most a few, variants will ever be used. The others do not need to be kept on-line, and certainly do not need to be kept in uncompressed form.

2. The naïve assumption that files are created, then written to, and subsequently accessed for reading or further modification leads to simplistic semantics for recorded metadata about the individual file. A great many files are copied from other files, often from other file systems. It is generally less interesting to know metadata about the instant of copying than about the origin of the content. De-fragmenting a file, moving a file to compact space in use or to optimize file access time, or compression and decompression of file content are other operations that should not modify metadata. Some system implementations have attempted to accommodate this by establishing conventions for metadata upon making a copy, but the results are not entirely satisfactory. Interpretation of metadata for copies thus needs to be done with care, and statistical summaries of file system attributes must be subject to caveat.

3. The tiny size of a typical file suggests not so much that there is little related information to be stored individually, but rather that large information complexes are represented by file structures of many related individual files.

Unfortunately the obvious mechanisms to represent such complexes in a conventional file system, such as filename conventions, subtrees, parallel subtrees, or inclusion files (where a file contains relative pathnames of other files) are crude, and complicate recognition of structured information. Similar challenges have been noted with respect to URLs on the World Wide Web.

Future Work

This paper illustrated results for a single home computer. Past surveys have studied computers in a business environment, whether central timesharing services or workstations on a LAN. There are reasons to believe that personal home computers might be used quite differently, say because of multimedia such as music or video [Evans 2002], and consequently demands on the file system could be unlike what has been observed in the past. Past studies have shown many file system characteristics are strongly affected by file type. For this reason, we plan a large-scale study of home machines.

References

Baker, Mary G., John H. Hartmann, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout, *Measurement of a Distributed File System*, Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles, pp 198-212, Pacific Grove, CA, October 1991

Chambers, John, Cleveland, William, Kleiner, Beat, Tukey, Paul, *Graphical Methods for Data Analysis*, Wadsworth, 1983

Crovella, Mark E., Taqqu, Murad S., Bestavros, Azer, *Heavy-Tailed Probability Distributions in the World Wide Web*, *A Practical Guide to Heavy Taile: Statistical Techniques and Applications*, (ed. Adler, Feldman & Taqqu), Birkhouser Boston Inc. pp. 3-25, 1998

Crovella, Mark E., *Performance Evaluation with Heavy Tailed Distributions (Extended Abstract)*, TOOLS 2000, B.R. Haverkort et al. (Eds.): LNCS 1786, pp. 1–9, 2000.

Douceur, John R., and Bolosky, William J., *A Large-Scale Study of File-System Contents*, ACM SIGMETRICS'99, Atlanta, GA, May 1-4, 1999. pp. 59-70, 1999.

Downey, Allen B., *The Structural Cause of File Size Distributions*, ACM SIGMETRICS/Performance Evaluation Review, Vol. 29, No.1, June 2001, pp. 328-329, 2001

Evans, Kyle M., and Kuenning, Geoffrey H., *A Study of Irregularities in File-Size Distributions*, Proceedings of the 2002 International Symposium on Performance Evaluation of Computer and Telecommunication Systems, San Diego CA 2002

Faas, Ryan, Take Back Your Mac's Hard Drive: The Best Ways to Reclaim Disk Space, PeachPit, http://www.peachpit.com/articles/article.aspx?p=761745&seqNum=1, 8 June 2007

Gibson, Garth, Unangst, Marc, and Dayal, Shobhit. *Static Survey of File System Statistics*, http://www.pdsi-scidac.org/fsstats/

Gong, Weibo, Liu, Yong, Misra, Vishal and Towsley, Don, *On the Tails of Web File Size Distributions*, Proceedings of 39th Allerton Conference on Communication, Control, and Computing, Oct. 3-5, 2001

Gribble, Steven D., Manku, Gurmeet Singh, Roselli, Drew, Brewer, Eric A., Gibson, Timothy J., and Miller, Ethan L., *Self-Similarity in File Systems*, SIGMETRICS 98/PERFORMANCE 98 Joint International Conference on Measurement and Modeling of Computer Systems, Madison, WI, June 1998, pages 141–150.

Iamnitchi, Adriana, and Ripeanu, Matei, *Myth and Reality: Usage Behavior in a Large Data-intensive Physics Project*. Poster at Supercomputing 2002

Irlam, Gordon, *Unix File Size Survey – 1993*, http://www.gordoni.com/ufs93.html

Muharemagic, Edin A., Mahgoub, Imadeldin O. and Milenkovic, Milan, *Analysis of File Usage In Personal Computer Environments*, Distributed and Parallel Databases, Vol. 3, pp. 315-324, 1995

Mummert, L. B. M. Satyanarayanan. *Long Term Distributed File Reference Tracing: Implementation and Experience*. Software—Practice and Experience, Vol. 26, No. 6, (June): pp. 705–736. 1996

NIST SEMATECH *e-Handbook of Statistical Methods*, http://www.itl.nist.gov/div898/handbook/, 2008

Ousterhout, John K., Da Costa, Hervé, Harrison, David, Kunze, John A., Kupfer, Mike, and Thompson, James G., *A Trace-Driven Analysis of the Unix 4.2 BSD File System*, ACM SIGOPS Operating System Review, Vol. 19 No. 5, pp. 15-24 1985

Satyanarayanan, M., *A Study of File Sizes and Functional Lifetimes*, Proceedings of the eighth ACM symposium on Operating systems principles, pp. 96-108,1981

Smith, Keith, and Seltzer, Margo, *File Layout and File System Performance*, Harvard University Computer Science Technical Report TR-35-94 1994

Tanenbaum, Andrew S., Herder, Jorrit N., Bos, Herbert, *File Size Distribution on Unix Systems – Then and Now*, ACM SIGOPS Operating Systems Review, Vol. 40, No. 1, pp. 100--104, Jan. 2006.

Traeger, Avishay, Zadok, Erez, Joukov, Nikolai, and Wright, Charles P., *A Nine Year Study of File System and Storage Benchmarking*, Technical Report FSL-07-01 (to appear April 2008)

Vogels, Werner, *File System Usage in Windows NT 4.0*, Operating Systems Review Vol 34. No. 5, pp. 93—109, Dec. 1999

Zhou, Min and Smith, Alan J., *Analysis of Personal Computer Workloads*, Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, March 24 - 28, 1999

Introduction to File-System Performance Guidelines, Apple Developer Connection

http://developer.apple.com/documentation/Performance/Conceptual/FileSystem/FileSystem.html 2007

File System Performance: The Solaris OS, UFS, Linux ext3, and ReiserFS A Technical White Paper, Sun Microsystems, August 2004

Program: Day 3 – Wednesday, July 16, 2008

Sessions	Room / Location				
Time	**Room: Colony Ballroom 2nd Floor**	**Room: Saint David 3rd Floor**	**Room: Saint Patrick 3rd Floor**	**Room: Terrace 3rd Floor**	**Room: Colony West**
7:30 AM – 8:45 AM	Continental Breakfast and Networking (Room: Colony West)				Registration and Sponsors booths
8:45 – 9:00 AM	Opening Remarks				
9:00 – 10:30 AM	**KEYNOTE - Cem Kaner, JD, PhD:** *The Value of Checklists and the Danger of Scripts*				
10:30 – 10:45 AM	Break (Room: Colony West)				
	Speaker Introductions:				
10:45 – 11:45 AM	**Steve Richardson and Adam Geras**: Seeking Data Quality	**Adam White:** *Software Testing To Improv*	*Lightning Talks*	*Vendor Presentations*	
11:45 – 1:00 PM	Lunch (Room: Colony West)				
1:00 – 2:30 PM	**KEYNOTE -Brian Fisher:** *The New Science of Visual Analytics*				
2:30 – 2:45 PM	Break (Room: Colony West)				
	Speaker Introductions:				
2:45 – 3:45 PM	**Bart Broekman:** *Testing Fuzzy Interfaces*	**Adam Goucher:** *Lessons in Team Leadership from Kids in Armor*	**Scott Barber:** *Testing Lessons From Civil Engineering*	*Vendor Presentations*	
3:45 – 4:00 PM	Break				
4:00 – 5:00 PM	AST Membership Meeting				

Keynotes

The Value of Checklists and the Danger of Scripts – Cem Kaner, JD, PhD

Exploratory testing is a general approach to testing, including all aspects of product/market research, test design, execution, troubleshooting, result reporting, etc. To see what is different about exploratory testing, contrast it with its opposite, scripted testing. In practice, most testing that people actually do probably sits in the middle, somewhere between pure exploration and perfect scripting. My bias is that most of the best testing sits a lot closer to the exploratory side of that continuum. And yet, I tell people they should use checklists to structure their work. How can that be? Aren't checklists really just abbreviated scripts? As a law student, and then as a lawyer, I relied heavily on detailed checklists and task outlines and templates for forms, but we were trained to use them as aids to critical thinking in the moment, rather than as directives to be followed. This talk considers that distinction, and how it has helped me approach many testing tasks in a way that provides structure but doesn't restrict exploration.

The New Science of Visual Analytics – Brian Fisher

Innovations in information and communication technology enable us to collect and process immense quantities of data about our physical environment and human activity. We accumulate these mass stores of information based on our belief that they will help to build understanding and inform decision-making in a wide range of areas of human interest. The resulting "data glut" has posed a huge challenge for data mining and related computational approaches. Visual analytics takes a different approach to the problem. The visual analytics approach to information system development is being explored in areas as diverse as science and medicine, design and manufacturing, and law enforcement and disaster relief.

Seeking Data Quality:
Using Agile Methods to Test a Data Warehouse

Steve Richardson and Adam Geras

Wednesday, July 16, 2008
10:45am - 11:45am

Room: Colony Ballroom 2nd Floor

Overview

The basic purpose of a data warehouse is to provide information that will help people make better choices, with the outcomes of those choices determining the value of the data warehouse. This value comes at the cost of building and maintaining the data warehouse and depends greatly on the data quality. Inspired by agile testing methods and value-based prioritization, we achieved great data accuracy at minor cost.

Seeking Data Quality: Using Agile Methods to Test a Data Warehouse

Steve Richardson
Ideaca Knowledge Services, Inc.
110-308 11 AVE SE
Calgary, AB CANADA
1 (403) 265 4332
steve.richardson@ideaca.com

Adam Geras
Ideaca Knowledge Services, Inc.
110-308 11 AVE SE
Calgary, AB CANADA
1 (403) 265 4332
adam.geras@ideaca.com

ABSTRACT
The basic purpose of a data warehouse is to provide information that will help people make better choices. The outcomes of these choices determine the value of the data warehouse. This comes at the cost of building and maintaining the data warehouse. How useful the information is depends on the data quality, this is the key aspect of a data warehouse. We have recently completed two data warehouse projects where we applied agile testing methods to achieve high levels of data quality.

It is critical to understand where the business value lies when testing a data warehouse. Based on our understanding we looked at these data warehouse attributes: data accuracy, data usability, and warehouse maintainability. Most of our focus was placed on the quality of the data; its relevance, completeness, correctness, and consistency. Yet the size of a data warehouse makes it impossible to test every record since the effort and cost would be immense. Given that we believe that effective testing of a data warehouse is a process of investigation and evaluation, we wanted to find the most costly failures given our time and resources. Inspired by agile testing methods and value-based prioritization, optimization of our work resulted in an efficient test workflow that helped us to achieve great data accuracy at minor cost.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms
Management, Documentation, Economics, Verification.

Keywords
Data warehouse, testing, test strategy, test planning, verification, validation.

1. INTRODUCTION
We have recently completed testing on two data warehouse projects. One of the basic premises of how we approach projects is that every activity is based on providing value. In the research we did preparing for these projects we found some information on the testing techniques used but little on why they were used and what value they provided. We did not think that this was enough for us to use for our testing strategy. This led to our researching where the business value lies in a data warehouse project and using that to determine our strategy.

Measuring the business value that better data provides is almost impossible. Let's say that the data helped to make a better decision, which led to some incremental increase in free cash flow for the business. To what extent did that data help in making the decision? Fortunately we did not need to estimate the benefits created by the data; we only needed to ensure that the best possible value was delivered to the customer. This is an easier question to answer.

The value of data warehouse lies in the use of the data that it provides. For the data to be used it must be of an acceptable level of quality to the customer. This level of data quality comes at the cost of testing, building, and maintaining the data warehouse. Yet the size of a data warehouse makes it impossible to test every record since the effort and cost would be immense.

We used agile testing principles to shape our test strategy. We wanted to find the most costly failures given our time and resources. Given that we believe that effective testing is a process of investigation and evaluation, we built our strategy around the guided exploration of those aspects of a data warehouse the provide business value.

The major driver of value in a data warehouse is data quality. From the risks of developing a data warehouse, to the ongoing impact on a business, data quality is the key to business value. It has been estimated that 60% of data warehouse project are seriously delayed, go over budget, or fail due to poor quality data [1]. In addition Larry English puts the business cost of poor quality data to be as high as 10 to 25 percent of revenue [2]. The goal of our testing became to provide the best quality data that we could in a way that minimized the cost of the effort. It is the best way to enhance the business value of a data warehouse.

2. THE VALUE OF A DATA WAREHOUSE
As the economy becomes increasingly organized around information, data becomes more of a strategic enterprise resource. Better data allows people to make better decisions, improve operational performance, and achieve a competitive advantage. Data warehouses are one tool to provide business people with the data that they need to make these operational and strategic decisions. These kinds of decisions can make or break a company and the quality of the data is vital.

CAST'08, July 16-18, 2008, Toronto, Ontario, Canada.

Like any product the important thing is how to provide value to the customer. A customer values a product to the extent that it solves a problem that they have. The product of a data warehouse is data that helps the customer solve a problem. Anything that does not help the customer solve this problem has no value to them, and to provide it is a waste of time and resources. The data warehouse must deliver reliable data that supports the customer in the decisions they make, if it does not it will be perceived as a failure [2].

It is critical to understand where the business value is in a data warehouse. The business value is in the data provided, not in the technology. Data provides value by reducing the risk involved in making a decision. The more accurately the data represents that state of the world, the better enabled the decision maker is to make the right choice. Not all decisions will have the same impact on the business. While high quality data should always be the goal, data that has the most business impact should receive the most attention.

The best measure of the value of this data is its use. The value of data is measured by the benefits that can be derived from it, in terms of future cash flows. The value of data is difficult if not impossible to isolate from its contribution to revenues, data is a catalyst rather than a direct source [3]. Fortunately we didn't need to estimate the benefits created by the data; we only needed to ensure that the best possible value was delivered to the customer. This is an easier question to answer. We needed to determine what the most important data value drivers were and see that they were delivered to the customer.

Based on the research of Daniel Moody and Peter Walsh [3], and with supporting evidence from the work of Rudra and Yeo [4], Neal and De Voe [5], English [2], and Loshin [6], we focused on these data value drivers:

- The more accurate the data is, the more useful it is, and therefore the more valuable it is.
- The value of data increases when combined with other data.
- The value of data increases with its use; in fact it only has value when people use it.

Providing this data comes at the cost of building and maintaining the data warehouse. The size of a data warehouse makes it impossible to test every record as the effort and cost would be immense. Cost effectiveness can't be achieved if we checked every relevant feature of every input and output in every possible way. Acceptable levels of quality must be determined. Often just knowing the accuracy of the data is enough for decision makers, who can then factor in a margin of error.

A prudent approach recognizes that some data quality problems present greater risks than others. We focused on applying limited resources to solve high risk problems instead of attempting to solve every data issue. To find the maximum number of errors in a limited amount of time we need to know what is good enough and how much testing we must do. Test planning is about decision making. The effective testing of complex products is a process of investigation and evaluation.

Based on our understanding we looked at these data warehouse attributes: data accuracy, data usability, and warehouse maintainability. Most of our focus was placed on the quality of the data; its relevance, completeness, correctness, and consistency. We felt that focusing on these characteristics of data quality provided us a way to ensure that we provided the value drivers of a data warehouse. This focus on data quality, and its delivery, gave us a way to maximize the value of the data warehouse while limiting the size of the testing effort.

3. AGILE AS BUSINESS VALUE DRIVER

Based on our assumption that the effective testing of complex products is a process of investigation and evaluation, we sought refuge in the agile principles as a means of dealing with the complexity and to give us a mechanism for guiding us towards the tests that would be of the most value to our customers.

Our goal was to enable a 'lean' test process that eliminated waste. We made a general assumption that writing test scripts and collating test data was one of the more resource-intensive activities in testing. Avoiding this effort on invalid test targets would be of paramount importance, and we would further strive to avoid this effort for low-value targets.

Consequently, our choice was to base our plan on motivated, skilled testers that would investigate the quality of the data without obsessive process. We chose to communicate the goals of the testing and provide guidance as to what needs to be looked at using lightweight scripts. Further, we chose to ask co-located business users to help us prioritize the test targets based on the data value context we provided earlier in the paper.

In agile development, the approach tends towards using business value as a guide for what to build next. In our case, we wanted to use business value as a guide for what to test next. The typical example of how an agile process drives towards delivering value is the product backlog from Scrum [7]. As a list of what to deliver placed in descending order of business value by motivated business users, the product backlog is the basis for release planning. The team chooses a number of the highest-value backlog items for an upcoming release, and further chooses from that number for an impending sprint. It seems reasonable that we can apply this same mechanism to a backlog of testing tasks. The result would presumably be a value-driven test effort, where the items that have the most value are tested first.

Other agile principles that applied directly to our context were simplicity, sustainability, face-to-face communications, and the reliance on retrospectives. Each of these principles contributed to our test effort, as outlined further by our test strategy.

4. TEST STRATEGY

Much like a project process, the optimal testing process is influenced by the product under test and the environment that it will operate in. For the purposes of understanding our test strategy, then, it seems appropriate to describe our sense of the product we were requested to test.

A data warehouse is a database that collects and integrates an organizations data in order to provide timely management information and data analysis. It is comprised of fact and dimension tables that are associated into associated sets called stars. A fact table is a collection of data elements that are usually numeric and represent some aspect of the companies operation. It usually has a multipart foreign key that is made up of its' associated dimensions. A dimension table represents one characteristic of the business that describes some aspect of a fact;

it has a single primary key that is represented in one or many fact tables.

Our test strategy focused first on the concept of a 'test target' and second on using a 'testing backlog' that was comprised of test targets. In our case, test targets were items that required testing that were at approximately the same level of abstraction – facts, dimensions, and the 'extract, transform and load' processes, or ETL's. Each target, we assumed, required approximately the same test approach. The testing backlog was therefore simply a list of the ETL's that were being built and therefore required testing. The business representatives on the team assisted us to order the targets in descending order of value (based on the earlier description of value of data above) and we then proposed test iterations that were comprised of a number of the targets.

A further assumption was that writing test scripts and devising test data to support test cases is an expensive, and arguably the most expensive, activity that testers are asked to perform in this context.

As test targets and as items on testing backlog, the ETL's were essentially the skeleton of an investigative and exploratory test effort. We started thinking of this testing as 'guided exploratory' testing since it wasn't purely exploratory – we had a guide that we could use as a monitor on our progress. We were even able to use a burn-down chart. In addition, we chose to abstract what was common in testing from across all the targets, and use that to establish a 'test script' for our testers to follow, albeit the script was not so formal as to include specific test input values and specific test output values.

With simplicity as our guide and our assumption that the testers were motivated individuals, the vague test script proved to be an excellent framework for being investigative and questioning and for documenting the test effort for the internal and external auditors. In our case, we needed to validate that we were compliant with the Sarbanes-Oxley Act and its' Canadian equivalent. The basic idea behind these government regulations is to make certain that public companies can produce reliable and repeatable results. To ensure that every step of a company's business process is documented and audited, and that all systems are in agreement and enforcing adequate internal controls. The test strategy should evolve as you continue to learn about the data warehouse, its development, and its typical faults. The contents of the test scripts evolved over the course of the project through the retrospectives that we organized for the end of the iteration.

4.1 TEST TECHNIQUES

The process to get data from the source systems to the target facts and dimension sets called stars can become complex. But our approach to the testing of that process was simple. At each step of the process where the data moved between one system and the next we provided a simple script to guide the tester. This gave us both a way to validate the quality of the data and provide an audit trail of the internal controls of the data warehouse for regulatory compliance. Originally our scripts were made up of 15 prompts for testing an ETL, dimension table, or fact table. As we went through the testing we were able to remove some of the prompts as development standards and automated checks were developed to validate those aspects of the testing.

We were in constant communication with both the development team and the end users to ensure that we were focused on the outcomes that were most relevant to the business. A user accesses the data warehouse through the set of a fact and its dimensions called a star. The dimensions are used to select data from the fact; this means that the ultimate testing target is the star. Both the data in the fact and the dimensions needs to be high quality, but so does the associations between them. It is this combination of fact and dimension that gives the data warehouse its power to provide business value.

For each star we tested the data as it flowed from source through the ETLs until it ended up in the star. At each step the data was validated. Many of the stars shared both dimension tables and ETL processes, the test results for these individual steps applied across all the stars that they were a part of. In addition we had the ETL tool capture and record as much validation information as possible. It can help with counts, totals, success, and errors. Having a dashboard to display the results of the daily loads is a useful tool both during testing and for ongoing maintenance once the data warehouse goes live.

We generally used a combination of visual inspection and SQL queries to validate that the data was correct, complete, and consistent are each transformation. We would take a cross section of data elements from the target table and compare them directly with those elements in the source system; these were checked at the field level. In addition, higher level tests were constantly comparing counts and totals between the source and target systems. Due to the sheer volume of records this type of checking should use SQL queries and software tools as much as possible.

There were three sets of scripts we used to guide the testing, one each for testing ETLs, facts, and dimensions. The scripts for ETLs were focused on the aspects of completeness, correctness, and consistency. Once all the ETLs up to a fact or dimension table had been tested we then tested that fact or dimension. Tables 1-3 show the guides we used as scripts.

Table 1 Examples of script contents used to test an ETL.

Aspect	**Guidance / Expected Results**
Completeness	Validate that no records are lost (including rejects)
…	…
Correctness	Validate that there are no data size errors
…	…
Consistency	Validate that the correct time offsets are applied
…	…
Consistency	Validate that the data is of consistent granularity
…	…

In all three tables, the purposeful absence of traditional testing step-by-step test execution instructions makes them lighter and easier to maintain and evolve. Including expected results in the 'guide' portion of the script left room for exploration, but at the same time, satisfied the Sarbanes-Oxley auditors.

Table 1 shows examples of scripts used to test an ETL. Since ETLs are used to move and transform data from a source to a

target they represent a major control point in a data warehouse. From a data quality and regulatory point of view it is essential that an ETL works correctly. Testing the ETLs in a data warehouse can be a massive task as there are typically hundreds of ETLs, moving thousands are data elements, for millions of records. To write detailed test scripts for each possible outcome is not a cost effective solution. Instead we chose to base our testing on the skills of our testers to investigate and evaluate those aspects of an ETL that provided business value and regulatory compliance. To aid in this investigation, and to provide documentation of the testing, we used a script of 15 guided prompts. These proved to be more than enough to ensure that our ETLs provided correct, consistent, and complete data throughput.

Table 2 Examples of script contents for testing a fact.

Aspect	Guidance/Expected Results
Completeness	Validate that counts and totals match
…	…
Correctness	Validate that the dimensional keys map to a record in the dimension table
…	…
Consistency	Validate that the granularity matches the specification
…	…

In a data warehouse a fact table holds the quantitative data that a business user is interested in. Table 2 shows examples from the script used to test our facts. While each fact table typically holds a unique data set we realized that to the business user the value of the fact was in the data it contained and in the ability to use this data. These aspects of a fact are consistent across all facts. Again we did not believe that developing test scripts at the level of detail where individual data elements were described for each fact was a cost effective solution. Our script for testing all facts was at a higher level of abstraction and we trusted the skill of our testers to ensure that each individual fact was tested thoroughly and achieved the best possible data quality.

Table 3 Examples script contents for testing a dimension.

Aspect	Guidance/Expected Results
Completeness	Validate that counts and totals match
…	…
Consistency	Validate that the granularity matches the specification
…	…

Table 3 provides a couple of examples from the script to test a dimension. A dimension or set of dimensions is used to access a subset of data from a fact table. This makes it imperative that a dimension table is correct, consistent, and complete. If it is not then the business user cannot access the data that they are interested in. While each dimension table represents a unique characteristic of one or more fact tables, how it is used by the business user is consistent. We believed that one script that could be applied to each dimension by skilled testers and would provide better value than developing individual detailed scripts for each dimension. The script we provided consisted of high level prompts for the type of tests that needed to be performed, leaving the details to the tester to work out. We were again seeking to achieve the highest possible data quality at the lowest possible cost. We feel that we achieved this result.

5. TEST RESULTS

As a result of our testing we were able to achieve a data accuracy rate of greater than 99.99995% with a testing effort of less than 20% of the development effort. We believe that our test strategy provided the best results that our customer could have hoped for given that it seemed further testing would have provided little or no gain in data accuracy or usability.

Our use of test scripts that provided guidance to our testers helped us achieve great data quality while minimizing the cost of testing. The common scripts allowed us to build a common understanding, common approach, and common reporting of our test effort. The use of common scripts provided a basis for communication of our results both with the development team and the business users. This helped in the further development of the data warehouse by pointing out areas for improvement, and in building trust of the data warehouse with the business users.

The focus of testing on the provision of data quality allowed us to test efficiently, rapidly, and effectively. By classifying the defects, for example, we were able to guide the team towards corrective actions that would yield the most value in terms of reducing defect rates. Root cause analysis demonstrated that the team and our customer would benefit greatly from ramping up enforcement of development standards (Table 4).

Table 4 Results of classifying the defects by root cause

Cause	Defect %
Development standards issues	23%
Implementation errors	22%
ETL errors	21%
Database issues	13%
Design issues	9%
All other issues	12%

The contents of the analysis in Table 4 is not, however, our point. Our main idea is that using a guided exploratory test strategy for our customer's data warehouse – especially when the end-users are participating in the exploration planning – worked for us, and we believe that the core test style (summarized in Table 6) is applicable to other data warehouses as well.

The contents of Table 4 are supported by the defect classification scheme outlined in Table 5. Both tables are presented as evidence that the testing yielded useful test results that had an immediate and positive impact on the customer's desire to launch the data warehouse in a timely and cost-effective manner.

Table 5 Defect root causes

Cause	Cause Breakdown
Development standards issues	Naming conventions

	Design standards
	Documentation standards
	Metadata
Implementation errors	Primary/foreign key problems
	Inconsistent field lengths
	Bad data
	Missing data
	Field types
ETL errors	Counts off
	Totals off
	Failed calculations
	Failed conversions
	Unpopulated fields
Database errors	Performance
	Indexes
	Partitions
	Tablespace
Design issues	Missing fields
	Extra fields
	Missing dimensions
	Mapping problems
All other issues	Miscellaneous

In Table 6, we have summarized the test strategy that lead to the early discovery of the root causes of problems with the target data warehouse.

Table 6 Summary - Contrasting Test Styles

Old Approach	**New Approach**
Focus on tool – database, data warehouse	Focus on value – data usage in business context
Focus on process – tables, views, stored procedures	Focus on outcome – stars/dimensions/facts
Test plans	Test backlogs
Test cases	Test targets
Detailed scripts for instructions	Light scripts as guides for exploration
No special emphasis on team communication	Team communication was vital

The focus on tools versus focus on values differentiation stems from our continued reliance on the customer's view of the data – we were less concerned about database and warehouse structure when it came time to plan the testing. Instead, we focused on how the data was to be used in daily business operations and helped our customer to create flexible and dynamic lists of things (the test targets) that should be tested (the backlog), from their perspective. This is the motivation behind much of the agile movement – getting the customer involved and strengthening their voice based on their use of system under development/test – and doing that efficiently.

6. CONCLUSION

New regulatory requirements such as Sarbanes-Oxley mean that the ability to test is now a design issue, even in systems like data warehouses. Besides the value of testing as an audit and control tool, we chose to apply value assessments of data to prioritize our tests of the data warehouse. We established a lean test process based on the agile principles of simplicity, people over process, sustainability, face-to-face communications, and retrospectives in order to minimize wasted time and effort and to maximize our ability to find more serious problems sooner.

We found that we were able to distinguish testing the data warehouse tool from testing the use of the tool. Prioritized testing backlogs of standard data warehouse elements that the user relates to were an important part of our strategy, as were limited-detail scripts that fostered guided, investigative, exploratory testing. We found that our ability to test and pinpoint serious problems was highlighted by our ability to identify development process changes that had the greatest impact on data quality. For example, there was a marked improvement in data quality after the development team implemented and enforced design and coding standards.

7. REFERENCES

[1] L. Dubois, Data Quality Supplies ROI Metrics, Business Intelligence, 2008.

[2] L.P. English, Improving Data Warehouse and Business Information Quality, John Wiley and Sons, New York, 1999.

[3] D.M.P. Walsh, Measuring the Value of Information: An Asset Valuation Approach, European Conference on Information Systems (ECIS '99), 1999.

[4] A. Rudra, and E. Yeo, Issues in User Perceptions of Data Quality and Satisfaction in Using a Data Warehouse – An Australian Experience, 33rd Hawaii International Conference on System Science, IEEE, Hawaii, 2000.

[5] K. Neal, and L.D. Voe, When Business Intelligence Equals Business Value, Business Intelligence Best Practices, 2007. www.bi-bestpractices.com/view/4744

[6] D. Loshin, Business Intelligence: The Savvy Managers Guide, Morgan Kaufmann Publishers, San Francisco, 2003.

[7] K. Schwaber, Agile Project Management with Scrum, Microsoft Press, Redmond, 2004.

Session

Software Testing To Improv

Adam White

Wednesday, July 16, 2008
10:45am - 11:45am

Room: Saint David 3rd Floor

Overview

Improvisational theatre is about exploring and extending your environment: you go with what you have in an attempt to make your team, product and company better. My experience testing on turbulent software projects is often informed by the same principles. I frequently ask myself "How do I make this the best test in the world?" "What tests will expose the best bugs?" "How am I going to explore my environment?" In this session, Adam White will explore improv concepts and give examples of how this relates to his experience as a software tester.

Session

Testing Fuzzy Interfaces –
Can We Learn From Biology And Wargaming?

Bart Broekman

Wednesday, July 16, 2008
2:45pm - 3:45pm

Room: Colony Ballroom 2nd Floor

Overview

We testers don't like fuzzy functionality—vague descriptions of situations and how the system responds—because we can't tell if we've "tested it all" or if we can "predict the result of our test cases".
This presentation explores other ways of designing test cases, besides logically deriving them from system specifications. From biology, Darwin's theory is applied to the process of designing test cases, whereby test cases evolve into better test cases. From wargaming, when acting as players of a wargame, we can find defects in situations that would remain undiscovered by 'classical' methods.

Testing Fuzzy Interfaces – Can We Learn From Biology and Wargaming?

Bart Broekman
Stationsweg 19
3603 ED Maarssen
The Netherlands
Tel. +31 (0)6 25 26 98 91
broekman.bart@gmail.com

Introduction

Usually testing involves "trying out certain situations and determine if the system behaves as expected and cannot be broken". The expected system behaviour is usually well understood and, ideally, well defined and documented in specifications.

What is meant with "fuzzy functionality"? It is about 'vague descriptions of situations and how the system responds'. The possibilities of situations that the system might encounter and the way it should react is not completely understood, let alone specified. A few examples: PDA touch screens; grey boundary areas between equivalence classes; weird interactions of functions/transactions that have never been thought of yet.

We (testers) don't like fuzzy functionality. Because it means "not deterministic". We often demand determinism of the system in order to be able to test it properly. Even more… we often demand specifications that are complete, correct and unambiguous. These we definitely loose in case of fuzzy functionality. So what is left of 'knowing you tested it all' (coverage) or 'being able to predict the result of your test cases' (oracles)?

Do other ways of designing test cases exist, besides "logically deriving them from the available information about the systems behaviour" which we learn to do in all testing workshops? This paper shows how ideas from 2 completely different worlds – the world of Biology and the world of Wargaming - might help the world of testing.

Biology

An interesting innovation in test design is the so called "Evolutionary Algorithms". It is Darwin's theory applied to the process of designing test cases, whereby test cases evolve into better test cases. Evolutionary algorithms are used for optimisation problems or problems that can be translated into an optimisation problem. A typical example is to try and find test cases for which the system violates its timing constraints. In that case test cases are 'better' when they result in higher response times. Darwin's principle of "Survival of the fittest" is the driving force behind these algorithms. Whereas test design techniques focus on individual test cases, evolutionary algorithms deal with 'populations' and the 'fitness' of individual test cases.

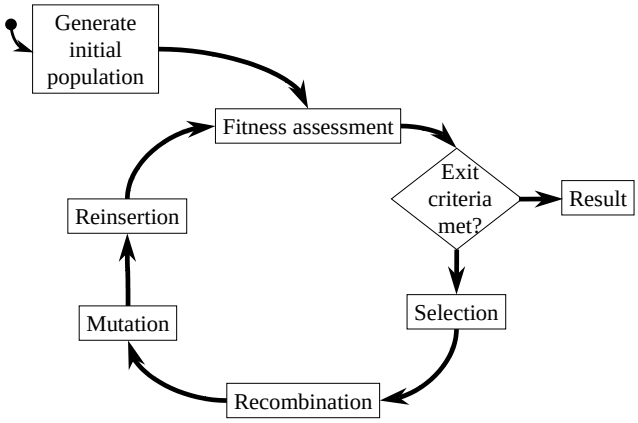
As Darwin should be praised for his brilliant theory of evolution, so should the people that saw ways to apply this theory to find new ways of designing test cases. This paper will merely aim at explaining the basics of how it works. For people interested in more solid scientific background of the theory of Evolutionary Algorithms and its applications, further reading of the following publications is suggested:

Pohlheim H. (2001). *Genetic and Evolutionary algorithm toolbox*. Chapter 1- 7.
http://www.geatbx.com/docu/algindex.html

Sthamer H. H. (1995). *The automatic generation of software test data using genetic algorithms.* Thesis at the University of Glamorgan.

Process overview

The process of evolution is illustrated in the figure below. It is a process that is usually applied to populations of animals, but as said before, it can be applied to 'populations' of test cases. The set of test cases will then go through the following steps:



1. An initial population of test cases is generated. This can be done randomly. A test case can be defined as a set of parameters, each with a value chosen from its associated domain. The initial population should not be

too small. A size between 50 and 100 is a good starting point.

2. Some test cases are 'better' than others. Why? Because they result in something closer to the desired end result. So it is the test goal that defines the concept of 'good' and 'better'. In Evolutionary Algorithms a fitness function must be constructed that adequately indicates how 'good' a test case is. For instance, when testing for violation of timing constraints, the response time itself is an adequate fitness function.

3. New test cases are constructed by combining the properties of 'parent test cases'. In practice this means that selected test cases will exchange some of their parameter values to create new combinations of parameter values, ergo – new test cases. To have a positive evolutionary drive towards better test cases, the properties of 'good' test cases should have a higher chance to be propagated than those of 'bad' test cases. But even bad test cases can possess some parameter values that turn out to be 'great' if combined with other parameter values. (It should be realized that the terms 'good' and 'bad' are related to the objective you're after. When testing to see if timing constraints are met, we try to violate those constraints, meaning that test cases that result in bad response times are actually 'good' test cases.) So the best test cases are preferred to propagate their properties, but even the bad ones should have a (slight) chance as well. Therefore this step involves random selection weighted by fitness values. How this works is further illustrated in the following intermezzo.

Intermezzo: "weighted random selection"

As an example, take 10 test cases with the following fitness values:

TC-01	12
TC-02	25
TC-03	31
TC-04	8
TC-05	12
TC-06	33
TC-07	15
TC-08	22
TC-09	6
TC-10	10

Now in order to do a random selection from this group, but in such a way that the ones with higher fitness value have a higher chance to be chosen, we can use the following mathematical trick:

Normalize the fitness values. This means "divide by the sum of all fitness values". The result is, that all normalized values will add up to 1.

Produce an "accumulated" list of fitness values. This means that for each test case its normalized value will be added to the previous accumulated value. By definition the final accumulated value will be exactly 1.

For our example these steps would result in the following:

	fitness	normalized	accumulated
TC-01	12	0,069	0,069
TC-02	25	0,144	0,213
TC-03	31	0,178	0,391
TC-04	8	0,046	0,437
TC-05	12	0,069	0,506
TC-06	33	0,190	0,695
TC-07	15	0,086	0,782
TC-08	22	0,126	0,908
TC-09	6	0,034	0,943
TC-10	10	0,057	1,000

Now if a simple random generator is used to generate a number between 0 and 1, and then pick the test case with the next higher accumulated fitness function, we get exactly what we want: random, but giving test cases with higher fitness values a better chance.

4. The step "Mutation" is necessary to enable new parameter-values to enter the population. Often this will turn out to be worse than the original test cases, but occasionally a 'winning' parameter value will enter the population.

5. Part of the parent generation of test cases will be replaced by an equally sized part of the new generation. Again this is a random selection weighted by fitness values: The new test cases with the highest fitness value have the highest chance to be selected. For the test cases that need to disappear the opposite holds: The parent test cases with the lowest fitness value have the highest chance to be selected to be discarded.

Then the process will repeat itself with the new (and hopefully improved) population of test cases. The principle of 'survival of the fittest' drives this process towards 'better' test cases, as defined by the fitness function. This continues until a certain stop criteria is met. For instance: the process continues until a test case

is found that results in a response time of more than 50 milliseconds.

This is a clear example of a test design problem where our everyday-test-design-techniques (based on deriving test cases from specifications) won't help. Because the situation we are looking for is not specified! The same holds true for the testing of fuzzy interfaces, where we 'suspect that strange things might happen'. To apply "Evolutionary Algorithms" to the testing of fuzzy interfaces, requires 1 major problem to be solved: Define a fitness function that correctly describes the goal you're after, which is "the kind of defect you try to uncover".

Wargaming

This paragraph discusses another example where test cases are NOT derived from specifications that tell you which situations should be tested. This example comes from the world of "wargaming".

Wargames are a specific kind in the vast world of games. They are usually complex games where the players command an army to battle against the army of their opponent. The army can consist of abstract counters or realistically painted miniatures, and can be historical (such as Napoleon at Waterloo) or purely fantasy (such as Elves against Orcs). For people that are interested to know more about this fascinating hobby, visiting the local game stores or browsing the internet could open up a new world.

In a certain respect playing (war)games is very similar with testing (IT-)systems. The rules of the game can be compared with the specification of the (game-)system. The games rules usually contains information that should look very familiar to us as testers, for example

- They define the different units – such as light infantry, shock cavalry, etc. - and their properties – such as movement speed and attack and defense values. This can be compared with information found in the data dictionary of an IT-system.
- They define the order of steps to play the game. Often it is a structure of "game turns", each consisting of "phases" that consist of "steps". This is very similar to the structured description of the process flow of an IT-function.
- They define the conditions under which certain actions are allowed or not. These can be very complex rules, defined in "IF…THEN…ELSE" kind of constructions.
- They define exactly what the result of certain actions is.

When playing a game, wargamers perform lots of unexpected creative moves to outwit their opponent, hoping to gain that 'edge' that will win them the game. In doing so they often end up in unexpected situations where they find problems in applying the rules. Disagreements about whether a move is allowed or not, or what the precise effect of a certain action is. Ergo they found a defect in the 'system'.

Some examples:

- From "Fire and Fury – The American Civil War in Miniature": Player A has a few units in a tight spot. A lone battery and a severely battered infantry unit are threatened by overwhelming numbers of enemy infantry. He decides to "attach" the battery to the remnants of the infantry brigade to still have a slim chance for his battery to survive. In the following turn Player B charges the unit. In the firing phase that precedes the hand-to-hand combat he wipes out the remains of the infantry, so the following melee is against the battery alone. An unlucky roll of the die gives the result "Hard Pressed", which means *"defending brigades are disordered and retreat until 2" from the enemy"*. Player A is not unhappy with this and wants to retreat his battery 2". Player B however objects and points out page 47 of the rules, saying *"When participating artillery stands are defeated and survive the combat effects, they are immediately silenced [….] They retreat their full movement rate […]"* which would put the battery out of action for at least the following turn. Player A objects to this interpretation and points to page 45 of the rule book, saying *"Attached batteries are considered as part of a brigade and are treated as any other stand of the attached brigade"* He claims that the battery was attached at the start of the turn and since it has not been detached, it still counts as attached, even though the infantry to which it was attached is gone during the actual assault.
- From "Warhammer Fantasy Battles": Magic is an important part in this game. To successfully cast a spell, the player must roll some dice and achieve a total die roll of at least the "casting value" of that spell. His opponent can then still attempt to dispel that spell. However, when the casting roll contains at least two 6's, the spell is cast with a so called "Irresistible Force" which cannot be dispelled. Player A has a nice powerful magic item called "Staff of Change" which can help spell casting a lot, but has a chance to be destroyed when being used. The effects of this item are specified as follows: *"The bearer can re-roll any number of the dice rolled to cast or dispel a spell. The second result(s) stand. […] If the bearer casts a spell with Irresistible Force using the staff's re-roll ability, the staff will exhaust its power and cease working for the rest of the battle."* Player A tries to cast a difficult spell with a casting value of 12 and he decides to roll 3 dice. He has an unlucky throw of 2 – 2 – 6. Thanks to his magic item, he re-rolls the 2 – 2 and scores a 4 – 6. So his score is now 4 – 6 – 6. Player B cries out with joy: "Hah! Irresistible Force. Your staff is out of action now". Player A is not very happy about that, but sees an escape, claiming: "I can still re-roll that original 6" and rolls a 3, resulting in a successful total of 3 – 4 – 6.Player B objects, saying that re-rolling that 6 was illegal, because the staff was already out of

action after the first re-roll. Player A disagrees, and the debate rages on.

These are examples of "fuzzy" situations where two totally different outcomes are both acceptable within the games rules (= system specifications). Clearly an unacceptable situation for us as testers! Clearly a defect of the system.

Would we as testers have found the same defects? When we regard the game rules as system specifications and then apply our test design techniques that we are so comfortable with? Probably not, because the situations that uncover the defect are not described in the specifications.

So what is it that the gamer does, that leads him to find defects that we as testers would probably miss? The critical difference between a gamer and a tester here is in the way they approach the specifications (game rules):

- For a tester the objective is to see if the system behaves as expected (or as specified). The specifications are part of his objective.
- For a gamer the objective is to WIN THE GAME. The specifications are merely the boundaries that limit his clever actions.

If we as testers want to find such defects as illustrated above, we must adopt the gamer's view on specifications. We must test the system as if to 'try and WIN THE GAME' and see the specifications as nothing more than attempts to limit our clever creative ideas to achieve this. Of course the challenge here is to find a meaningful answer to the question "when can I say that I won the game?"

(And as another intriguing afterthought… should the player of a game be considered as "the user of the system" or as "part of the system itself"?)

Summary

The concept of "fuzzy functionality" refers to vague descriptions of situations and how the system responds. To find defects related to such unspecified situations, deriving test cases from specifications obviously won't help. From 2 different 'worlds' new ideas enter our testing world that could inspire us to try new ways of designing test cases.

From Biology: Define your test goal in terms of a fitness function and use an evolutionary process to derive test cases that get better and better in achieving that goal.

From Wargaming: Translate "testing the system" into "trying to win a game" and consider the specifications as mere attempts to limit your clever moves.

Association for Software Testing

CAST 2008: Beyond the Boundaries

July 14-16, 2008 – Toronto, Canada

http://www.cast2008.org

Session

Lessons in Team Leadership from Kids in Armor

Adam Goucher

Wednesday, July 16, 2008
2:45pm - 3:45pm

Room: Saint David 3rd Floor

Overview

Groups of testers within an organization are often given the label of a team and point person is often a 'lead' or 'manager.' In sports, the person who leads the team is the 'coach'. While leading a group of testers or 4-year old lacrosse players might seem at opposite ends of the spectrum the similarities are striking.

Lessons in Team Leadership from Kids in Armor

Adam Goucher
http://adam.goucher.ca
adam@goucher.ca

1 INTRODUCTION

Groups of testers within an organization are often given the label of a team and point person is often called a 'lead' or 'manager.' In sporting, the person who leads a team is the 'coach'. While leading a group of testers or four-year old lacrosse players might seem to be at opposite ends of the spectrum the similarities are striking. This paper looks at three areas of commonality: skills development, dealing with stakeholders, and test planning

2 SKILLS DEVELOPMENT

2.1 Skill Progression Charts

The local lacrosse association [1] provides coaches with a number of Skill Progression Charts which outlines the skills players are expected to have learned and at what point. These serve as the guide for coaches in both planning practices throughout the season and as the measuring stick against which players and coaches are measured.

Table 1. Three rows of the WMLA Catching chart

	Paperweight	Tyke	Novice	Peewee	Bantam	Midget
Hand Position	I	I	R	R	R	R
Stick Position	I	I	R	R	R	R
While Moving Towards		I	I	R	R	R

I – Introduced
R- Reviewed

What this shows is that players is that Paperweight players (ages 4 and 5) only need to only worry about knowing where to put their hands and stick to catch a pass, but by Novice they should need no prompting or correction. Paperweights also do not need to worry about catching while moving towards the passer.

Testers are often affected by charts of this kind but they are presented in terms of job descriptions and definitions. These job descriptions tend to be prepared in such a manner that they could be recycled on job advertisements. Converting them to a chart however gives a different perspective and allows for easier consumption on information.

Table 2. A skill progression chart regarding test case design

	Junior	Intermediate	Senior	Lead
Test Case Execution	I	R	R	R
Test Case Creation	I	R	R	R
Test Case Analysis	I	R	R	R
Scenario Execution		I	R	R
Scenario Creation		I	R	R
Scenario Analysis		I	R	R
Session Debrief			I	R
Session Execution			I	R
Session Creation			I	R

Here we see that Junior testers are expected to know how to Create, Analyze and Execute traditional scripted tests whereas Senior ones are expected to be able to do that as well as for Scenarios and Sessions[2].

Skills Progression Charts are a powerful tool but are both fluid and highly context sensitive. Given the differences between organizations both in products or services produced and organizational culture, the charts of each organization will differ as well. If you do not use Scenarios as a test case strategy then it should not appear in the chart, or perhaps Session Execution is introduced at the Junior level with Debriefing solely the job of the Lead. The chart should also be used as a guide, not as a rule. Just because someone, according to the chart, is not supposed to be doing a task in a certain manner does not mean they should be prevented from doing so if they show an affinity for it and it is the appropriate technique for the task at hand.

2.2 Role of Coach

The role of a sports coach is all about preparing the team for success. This preparation can come in many forms including practices, workshops, targeted training, team relationship building and both positive and negative reinforcement. The role of a team lead and the tools available to them is very similar.

Part of a Coach's success is that of the individual over the team; especially in the lower house league levels. So too should the success markers of a team lead be set against the success of their members. Are the people you are leading succeeding?

- Are they discovering the type / quantity / quality of bugs expected of them?
- Are tasks taking the expected length of time?
- Are they being innovative?

If for the answer to even one of the three questions, then there is likely an underlying cause that could be met through some form of training. It could be something as formal as attending a conference, a local course or direct coaching or mentoring. The

sporting community has been using these as both learning and motivational tools for years yet it is disturbing how many testing teams let their members stale and stagnate rather than employ them. In competitive sports, coaches are removed midseason (for a number of reasons, but mainly) when their teams are underperforming. Team leads are often insulated from this form of drastic action but if they were not it is a safe bet that testers would be better trained and more engaged in their work. Both of these will lead to greater success for the team which translates to success for the lead.

2.3 Methods of Learning

Skill development in sport is largely about building up muscle memory [3]. This is because the body needs to know how to react instinctively to various situations. Successful players have practiced for hours on end both individually and as a team.

Testing software is not an activity where knowing where to move a stick to catch a ball or angle to fake out a goalie is relevant. Testing is a cerebral activity which means what leads need to help train is their team's brains. A large portion of Malcolm Gladwell's *Blink* [4] discusses training one's intuition. The traditional method of building this trained intuition in testers has been through the rote execution of prepared test plans and test cases. While this does train testers to spot the types of defects covered in the existing test cases it does not help them deal with discover different classes of defects or in some cases even new variations of a known class. A newer technique for training tester intuition deals not with testing, but about asking questions – of anything, not just software.

James Bach [5], Michael Bolton [6] and Matt Heusser [7] (to name a few) have been known to carry with them objects which they ask people to test such as a yo-yo, a salt shaker or an ice cube that glows when wet. The governing rule of this game is pretty simple: ask questions. Experienced testers are able to come up with a stream of questions which approach the problem from a number of angles (sometimes simultaneously). Newer testers might come up with a few questions initially but quickly exhaust their supply.

The sport equivalent of the traditional learning method is how kids are taught to pick up loose balls in lacrosse. First they learn to trap the ball then flip it back into the pocket. Once they have mastered that they remove the trapping aspect and just pick up the ball while moving. Advanced players can learn what is known as the Indian Pickup where they chop at the ball and pick it up using a well timed twist of the wrist. This would be akin to spending a period of time executing scripted tests, then scenarios and eventually running sessions.

With younger players variation is important for them to remain engaged with individual activities lasting no more than 10 minutes. Adult attention spans are longer but no less prone to wandering. Shutting off a tester's natural state of questioning to follow a script would then seem to be counterproductive to keeping them engaged in the task at hand. Pairing a tester learning a new technique with someone already skilled in it and letting them ask questions as they go through it would therefore have more lasting value as far as learning than just going through a series of test cases alone would.

3 DEALING WITH STAKEHOLDERS

Most of the leading definitions of what software testing is involve some mention of the people that the software interacts with. It has also been argued that testers should look more at the social sciences [8] than the hard sciences for their inspiration. Once a tester has assumed a leadership role, the interactions with other people dramatically increase. The same applies to sports; when you are a player you associate with the other players and are insulated from everything else through the coach. But when you are the coach you have to deal with many different groups of people – all with different and often conflicting agendas.

3.1 Players / Testers

3.1.1 Relating With

In both situations, the primary category of person that a coach / lead deals are those who are on their team. This paper is going to make the assumption that the members of your team want to be there. The role is then one of keeping them happy and making them effective which can be achieved by developing a good relationship with each person. Here are 4 techniques for achieving that

3.1.1.1 Open Door Policy

So often do people profess to have an Open Door Policy but then have their door closed most of the time that it phrase itself is now cliché. But approachability is critical to good leadership. The trick is however to not wait for someone to come through the door, instead, bring the door to the players who might not come talk to you for some other reason (power distance for example)

3.1.1.2 Ask Questions

Do not blindly take on faith than your testers are not stuck on something, are being as productive as you would like, or do not have some brilliant solution to a problem. The only way to find this out is by asking questions. Remember that as a coach you are not on the floor playing the game, nor are you hip-deep in pre-release software testing as much as they once were.

3.1.1.3 Meet One-on-one

It used to be that coaches set themselves as being far superior than those on the team and people performed for them our of near fear. Nowadays coaches need to show empathy and listen to people on the teams as individuals. This individual attention is the major glue that binds the relationship. Regular one-on-one meetings [9] with everyone that you are responsible for not only helps reinforce the first two techniques but the individual attention people receive will help them feel more comfortable in the relationship.

3.1.1.4 Positive Spin

Even when dealing with someone on a negative topic, padding it with something positive can soften the blow. This is especially important if the event has occurred a number of times recently. For instance, rather than just focus on a goal that was let in, mention it along with all the shots that were blocked. Or in software testing, focus not only on the bug that made it to production, but the ones that did not.

3.1.2 Adding and Removing Members

If a coach is in a non-house league context, then they will have to deal with team try-outs. This is equivalent to team leads having responsibility for adding and removing from their teams; once again, the methods used in both are similar.

3.1.2.1 Don't Just Talk, Do

Having a potential player sit in a meeting with a coach rather than on the floor to determine their skill is unheard of and yet, interviews with testers often do not include a demonstration of testing. An effective interview includes some form of testing by the potential candidate be it in a fictitious application drawn on a whiteboard, a common application or one built explicitly for that purpose.

3.1.2.2 Diversity Is Key

A lacrosse team cannot have only attackers. It needs to have defenders, loose ball experts, face-off specialists and a goalie. A test team needs a range of specialties too. A team that is overly weighted in performance, automation or black-box testers is not going to be as effective long term than if skills were appropriately balanced.

3.1.2.3 Role Players Are Necessary

Unlike the testing market, when choosing players the available talent is often determined by their birth year. This forces a coach to select not only the couple players who will be leaders on the floor but supporting players as well; and they will know their job is to provide support. Once a test team reaches a certain size some team members will have to fall into a support role as well else you risk having too many divergent strategies around testing which could take the whole effort astray.

3.2 Product Management / Parents

The relationships that a team lead or coach has with the player's parents or company management team is often the source of most of their day-to-day grief. Unfortunately that comes with both roles and can be a distraction from achieving the desired results of the team.

First and foremost, parents want success for their children. In house league situations there are often rules to make sure everyone gets equal playing time and rotation through positions so is largely mitigated. In representative teams where the better players get more playing time and the environment is more competitive that this becomes an issue. The coach's job in this type of situation is to take the heat for their decisions and not let it become a distraction for the players.

Product Management in turn, cares about their product (or products if there team is working concurrently on different tasks). The role here for the team lead is the same: prevent the distractions coming from management from affecting the rest of the team.

3.3 Officials / Auditors

Both officials on the playing floor and corporate or regulator auditors are there to ensure that certain rules and safety measures are followed. The attitude towards these two groups that the lead establishes towards them will inevitably trickle down to the rest of the team. It is important then that they be treated with respect and professionalism even when faced with an unreasonable request of obviously incorrect call. And over the long term having a civil relationship with officials and auditors tends to pay off harkening back to the adage of 'you get more bees with honey than with vinegar.'

4 TEST PLANNING

A large part of both a coach and a lead's responsibilities revolves around planning. For a coach it is about practices and game day. For test leads it is for project and feature test planning.

The largest module of the Coaching Association of Canada's NCCP Level 1 certification program that deals with practice planning and is portable to test planning.

Before you can design an effective practice you need to be able to provide answers about the following areas [10]:

- Logistics – equipment needed / available, length of practice, availability of assistants and their experience
- Athletes – number of athletes, skills and abilities, gaps in skills among athletes, reason for involvement
- Safety Risks – equipment, human error, emergency procedures in case of an accident
- Goals – purpose of practice, team goals and short-term objectives, links to previous and future practices
- Organization – structure of session, activity choice, sequence of activities
- Delivery – how will lead position themselves, what will lead watch, teaching methods employed
- Skills – technical / tactical skills, mental skills

If you change the context from sporting to test planning you end up with the beginnings of a mission from which you can start your actual planning.

A well-structured practice has five parts

1. Introduction – Coach tells team what will happen during the practice
2. Warm-up – prepare the body for the efforts of the main part
3. Main – perform activates that will help athletes improve sport-specific abilities and fitness
4. Cool-down – low intensity activities to initiate the recovery of the body
5. Conclusion – Coach provides some comments on the practice and give athletes and opportunity to provide feedback

This structure is used by coaches at every level from house league to professional leagues. The structure also maps well to the format used in Session Based Test Management; sessions will often last 60 – 90 minutes which is about the same length as a typical practice, sessions are associated with a specific mission and session sheets can be re-executed.

A comparison between the a practice plan and a session charter shows the how each phase matches

Practice Plan	Session Sheet
Introduction	Charter
Warm-up	Session Setup
Main	Test Design and Execution

Cool-down	Session Teardown
Conclusion	Debrief

- Charter – what we are testing or what problems we are looking for
- Session Setup – configuring environment, locating materials, reading manuals
- Test Design and Execution – testing to fulfill the charter
- Session Teardown – prepare session report, deconfigure environment
- Debrief – analyze the results of the session and provide opportunity to provide feedback

SBTM is often seen as a radical approach to testing when first being introduced into an organization. Sport practices have been planned in a similar form for a long time and can serve as a useful reference model when starting to implement Sessions.

5 CONCLUSION

The area of sports coaching is far more mature than software testing in terms of available learning materials and research. Most countries have a national coaching body to assist in the development of both grassroot and elite level coaches. Many, including Canada and the UK have national certificate programs, NCCP and UKCC respectively, in general coaching while in other jurisdictions (like the US) leave training up to the individual sporting federations.

Of course no amount of theory can replication the experience of actually being a coach and the hands-on learning it provides. The common perception is, especially at the house league level, is that you need to have kids (that are on the team) to coach. After checking with the three major summer leagues (lacrosse, baseball, soccer) about any policies around non-parents as coaches they all confirmed that you do not need to be a parent to be a coach. The non-parents who are coaches do it for a variety of reasons including specialized training / experience or love of the game. It is my recommendation that 'improving skills as a test team lead' be added to that list. Not only will test team thank you, but so too will kids on your team.

6 ACKNOLWLEDGEMENTS

The author would like to thank the following people for reviews and feedback on early drafts of this paper: Dr. Jacob Slonim, Kristina Woolfson, Dr. Gregory V. Wilson, Adam White and John Gilhuly. Also the 2007 WMLA Minor Paperweight Yellow team and the 2008 WMLA Minor Paperweight Purple team for experiencing the coaching journey with me.

7 REFERENCES

[1] Whitby Minor Lacrosse Association http://www.whitbyminorlacrosse.com/
[2] Hinkson, Jim. 2001. The Art of Team Coaching. Warwick Publishing Inc.
[3] Bach, Jonathan. 2000. Session-Based Test Management. In Software Testing and Quality Engineering (November, 2000)
[4] Gladwell, Malcolm. 2005. Blink: the power of thinking without thinking. Back Bay Books.
[5] http://www.satisfice.com
[6] http://www.developsense.com
[7] http://www.xndev.com/
[8] Kaner, Cem. 2006. Software Testing as a Social Science. Presented at the Toronto Association of Systems & Software Quality. October, 2006.
[9] Rothman, Johanna, Derby, Esther. 2005. Behind Closed Doors. The Pragmatic Programmers
[10] Coaching Association of Canada. 2007. Multi-Sport Modules Reference Material - Competition – Introduction (Part A)

Session

Testing Lessons From Civil Engineering

Scott Barber

Wednesday, July 16, 2008
2:45pm - 3:45pm

Room: Saint Patrick 3rd Floor

Overview

When tasked with designing and building an aesthetically pleasing bridge, using new environmentally friendly materials, over a river that is likely to flood over the roadway of the bridge once every ten years, an engineering firm doesn't start testing when the bridge is mostly complete. They don't even start testing at the same time that they start building the bridge. Rather, they start testing as soon as someone proposes an initial design. And even though bridges do fail sometimes, they fail catastrophically significantly less often than software does.

While I agree that testing a bridge is fundamentally different from testing software, the general approach and thought process that I learned while earning a B.S. in Civil Engineering have turned out to be extremely useful to me as a software tester. What surprises me is how few software testers have been exposed to these approaches and thought processes.

Come to this session for an introduction to the ways in which Professional Engineers test their creations and what lessons we can learn and appy to our jobs as software testers that will make our testing more effective and help our developers make better applications more efficiently.

Testing Lessons From Civil Engineering: Prologue to an experience report, discussion, and (hopefully) additional research

Scott Barber
sbarber@perftestplus.com

Introduction

Engineers don't look at the world the same way that testers do. Engineers look at the world with an eye to solving problems. Testers look at the world with an eye toward finding problems to solve. This seems logical. What is less logical is the fact that engineers, and I'm talking about the kind of engineers that deal with physical objects, seem to be much more sophisticated in their testing than testers. In fact, most of what I know about testing, I learned as a civil engineering student. We didn't call most of it testing. We didn't even identify it as anything other than "You really want to get this right."

Maybe Civil Engineers test better than software testers because of the motivations to "get it right". Consider what happens when a piece of Civil Engineering, like a bridge fails:

- Huge amounts of money is lost.
- Engineers lose their jobs and their licenses to practice.
- Lots of TV news coverage.
- Innocent, unsuspecting people die.

Consider what happens when a piece of software, like a program to assist with submitting personal taxes, fails:

- Some executives probably don't get bonuses.
- Some smart people put in some overtime.
- The Government extends the tax deadline.
- Even more people use the software the next year, figuring the problem has been resolved.

I guess it's no wonder Civil Engineers go the extra mile to "get it right". Maybe it's not even appropriate (let alone cost effective) for software teams to test with the same kind of rigor as Civil Engineers. But wouldn't it at least make sense for us to take a look at how they approach this testing? Might there not be lessons there that we can apply without breaking the budget or extending the project duration?

I believe so.

Some of the principles and techniques, as I recall them from engineering school, that I've applied or adapted to software testing, which I believe have had a positive impact on my testing include:

- Prototyping
- Safety Factors
- Failure Modes
- Risk Assessment
- Independent and Collective Testing of Materials and Designs
- Experimental Design and Execution
- Thought Experiments
- Realism in Environmental and Usage Modeling
- Validation of Models
- Sub-section Isolation

I make no claim to being an expert in any of these areas as they relate to Civil Engineering. I'm 15 years removed from these topics in that context. In fact, the titles I've given these ideas predominantly come from my head. I'm certain that, from a Civil Engineering perspective, what I recall about these items is at best incomplete, and possibly, just plain wrong. Be that as it may, the influence that these items have had on my development and success as a tester are profound. I believe that makes this concept worth exploring.