A BZ Media Publication

Software Test & Performance

VOLUME 4 • ISSUE 4 • APRIL 2007 • $8.95 • www.stpmag.com

BEST PRACTICES: Test Automation

Memory Leaks in Your Java Apps? Sun Tells How to Plug Them

You Are Entering The No-Code Zone

Stress Testing for Boundaries, Limits and Tolerances

Appetize Your Apps With a GUI Bug Hunt

Grooving With the Gremlins: Tips for Embracing Complexity

Software Test & Performance
VOLUME 4 • ISSUE 4 • APRIL 2007

Contents

A BZ Media Publication

Departments

Contributors

Dedicated to helping companies succeed at building teams and high-quality software solutions, ***ROBERT SABOURIN*** and his Montreal-based consultancy, Amibug.com, count among his customers academic publisher Addison-Wesley, IT training company Logisil, McGill University and smart-card maker Gemplus.

Sabourin concludes his three-part series on boundary testing with our cover story, in which he describes how he has applied stress testing to expose boundaries in customer applications found when the limits and tolerances of a system are exercised. The article begins on page 12.

A pair of development experts from Sun Microsystems explain how memory leaks might occur in your Java applications, and share their specific tools and techniques for finding and eliminating them.

Technical evangelist on the NetBeans project, Sun's ***GREGG SPORAR*** (shown) has been a software developer for more than 20 years and has used Java since 1998. Eleven-year veteran developer ***A. SUNDARARAJAN*** is a member of Sun's Java core technologies team (part of the JDK development team). The first article in this two-part series begins on page 22.

In "Learning to Groove With the Gremlins," ***DR. LINDA J. BURRS*** takes a whimsical look at the chaos that exists all around us and offers serious advice on how to cope. Gremlins start on page 30.

With multiple degrees in organizational leadership and management, Dr. Burrs has spent more than 25 years bringing her dynamic approach to coaching, training and team building to corporations and professionals. Clients of her Step Up to Success! consulting firm include law firms, technology organizations, educators, business professionals, leadership groups and nonprofits.

DAN RUBEL (left) is chief technology officer at Instantiations, and has been designing object-oriented software since 1987. He and ***PHIL QUITSLUND***, architect of the company's Window Tester GUI test automation tool, provide techniques for GUI test mechanization specifically for several popular commercial and open-source tools beginning on page 35.

Both men are considered experts in their respective fields. Rubel was instrumental in the design and development of several successful coding products and frameworks, and co-authored "Eclipse: Building Commercial-Quality Plug-ins" (Addison-Wesley, 2004). Quitslund has been active in the Eclipse research community since 2002 and has built numerous tools and extensions.

TO CONTACT AN AUTHOR, please send e-mail to feedback@bzmedia.com.

Using Stress Tests to Find Behavioral Boundaries, Limits and Tolerances

By Rob Sabourin

As a child, I broke a lot of toys. My parents warned me that if I kept doing that, I wouldn't have much fun, but I persisted. And guess what? My parents were wrong!

My favorite destructive experiences focused on a model electric train set. I'd run the train and try to see how fast it would go. I'd rearrange tracks to see how long the train would run without jumping off. This was great fun. I changed the train configuration—box cars before flat cars followed by passenger cars, and of course the caboose near the front instead of the back. I changed the track configurations, with different long, straight stretches followed by twists and curves with the occasional figure eight.

I eventually had the insight to see what would happen when I changed the track layout from a two-dimensional pattern into a system with hills and valleys, slopes and gradients. I also changed the train speed: slow when it should be fast; fast when it should be slow.

My experiments in train configuration led to some interesting experiences. It was fun to find out that the train would run just fine until I crossed a threshold with one of my variables. This occurred only in the beginning. Then I realized that I could have more than one train on the same track system at the same time. I could create dramatic situations by varying the parameters of both trains in the same environment to try to avoid or precipitate collisions.

I now realize that my early experiments in determining the boundaries of a train system were the precursor to my later hunt for boundaries in the behavior of

Rob Sabourin is president of a software consultancy, Amibug.com, in Quebec.


NO CODE
BEYOND
THIS
POINT

complex software systems. Though my aim was scientific, I used a testing approach that ended up breaking a lot of trains and track—much to my parents' chagrin.

I find traditional domain-testing approaches, such as equivalence partitioning, useful when finding boundaries related to processing, computations, data and reporting variables. Several techniques are available to help us discover those factors using both analytic and exploratory approaches. But I've had to apply very different testing approaches to identify boundaries related to a system's behavior.

Software engineering projects have many sources of bugs. For each source we can discover different boundaries. I use techniques associated with stress testing experiments to help me understand the boundaries of system behavior. If I can identify these boundaries, I can help my clients get a better feel for the limitations of systems before they're deployed.

Stress Testing Experiments

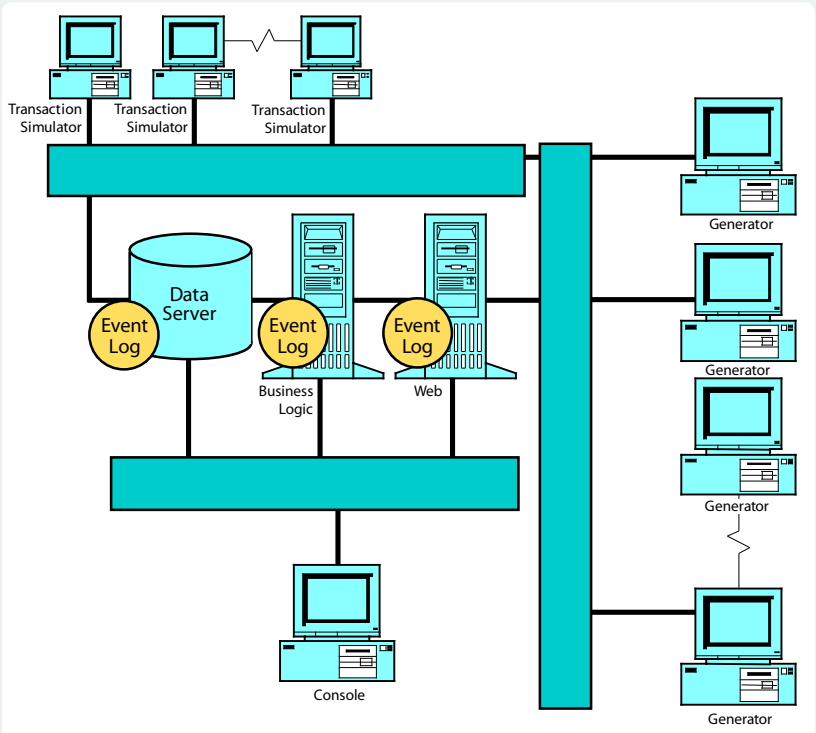
To find the limitations of a software system, I often use stress testing experiments, which explore system behavior by varying load and environmental conditions while allowing observation of system characteristics and parameters.

To illustrate the concept of stress testing experiments, I'll relate a relevant example from a recent project I completed. For a critical series of testing experiments to determine limitations of system behavior, I studied elements of the system's behavior while varying the load in a controlled test environment. I was concerned with how the system used processing capacity and available memory resources as well as the time to complete processing of typical transactions.

TABLE 1: FENCED IN

Time	Received	Processed	Queued
1	100	80	20
2	200	160	40
3	300	240	60
4	400	320	80
5	500	400	100
6	600	480	120
7	700	560	140
8	800	640	160
9	900	720	180
10	1,000	800	200

FIG. 1: TEST COMPOUND



The system being tested was a centralized examination server used to control interactive sessions of exams completed by delegates as part of a professional certification program. Several hundred concurrent users would run exams at a predetermined time slot from many geographically disparate locations. The exams were held in many different time zones with overlaps between start and finish times. Exam duration was about six hours, with several well-defined sections with well-known start and stop times. During the examination, transactions were generated from delegates' workstations, which were PCs running Windows XP Professional with current versions of Internet Explorer.

A client layer was implemented with a blend of Web-based HTTP and JavaScript, as well as some light dynamic objects. Every 15 seconds, a transaction was generated from the workstation to synchronize the exam, update questions and responses, and validate authentication. The servers ran a traditional multi-tier Web-based architecture using some operating system services to perform continuous data replication. Data was transparently replicated. The servers were running Windows 2000 using a blend of Microsoft and open-source Web software. Many different programming language environments were used. The data layer was implemented using a series of MS SQL applications and stored procedures. Load balancing was implemented using router-based approaches.

My customer was concerned about whether the system could handle the amount of load over the prescribed period of time. They had suffered through some bad experiences in the past owing to server failures when roughly 20 concurrent transactions occurred.

So I had to find the limits of the system in place. Could it handle the load and pattern of work expected? What were the boundaries?

I set up a series of experiments to identify these boundaries. Figure 1 illustrates the type of testing environment I used to find them.

The testing environment includes three major components: a load gen-

erator, a transaction simulator and a system console.

Load Generator

Load was generated via scripts simulating examination sessions using commercial load-testing tools. Some high-power PCs (fast CPU and plenty of RAM) were used for the testing period, each of which could simulate approximately 100 virtual EXAM sessions (each running in a separate thread).

Transaction Simulator

A workstation was reserved for the purpose of simulating typical EXAM transactions in a continuous loop as load was generated on other computers. Commercial test-automation tools were used to run scripts a number of times, keeping a detailed log of all HTTP events. Logged data used included all load times and times to accomplish transactions.

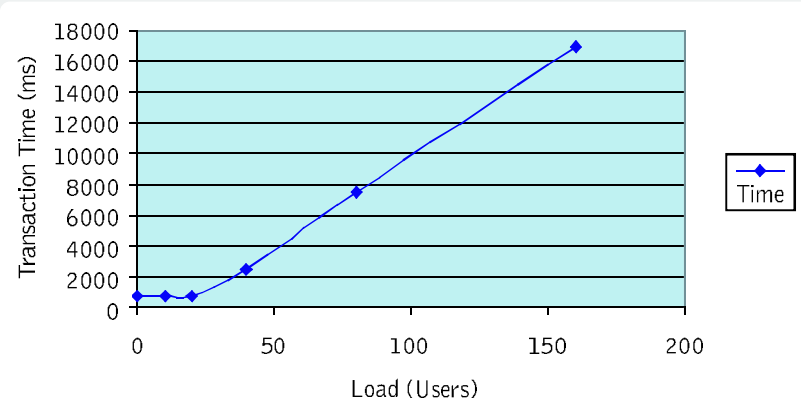
System Console

Microsoft Performance Monitor was used to monitor server performance during load generation. Sampled values included available memory and processor usage. The Web transaction IIS servers were configured to generate detailed logs of all HTTP events. The results are depicted in Figures 2, 3 and 4.

In studying the transaction time while varying load, I discovered an interesting boundary. The point of the graph at which the transaction time starts increasing is about 30 users. From one to 30 users, the time to complete a transaction is relatively constant. After 30 users, the time to process a transaction increases linearly.

The client established thresholds of acceptable system response times based on their experiences running such interactive examination sessions. The acceptable behavior of the system was set at less than 12 seconds for transactions to complete. This threshold occurred at roughly 125 concurrent users. The point on the curve at which the required threshold is passed represents a boundary at which the response is outside of the requirements. The point on the curve at which the response time changes from constant to linear is a critical boundary of system behavior.

FIG. 2: TIME RUNNING OUT



Available Memory Boundary

In the same experiment, I studied available memory resources on the server as load was varied. The boundary of memory-usage behavior change occurred at a similar point as that associated with transaction time. As shown in Figure 3 (see page 16), the available memory changes from constant to an approximate linear drop after about 25 concurrent users.

TABLE 2: MOVEMENT RESTRICTIONS

Experiment Number	Distribution Pattern	Arrival Average Transactions	CPU Capacity Available
001	Uniform Random	100 tps	90 (light)
002	Uniform Random	100 tps	60 (normal)
003	Uniform Random	100 tps	40 (heavy)
004	Uniform Random	200 tps	90 (light)
005	Uniform Random	200 tps	60 (normal)
006	Uniform Random	200 tps	40 (heavy)
007	Uniform Random	400 tps	90 (light)
008	Uniform Random	400 tps	60 (normal)
009	Uniform Random	400 tps	40 (heavy)
010	Normal Random	100 tps	90 (light)
011	Normal Random	100 tps	60 (normal)
012	Normal Random	100 tps	40 (heavy)
013	Normal Random	200 tps	90 (light)
014	Normal Random	200 tps	60 (normal)
015	Normal Random	200 tps	40 (heavy)
016	Normal Random	400 tps	90 (light)
017	Normal Random	400 tps	60 (normal)
018	Normal Random	400 tps	40 (heavy)

Processing Capacity

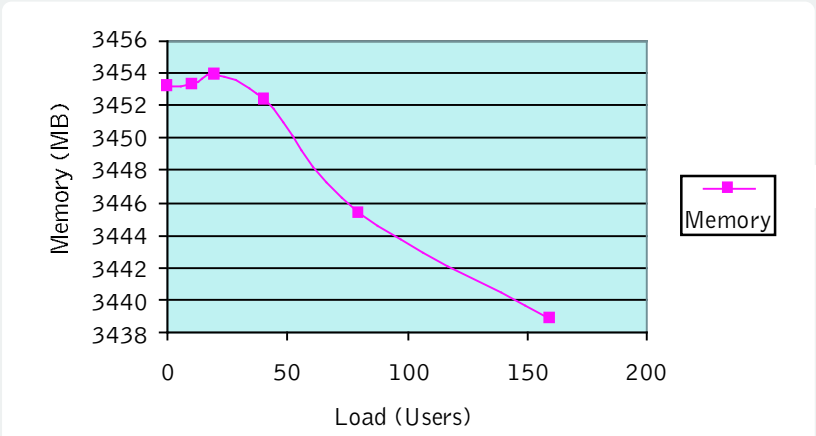
I also studied the amount of available processing capacity. I observed that while the behavior starts to become nonlinear after about 25 concurrent users, the processor did not saturate until about 80 concurrent users (see Figure 4, page 16). In my experience, it's critical to know the point at which the processor reaches saturation. If available processing resources are exhausted, transactions take longer to process and will wait longer in a queue before processing can even begin.

By performing stress testing experiments, I can find boundaries in resource usage and system behavior. I model system usage in a manner consistent with what is expected on the live system. Finding boundaries using stress testing experiments can help minimize risks of deploying underpowered solutions. Results provide a better understanding of how system resources are actually used. With these results, performance engineers can tune systems for optimal performance, and software engineers can perform what-if analysis for architecture, design and deployment alternatives.

Boundary Trends From Real Usage Scenarios

Figure 5 (see page 18) represents the processor usage of a system with a realistic simulation of several hundred users over a 24-hour period. The experiment aimed to deter-

FIG. 3: UNPLEASANT MEMORY



mine processing capacity. Load was modeled with arrival times of transactions matching those realistically expected from the client workstations during live operation. Past data and log files from operational systems were used to confirm the model's validity.

Note that the processor usage reached a peak value of 50 percent, and generally falls between 30 and 40 percent. Studying processor usage over time with a constant load helps us identify a different type of system usage boundary.

In this example, we see plenty of available CPU capacity that doesn't vary much over time. The CPU usage "trend" boundary is at about 35 percent, indicating how much of the system will be used during live operation. Some of my customers won't release systems if the processor usage exceeds 80 percent during live operation over prolonged periods of time.

Related Rate Boundaries

By blending some analysis with stress testing experiments, we can better understand boundaries related to the transaction arrival and processing behavior of systems under test.

Often I need to find boundaries related to processing in transactional systems used on mainframe and multi-tier architectures. The boundaries of concern are related to how queues fill up with pending transactions before they can be processed. I create models of how transaction events arrive and how they are subsequently processed.

If a system can process transactions faster than they arrive, the queue of pending transactions will always remain small. If transactions arrive at a rate faster than the system can handle them, we have a potential problem.

Several important boundaries can be found by looking at related rate problems: When will the process queue be full? When will database cache memory be full? When will virtual memory have to start using secondary storage?

Transactions arrive with a pattern based on the type of transaction, the timing of arrival and the distribution of transactions. Transaction processing depends on the type of transactions and what else is going on in the system while a transaction is being processed.

I like to use the tools of differential calculus to help me to find boundaries related to the behavior of the arrival and processing processes. If I can model the arrival process as some type of mathematical function and the processing as another, I can find the circumstances in which the pending transaction queue is full.

The Bucket of Water: Arrival And Processing

A related rate problem would examine the rate at which transactions go into the queue as well as the rate at which they go out of the queue. The queue is like a bucket holding water. The arrival process is like the flow of water into the bucket. System processing is like the flow of water out of the bucket. We want to find out when the bucket fills up as we vary the input (or source) functions as well as the output (or sink) functions.

Say that a(t) models the arrival process, and b(t) models the system processing. Then we can define the depth of the queue as a relationship c(t). At any time, c(t) is related to a(t) and b(t). The rate of change of c(t) is related to the rate of change of a(t) and b(t). I'm interested in discovering some characteristics of c(t). When is c(t) at its maximum value? When is c(t) at a threshold value? When does the behavior of c(t) change?

Here's a simple example. Let's say I have a queue that can store a maximum of 200 pending transactions, a uniform arrival process that delivers 100 transactions per second, and a system that can process transactions at a rate of 80 transactions per second.

Assuming that it starts empty, when does the queue become full? (See Table 1, page 14).

When viewed graphically as in Figure 6 (see page 18), the queue reaches the threshold level after 1,000 transactions have arrived and 800 have been processed. The rate of increase

FIG. 4: CRAMPED CONDITIONS

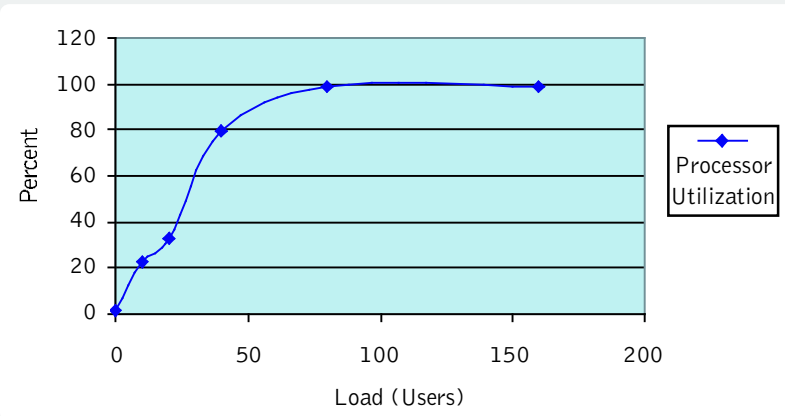
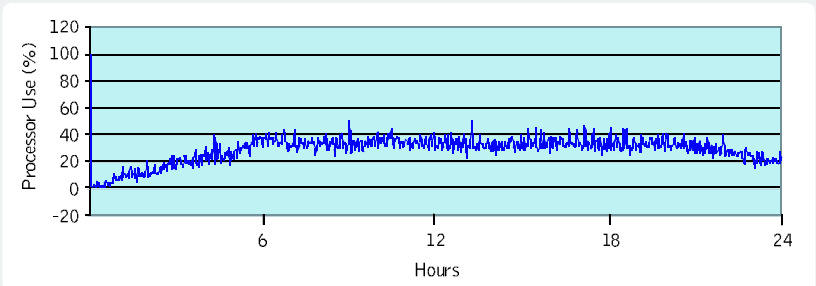


FIG. 5: ONE DAY AT A TIME



of the queue is the difference of the rate of arrival and processing: in this case, $c(t) = a(t) - b(t)$.

In real systems, arrival rates aren't simple linear equations, and processing rates depend on many factors. To determine when the queue will fill, you must study the relationships between different arrival and processing rates.

To find these boundaries, I set up stress testing experiments controlling arrival and processing rates. For each experiment, I use one transaction-generating model and one processing model.

For example, I could have three possible processing models depending on what other activities are going on in the system: a typical model, a harsh model and a light model. I could control access to system resources using other programs that consume system resources while testing takes place. I call these applications "resource hogs." I consume different processing capacity for each experiment.

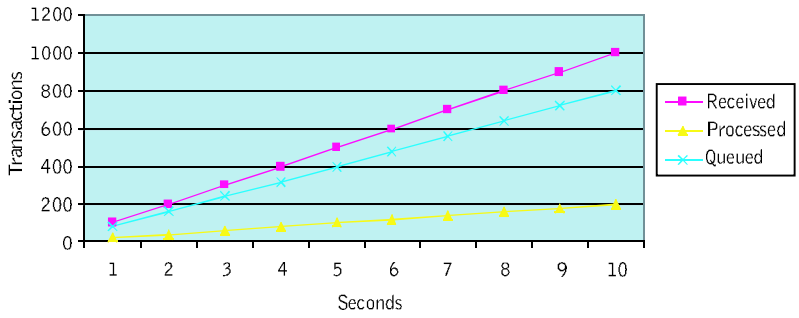
For the arrival process, I control the distribution pattern and load. I conduct a series of experiments studying transaction timing while load is applied with the target pattern. An example summary of these experiments is shown in Table 2 (see page 15).

By varying the distribution, transaction rates and available processing capacity, I can identify whether the application processing can keep up with the arrival of new transactions. I usually study transaction processing time, but I can also look at other aspects of the system, especially if I have access to performance monitor information or database log files.

Figure 7 (see page 20) plots results from a series of experiments in which each curve represents a different configuration and the variation in load changes the number of transactions generated per unit of time. Note that the behavior of each curve is slightly different, but there are two clear trends. In A, B and C, the behavior is consistent, and in E and F, the behavior is consistent.

This tells us that there is clearly a behavior boundary observed in experiments A, B and C at a load of about 50 transactions per second. At this point, the slope of the transaction-processing time curve and system behavior changes. It may be related to queues or buffers filling, or cases in which cache memory starts swapping to secondary storage.

FIG. 6: A QUEUE QUAGMIRE



The experiments E and F didn't help us identify or expose any boundaries of system behavior. If the experiments continued at a load-transaction rate of greater than 200 per second, I would probably have identified a point at which the curve slope changed.

It's critical to note that the experiment explored behavior ranges of importance to the project. During the range of interest there was no clear change in slope for E and F. Stress testing experiments don't always identify boundaries in behavior.

Surprisingly Straightforward: Boundaries in Quality Factors

I've found boundaries related to almost every aspect of software testing I've ever confronted. When the characteristics of a system are hard to quantify, boundaries are often surprisingly straightforward. Methods used to quantify quality factors are often exposed by boundaries.

Usability. For example, how can we quantify elements of usability? Usability is the quality factor related to the ability of users to achieve their goals using the software being tested. It's challenging to write requirements about usability, and even tougher to confirm that your software meets usability requirements.

In his book "Competitive Engineering" (Butterworth-Heinemann, 2005), Tom Gilb teaches a powerful approach to quantifying quality factors. The technique, which uses a tool called Planguage, allows quality factors to be described using attributes such as *scale*, *meter* and *goal*. Scale is the unit of measure, meter is the method of measurement, and goal is the required target value. By looking at quality factors from the point of view of scale, meter and goal, we can quantify the objective and establish means of testing to validate their existence or absence in the software we're testing.

Let's investigate how usability can be quantified with Planguage. Suppose we want to make sure the user interface is intuitive for our target market. Imagine a meter in which we set up a controlled environment where target users attempt to complete a transaction with the software under test. The user is given access to the system, including online help and nor-

mal documentation. While the user attempts to perform the task, we can measure the number of times that documentation or online help are referenced. We can set a goal of an average of one documentation access for 10 transactions for users of the target demographic, with suitable subject-matter expertise and experience.

With a series of controlled experiments, we can determine the trend in the number of documentation accesses per transaction to see if it is below the target. Our target is on the boundary. If the average is below the target, our software is usable. If the average is above the target, our software is not usable. Quantifying the quality factor of usability has enabled us to study boundaries in usability.

Maintainability. Another challenging quality factor is maintainability. Software systems require maintenance. Developers need to quickly adapt software, add features, change system behavior or fix bugs in an efficient manner. Planguage allows us to quantify maintainability.

For example, we could measure the time required to fix bugs during system testing. The meter would be the average time to fix bugs found in system testing. The scale would be the hours from the time the issue was triaged to the time of redelivery to the testing team. The goal could be set to an average of four hours. Now during system test, we could monitor the bug-fix time as a measure of our maintainability. If it takes too long, maintainability is below the target threshold. Goals in quality factors become the upper boundary of our testing targets.

We also can examine the problem from the opposite point of view. Sometimes the goal is not known, and we must reverse-engineer it. For example, if we don't know what the acceptable target value is, we can set up a series of experiments to determine what the actual results are and then assess whether they acceptable or not. Once we've quantified them, we can start setting measurable goals for improvement. This is akin to the old gasoline commercials that demonstrated that a car could go farther on a tank of gasoline by using a better-quality fuel. The key then and now: A meter existed, a scale was defined, and experiments determined the current value. The goal can then be established to improve the measured value.

OUNDARIES INSPIRED BY DISASTER

RESONANT FREQUENCY BOUNDARIES AND 'GALLOPING GERTIE'

Sometimes, perhaps too often, it takes a tragedy to open our eyes to a dangerous intersection or stretch of road, a deadly escalator defect, or a bridge that will not withstand the forces of nature.

One such bridge was Tacoma Narrows, which famously collapsed in 1940. Throughout its brief lifespan, locals came to call it "Galloping Gertie" because of its swaying motion whenever the wind blew. On the morning of Nov. 7, 1940, just four months after it was completed, the winds proved too much for Galloping Gertie, and it swayed and twisted until it fell apart.

What made Gertie famous was that the entire incident was captured on film by a local cameraman (http://www.youtube.com/watch?v=P0Fi1VcbpAI); the chilling images stand testament to the need for simulation testing.

Structural engineers studied the design to determine the cause. Finally, conclusive evidence was presented to demonstrate that the pattern of oscillation caused by the wind conditions hit the critical point that led to the failure. The critical boundary was not the strength of the wind, but its pattern, and the structure's resulting oscillation. The bridge was found to have a critical resonant frequency, and when the load pattern matched that frequency, the bridge structure became unstable.

In software testing, I use the notion of frequency domain boundaries to help me find system limits based on the pattern and frequency of transactions, not just the number of concurrent patterns. Stress testing experiments can be used to determine if system behavior is stable, given different distribution patterns. The load on a real system can be considered a combination of many different component loads, each operating at a different frequency. Varying the frequency patterns of the load being generated can help you understand the frequency response of the system. I can often identify boundaries in the frequency domain, and determine if there's a critical frequency in which the system behavior will change and become nonlinear.

SCALABILITY BOUNDARIES AND THE QUEBEC CITY BRIDGE

In 1907, the Quebec City Bridge collapsed during construction, taking the lives of 75 workers. In 1916, during reconstruction of a second bridge on the site, 11 workers were killed.

In 1919, a third and successful attempt of the bridge was opened. With a center span of 549 meters (1,800 feet), it remains the world's longest cantilevered bridge and is considered a major feat of engineering.

The problem with the first two attempts to build the Quebec City Bridge was a matter of scalability. How wide a span could be crossed with a cantilevered bridge? The bridge failed in 1907 and 1916 because the design was being extended beyond what was physically possible. Metal properties and construction techniques failed on wider bridges.

If there's a message to software engineers from this, it's to not assume that a project can handle increased capacity merely by adding more hardware.

To this day, members of the Canadian Council of Professional Engineers wear an iron ring that symbolically shares the materials of the Quebec City Bridge as a constant reminder of the social responsibilities inherent in civil engineering. Perhaps software testers should have the same sort of steady reminder of their responsibility to those influenced by their software.

The Wal-Mart Approach

Sometimes I'm asked to define the minimum system configurations recommended for desktop software. As a testing professional, I'd love the marketing department to give me a specific list of characteristics I could use to run a simple test. I'd set up a machine with the minimal configuration and test the application to ensure that all basic functionality worked and that transaction processing time was within reasonable tolerances. Unfortunately, my marketing departments have almost always reversed the question on me. They want to know the minimum hardware requirement of the software under test. In essence, they're asking me to find the lower boundary of acceptable systems to run the application on.

To find this boundary, I use a simple but effective technique that I call the Wal-Mart approach. I take configurations of typical desktop systems sold off the shelf at Wal-Mart for the general market over the past several years. I run a test on several sample systems to see if the application can be installed and run acceptably. If an older system fails, I try a younger one until I get to the year in which the behavior of the application is acceptable. The Wal-Mart PC configuration of the year in question becomes the resulting lower boundary threshold.

I could have experimented with processors, memory configurations and storage capacities to hunt down the optimal and minimal configurations, but the Wal-Mart strategy is cheaper and quicker. I also identify real systems that consumers actually have in their homes.

Boundaries of Credibility And Absurdity

There's one last quality factor boundary I want to share: credibility and absurdity. When we test software, we usually have insufficient time to validate many aspects. While working with these tight deadlines and limited resources, it's important to assess how reasonable and realistic our testing is.

If we focus on absurd interactions between systems and components, we'll probably expose many interesting problems, but we won't be sure if they'll show up in the real world. At the same time, we employ information about how customers will use the software to establish operational profiles and define usage scenarios to help us build meaningful test cases.

I often work with Jason DeSimone, a contract tester who's particularly gifted in bug-confirmation testing and respected by testing and development teams. Having tested software since age 15, he's often called upon to confirm that bugs have really been fixed; to confirm bugs that often he didn't find or identify in the first place.

When working on a tight deadline, you must assess how reasonable and realistic your testing is.

I tried to understand why Jason was so highly prized as a bug confirmation tester. I always suspected that he excelled because he was technically savvy and could talk with developers about all sorts of nitty-gritty technical issues. I was surprised to find out that instead, the source of Jason's success had to do with his notions of boundaries.

Here's Jason's approach to confirming fixed bugs:

1. Make sure the bug can't be repeated as described in the bug report.
2. Identify a normal usage scenario in which a user would have encountered the bug.
3. Confirm that on the baseline, bugged system, the scenario would really expose the bug.
4. Run the scenario on the fixed software to make sure the bug isn't present and to validate it hasn't just shifted.
5. Run the scenario repeatedly with variations, changing steps, sequencing, orders and values.

Jason would stop creating scenario mutations only when he felt that the scenario was so absurd that he would never be able to convince a developer to fix a bug in it. He was identifying a subjective boundary, but one that he and developers could easily agree upon. He could chat with the developers to confirm that he had crossed the line. If the bug didn't reappear, it was confirmed. If it did, he could easily convince developers that it was important enough to fix.

To Boldly Go...

In the conclusion of this three-part series, we've learned that all phases and workflows of software engineering are constrained by some sort of boundary or limitation, and that boundary testing is a critical part of the software quality-assurance experience.

With boundary testing, we "boldly go where no one has gone before"—or perhaps more accurately, where no application *should* go. As we learn the capabilities, extremes and limits of the software we're testing, boundaries are truly the final frontier!

FIG. 7: BEHAVIOR BOUNDARIES