"Rapid automated test script development in a fast paced web shop"

Clay Givens, HotJobs, a Yahoo! Company
clay_givens@yahoo.com

Documentation Presented for STARWEST 2003
San Jose, California

Table of contents

Introduction

Imagine if you are using your computer to do a transaction on the internet, and a hiccup, inefficiency, flaw, bug, hang, blue screen of death, sizzle, crash, or fatal error occurs. What do you do? If you're like me, you cuss a little, wonder how this bug made it into the software, and you try again. If it happens again, you might troubleshoot a little, and try again. If that doesn't work, you'll probably just go away, and use the other website that works. Preventing this scenario is what makes quality assurance (QA) an invaluable asset to any software development effort, and is what this paper is all about.

Practically everyone who's used some kind of machine controlled by software has seen a bug whether they knew it or not. We all have our way of dealing with them. It's a funny name to describe a software glitch, but it's applicable. Just the other day at home while working on my computer, the biggest, nastiest, looking bug crawled up the leg of my desk. Its tentacles were probing the air, its multitude of spiny appendages were gripping the vertical post. I had to do something about this disruptive creature. I don't know if you've seen a palmetto before, but it's like a giant cockroach with wings. They crawl up the pipes in your wall and come in via the little cracks in the floorboards and around pipe holes. I was angry this bug had violated my space, so I started hunting it down. It's precarious position on the post made it difficult to smash with my shoe, but that was my plan of attack. I snuck up behind it and attempted to swat it, but I missed! It jumped off the post and ran under the bed. I looked around for it a while, but wasn't able to find it. Maybe I hurt it, maybe I didn't. Either way, it disappeared, and I've yet to see it again. As a result, my confidence is lower that my room is bug free.

Sometimes, I feel like bugs in software behave like this by causing comparable problems. They take on a sort of personality all their own. They come at unsuspecting moments, they disturb your rhythm, they make you angry, and they are almost always right there under your nose hiding out. In a house, or apartment, we get exterminators to come in and take care of the bugs. This gives us peace of mind. It lets us sleep well at night. In software development, we use extermination techniques to find bugs as well. However, it's usually not as simple as spraying roach killer around the pipe holes and floor boards. We use tried and true techniques of design, documentation, development, testing, and deployment which is collectively called the software development

lifecycle. It's a big spinning wheel that starts with an idea, and ends with a product that gives a user a new tool or improved method of doing something.

In general, there are two main categories of testing which have commonly been referred to as black box, or manual testing and white box, or automated testing. Depending on who you talk to, these terms will be defined differently. To clear up any confusion, I'm referring to white box, and automated testing as testing that involves code. Debugging of existing application code, writing scripts that test other code, and using any kind of system status indicator that is not part of the actual public application, like viewing server logs, or watching the server process load via the command line. These concepts can be further stratified into the sub concepts of unit testing and functional testing. This document will focus on automated functional testing, which covers testing an application how a user would "functionally" interact with a software application by simulating a real world user interaction with the application itself. Making this distinction is important not only in the scope of this document but also in any software development effort since there are many forms of automated tests.

It's well known that automated testing will benefit software quality. However it's not a secret that automated testing can be flawed to the point where it's inefficient, or worse, ineffective and abandoned altogether. If taken seriously, it can become an effective tool for uncovering defects in the software development lifecycle. To achieve this effectiveness, it's necessary to approach automated test creation using the same methodologies used in traditional software development. Thinking of automated test creation as a "parallel development process" that co-exists with the main application under development or test will inevitably mature the test script software to a point where it will be maintainable and sustainable throughout the lifecycle of the application.

The trick is to figure out what needs to be automated, and how to approach this task. An array of questions must be considered in the course of automating tests such as:

- How much will it cost to automate tests?
- Who will do the automation?
- What tools should be used?
- How will test results be displayed?
- Who will interpret the results?
- What will script maintenance consist of?
- What will happen to the test scripts after the launch of the application?

Answering these questions before starting any automation effort is advisable for an increased chance of success. This document will attempt to explain one way of overcoming the challenges of automated testing in a fast paced web development shop. By fast paced, I mean two major releases per quarter. In order to be competitive, that's what it takes. In order to write test scripts for such an aggressive software pipeline, its imperative that solid design concepts are incorporated into the test script development.

The functional automated testing tools we primarily use are Segue software's SilkTest and Mercury Interactive's Quick Test Professional. We also use open source technologies for results display like the Apache web server, and PHP scripting language. Using these home grown tools has saved us cost on implementing the more expensive alternatives to these technologies, and the immediate availability of these tools on the internet has facilitated rapid script development. To keep up with the software development team pumping out code at a rapid pace, and in order to verify our frequent software release (builds), we have focussed on automating regression tests. These tests focus mostly on the core functions of the application, since those are the points in the application that must work. If a major component is broken after a build, we can identify it quickly, and keep progress moving forward.

Automated test result information flows freely within the quality assurance team as well as within the software engineering group as a whole. If something is identified as a failure in the automated tests, the manual testers, and the person monitoring the results of the automated tests becomes immediately aware of the problem, and can attempt to replicate it manually and report it to the appropriate engineering team via the defect tracking system. Areas of the application that are not automated are covered in detail by a team of manual testers who rely on human skill and tremendous knowledge of the application to find defects. Having both types of testers is

invaluable since they complement each other so well. Manual testers have the benefit of using the automated test results for further investigation of potential defects, and there's really no way a test effort will succeed if it's totally automated.

Dynamic requirements, dynamic test scripts

In a fast paced software development shop, the perpetual question of "how much time do we have" inevitably is answered by the compromises made between business objectives and engineering ability. Once the new feature requirements have been agreed upon, and a launch date determined, the software development starts, and the marketing and advertising groups prepare for marketing the release. Promising an unspecific set of new features to the users in an environment like this is strategically better than promising a specific set because often times features get dropped from the requirements, added to a later release, or scrapped altogether. Inevitable, on the fly re-designs or "requirements creep" occur continually throughout the build cycle almost always right up to the release date. Also, bugs found in the build cycle should be expected, and accounted for in the schedule. Extreme programming techniques allow for this, and in today's competitive internet landscape it's necessary to overcome these problems rapidly. "Extreme Programming Explained" by Kent Beck is a good place to further study extreme programming concepts.

Determining the risk posed by making changes to the application is always a factor when deciding whether or not the change is warranted. The testing time involved with that change is always a major factor in the decision. The amount of test coverage included in the automated scripts suddenly becomes extremely important. Each incremental build, last minute change, database change, and server process restart presents a golden opportunity for automated testing. Regression testing to be exact, which is defined by E.M. Bennatan in "Software Project Management", as:

> Part of the test phase of software development where, as new modules are integrated into the system and the added functionality is tested, previously tested functionality is re-tested to assure that no new module has corrupted the system.

Flexible automated regression tests can be extremely useful, not only because they cover a lot of the site quickly and report errors, but because they add confidence to the build, and diminish the need for manual testing in areas that are automated.

For example, during the build cycle, a bug is found in a major workflow, and is classified as tremendously difficult and risky to correct given the product launch deadline. The fix for this bug requires an overhauling of a core module affecting thousands of pages. To be late on the software release jostles other dependent initiatives like corresponding advertising campaigns, announcements and other marketing initiatives. The fix for the bug requires a full site regression test and major feature verification. Based on the time saved in testing, automated regression tests can be the difference between an on time launch or a delay.

Determining key areas to automate

Before determining how to automate the application, it's necessary to figure out what can and should be automated based on a host of basic parameters. It's easy to get excited about automating a certain piece of the application without really considering how this test will fit into the big picture, and how much return on investment this test will produce. Programmers want to code, just like horses want to run. For this reason, it's important to constantly be looking for opportunities to take a step back and do some analysis of the situation. The key questions that must be answered are why this module needs to be automated, what is the benefit of automating this module, how much time will it take versus how much time we have, and of course, can our budget afford this initiative. Answering these questions is based on the orientation of the application within the software development lifecycle, the available personnel to write and maintain the test script, the benefit of running the test, and the length of the test's lifetime.

Excellent candidates for automated tests are major user workflows, areas of the website that would be repetitive or tedious manual tests, and tests that are difficult to perform manually. Web forms make great tests since that's where most transactional websites interact with users. Web forms can be search engine interfaces, credit card processing mechanisms, user sign-in points, user record creation or edit methods, and just about any kind of user to computer interaction using text entry fields, or structured data displayed in list boxes, select boxes, radio buttons, or check boxes. Forms access core modules, and they can contain literally thousands of combinations of user entry given the complexity of the interface.

Once the web forms have some tests that run through them, it's possible to take the tests further to include more of the back end processing. For instance, if one web form creates a record that's supposed to appear somewhere else, it's possible to test for the down stream processes by checking for the record where it should appear. This is basic controlled testing. You know the input values, and expect certain output values. Boiled down, this is the idea behind automating a workflow. Tests that simulate user actions, and do some system verification along the way tend to be excellent gauges for showing the health of the system. Identifying these major workflows within the system is simple. What is the main purpose of the application? At HotJobs, we connect job seekers to employers. So you can imagine, there's a host of actions that comprise major workflows. Creating a resume, applying to a job, searching for a job, searching for a resume, creating a job, editing a resume, editing a job, and deleting a job or resume. Each of these actions is a test in itself, and combining them is a new type of test. Once a resume is created, it's in the database, but will it appear in a recruiter resume search? Figure out what the major workflows are and how the system can be manipulated by the available testing tools before creating the test scripts.

Designing automated tests for the web

Once the major workflows within the system have been identified, it's necessary to take it a step further, and write some documentation on what will be automated. If time permits, it's a good idea to use a standardized format like IEEE. The book, "Systematic Software Testing" by Craig and Jaskiel explains how to apply excellent documentation for websites. The point is to use some kind of system that makes sense to your application, and will shed light on what the overall testing process looks like. When it comes to the actual design of the tests themselves, I use a pseudo Unified Modeling Language (UML) that makes sense for designing automated scripts. Remember, designing automated tests is it's own software development process. Therefore, using a modular approach keeps test entities separate, yet able to communicate with each other. This is the secret to designing automated scripts. Using an object oriented approach to writing test scripts keeps them flexible and maintainable. Every time a build occurs during the develop test cycle, an automated regression test can run through the automated test cases that are based on use cases. Determining what these use cases are, and why they are good candidates for automated testing is based on the major application workflows. When test scripts are designed with a modular approach, pieces can be added and removed during runtime in order to accommodate a change or bug found in the system under test. Therefore, a great deal of forethought must accompany any full scale automated test development effort.

Designing the test script suite can take place before the system under test is ready to be tested if there's a half way reliable requirements document. If the requirements have been nailed down, and development has started, it's still not too late to do the analysis and design of the test suite. The design of the test suite should take place once the first requirements document is circulated. Then, once the system is being developed, developing automated tests should begin as well. Communication during this period is necessary for success. Developers should be attentive and responsive to automated tester's needs along the way. Resistance to this communication can kill an automated testing effort.

In general, automated tests that hit the graphical side of a web site must be developed on the site itself. Therefore, during the building of the application, automated test development can and should be happening as well. At HotJobs, the newest development takes place in a development environment. This environment is where early test script development takes place as well. When a developer gets a certain set of pages completed, such as a form. That form can be automated and wrapped up in a function. Developing functions makes it

possible to start developing automated scripts as soon as certain pieces of the application are completed. The functions can be constructed to make test cases that simulate user interactions with the system.

A common misconception is that design ends, and development begins. In practice, design stops, then starts again as new modules are added to the application or the test suite. Iterative development is a process of revisiting design and strategy throughout the build process. Building test scripts iteratively is necessary since it's impossible to foresee all aspects of the application and the accompanying test suite. Keeping a wary eye on the big picture, and always reviewing the development schedule goes hand in hand with iterative development. If a certain test slated for construction is taking too long, scrap it and move onto something else. Time is a precious commodity that cannot be wasted on overly time consuming tests. Re-evaluate, adjust, be proactive in design and development of test scripts.

Design modular tests

Thinking in terms of objects, segments, separate actions, and packages is the name of the game. The classic example of a module that can be automated and separated from other actions is authenticating, or logging in to a website. The action of logging in is generally error prone, sometimes complicated based on the specific security model in place, and can be accessed several times in a test script. Writing the log-in script over and over in the test case is a bad idea. What happens when the application moves from one environment to another and suddenly a security warning pop-up appears? Did you account for that in every one of the times you wrote the login script? Even if that segment of code was simply copied and pasted into the areas of the test case where it was needed presents a maintenance nightmare. Now you have to go back to each spot where that log-in script exists and change a line or two. Instead, think about what is really needed for logging in. A username, a password, and a set of scripted actions. A real world example is the use of secure servers, or Secure Sockets Layer (SSL). As you may already know, there are certificates that verify the integrity of a SSL transaction on the web. These certificates are distributed by vendors who maintain encryption keys which can expire. It may be that an expired certificate is being used in the development environment. When the application under test is moved to the production environment, the certificate is valid, and a pop-up security warning is not generated by the browser. As you can see, the complexity of logging in is taken up a notch when you have pop-ups coming and going.

Sometimes the browser will complain that an action is insecure. If this isn't already scripted into the login code, a login failure will occur. This isn't a "real-world" error since a user would simply click the "yeah whatever" button and go about their business. So, designing a function that can be called which will login to the certain area of the site without dying based on these little "gotcha's" is necessary. Once the login functions have been developed, an accompanying log-out function can be developed. Then, you can start building a new test case that starts with the login function, followed by an action, which could be another function like build a resume, then end the test case by calling the logout function. Then, when the login page changes in a week, you simply change the login function, and all your scripts that call that function still work. Short functions are easier to maintain, and keep the system modular. A function that takes up more than one page of code becomes hard to read, and follow. Splitting up the logic into smaller segments is easier in the long run to maintain.

Beyond functions, building flexible test plans is a good idea. An automated test plan should be able to start any test case individually without being forced to run the whole. This is usually built into the various popular vendor tools, but it's also easy to override those features and make a series of seemingly dependent test actions be coupled together. Avoid this at all costs! Look for ways to split up test modules, look for ways to de-couple, not couple. If a test is dependent on something else, the rule of thumb is to compare that coupling to the application under test's coupling. In the login function example, there's a coupling between logging in, and having a valid user account. So in your test scripts, write in verification tests that make sure a valid user is actually logged in, but don't hard code the user name and password in the login function. Make the login function accept a username and password argument that can be used for logging in. Otherwise, your function is coupled to a specific username and password.

Test data should also be modular. Remember, we are testing code that's in development. Code goes through stages of readiness, and will move from one environment to the next until it's finally launched to

production. Is the data shared in each of those areas? If not, then you must decouple test data from test environments. Dynamically loading test data at runtime that is applicable to the test environment is necessary. Of course, some data is not tied to one specific environment and can be reused without a hitch, but other data, such as URL's, usernames, passwords, and database connection names are not. Decoupling this data from the test scripts makes them portable from one environment to the next. This allows us to launch the test scripts in the production environment for testing at the same time as the application itself is launched into production. There are many ways to keep data separate from scripts. Using text files, like *.ini initialization/configuration files is simple, portable, and easy to read. Also, using a database works. But simplicity is generally a better choice here. Simply declaring global variables in a configuration file that contains certain applicable environmental variables is quick and easy.

Design end-to-end tests that hit core functionality

An end to end test simulates how more than one identified system module interacts with another based on expected cause and effect verification methods. For example, at HotJobs when a job seeker creates a resume, that's one test, but it's not an end to end test. To make it an end to end test, there must be a combination of actions to verify. Once the resume is created, the job seeker is given a choice of whether or not they want to make their resume searchable. If a resume is made searchable, then it should show up on the resume search, otherwise it should not. An end to end test would be testing for the completion of that resume creation and search process. This test hits core modules of the application without delving into the back end processes. If the resume doesn't show up in the resume search, there was a failure somewhere in the transmission of that record to its expected destination. This test now covers the back end process which handles making a resume searchable, and the test is very simple.

Choose dynamic object tests wisely, error verbosely

Automating tests of dynamic content is difficult and time consuming to get right. If a specific dynamic object or workflow must be incorporated into the test suite in order for the suite to be effective, then special attention should be spent in designing the test. If a page is dynamic, or tends to change often, yet is not part of the core application, try to avoid automating it if possible. What we are trying to avoid is reporting non-bugs since a test may fail, but the section of the application under test may not have a bug. If you consider what it takes to verify a dynamic object, or a page that changes often, it becomes apparent that you must know what the correct system response is before a test can pass or fail.

Automating dynamic objects is done by supplying test data that will output expected, verifiable results. This kind of testing gets complex quickly, and can become error prone in itself. Good modular design and verbose error trapping and reporting can help an automated tester find where a script failed, and determine whether the failure was due to a bug in the application, a bug in their script, or an unforeseen modification to the application under test.

Clean up test data

Consider a test that runs multiple times in a day. Does that test create a database record? If it does, will the creation of that record affect anything else? If it's been created on the production site, will that test data interfere with the users of the site? When designing a test case, if it's going to eventually be used in the production environment, keep in mind that test data is being created, and consider cleaning it up once it's served its purpose, or using some kind of masking technique that keeps it out of the user experience. One way I've seen data slip through the cracks and remain "public" after a test case run is due to a test case failure. In a sequential test case that creates a record, verifies a downstream process, then deletes the record, what will happen if the downstream process fails before the deletion of the record? Will the test case continue to execute through to the delete action, or will it fail and move onto the next test case leaving the garbage test data live?

A good way to avoid leaving test data public is to execute tests in do except or try catch statements. In the do or try block of code, call the function that does the action necessary to move the test case forward, such as create a record. If that function fails, then catch the error in the except or catch phrase, error verbosely, and move onto the next logical step in the event of this failure, like deleting the record if it exists.

Test using rollback and start over techniques in sequential tests

Redundancy in test case design is a great way to find patterns. The more patterns identified in the test results, the more reliable the results become. If possible, running the tests on multiple machines will help eliminate specific machine nuances, and help to isolate errors on the actual application. In a sequential test, there may be several actions that need to occur in order for the test to be a success. What happens if there are five major actions that need to happen, and sometimes for whatever reason an action fails, but not every time? Do you fail the whole test case, or do you start the failed action over and re-try? Of course that depends on your point of view in terms of test case design. If you think all test cases should pass completely without starting over, then you may miss important down stream information. One way to test is by setting an error threshold for test cases. If say, three is your magic number, then you will allow a function to fail three times before abandoning the test case sequence. If the action fails three times in a row, on all the machines running the test, you can be pretty sure there's a problem somewhere. A failure like this when compared to historical results of the same test yesterday or two days ago without a change to the test script is a dead giveaway that there's an error in the application under test.

Take into account this scenario though.... say on one machine running sequential test A, there was one failure in action 5, however it was retried and passed. On the other 3 machines running test A, there were no "hiccups" in the test results. How do you interpret that? It's possible to say "forget about it", and move on, but it's also a good investigation opportunity. If possible, go to the testing machine that had the failure, and retry the action several times manually, and see what happens, or run the failed automated test over and over on that machine. You may find a strange circumstance that only comes up every third or fifth try of a certain action. In the web world, this may mean a certain machine on a redundant array of machines may have a glitch or misconfiguration that only appears when a bad machine is hit. Load balancers used to route user traffic to available, non-stressed web servers can easily mask errors that don't exist on all the web servers. If the test occurred on a good machine when run, an error will not appear, but the bad machine may have been chosen once during the automated tests, and the error occurred. Therefore, when "glitches" are seen in automated test results, it may be due to a bad machine in the server farm that's masked by the load balancer.

When testing a workflow, there are certain actions that must pass, and some that can fail without causing the whole test case to fail. Identifying the critical actions is necessary. When a critical action fails, the script can be rolled back and tried again for the purpose of preventing a premature death of the test script. However, it's important to set logic into the test case that logically says, "stop if X happens". Using HotJobs as an example, if the login function fails, there's no way a job can be created, applied to, and verified. Therefore the rest of the sequential test should be aborted, otherwise errors will be reported that may or may not be errors due to an early failure in a dependent test. Testing this way will also speed up testing if there are major failures in the application under test since it will error on major dependent pieces and abort quicker than if it were to run through the whole suite and throw irrelevant and misleading downstream errors.

Use configuration files

Inevitably, there will be script movement from one environment to another as the application evolves. Keeping up with these movements can be easier with configuration files containing variables pertaining to script location. Configuration files can contain as many variables as necessary for the script. The main point here is that configuration files contain runtime variables that are centrally located. The variables can range from how many times a script rolls back on error, how many errors a script can have before failing, domain names, paths to applicable files, lists of data, and pretty much any other kind of variable that may be needed in more than one place, or that may need to be changed later. As a rule of thumb, if it's looking good to "hard code" a

potential variable into a script, please reconsider. If hard coded values are removed from functions and replaced with either dynamic arguments, or global variables contained in the configuration file, when it comes time to change a value it's much easier since it's not buried in a script somewhere. Burying a hard coded value somewhere in a test suite is actually dangerous since it may introduce an unforeseen bug in the script. Using configuration files for all environment variables makes script maintenance much easier and faster.

Test scheduling pitfalls

Scheduling tests allows 24-hour test coverage, or more targeted specific runs during specific times. Either way, there's a great deal of guilt felt by the automated tester when a manager asks research questions about a failed module in the application, and the automated tester has to explain that the tests failed to run on schedule, and there's no historical record of the application tests. Imagine if this happens at a time when that historical information may mean the difference between figuring out a critical bug's behavior affecting the core business application. Scheduling takes a whole new importance when put into this light, but for the most part scheduling tests allows us to go home at the end of the day, and the computer to keep testing.

Determining the test schedule

What happens when automated testers leave for a weekend, and keep their scripts running via schedule and a bug pops up that causes the script to create bunk data on the production site over and over? Will someone be available to shut down the script or clean up the data? This is potentially a bad situation and should be avoided. If someone was aware of the tests running and they see a problem, they might be able to halt the test without it causing too much of a burden to the system. Building in remote controls and remote results viewing can give testers the ability to control the tests outside the office. Also, scheduling weekend QA staff can help to resolve issues with automated tests running amuck.

Merging newly developed tests into the main test suite can introduce dependencies between the tests, the machines, or the data used. For instance, if a module is personalized, meaning a specific user has a specific set of data or area they call their own, beware of crossing wires with these system account users during test runtime. If a personalized module relies on a piece of data that's being handled by two test machines at once, a conflict of some sort can occur. Avoid scheduling these conflicts into the test suite since they may cause false alarms and wasteful log information.

If tests are being run around the clock, and a system goes down due to an un-communicated planned outage, a test script may fail. Upon analysis of the script results later, a tester may be misled by this "anomaly", and waste time researching it only to find out it was nothing. This can reduce the credibility of the test results in the future, and potentially be a reason to abandon scheduling tests. It can also shout out system errors that happen over night, and cause network maintenance to be more closely scrutinized when it's causing errors on the production site. So there are pros and cons to 24 hour 7 days a week, back to back scheduling, which boils down to how results are processed based on test design.

To stagger test suites to run at different times helps in back pedaling through results to see what was happening when. For example, if there are three identical tests setup to run on different machines, and each take three hours to run completely, it may make more sense to start one every hour instead of all of them at the same time. That way at all times there will be at least one test running, and if an error appears in an earlier test, it's possible to see if the error is still occurring an hour later, or if it was happening at the same time but in different places in the script. Information like this is useful for identifying isolated application bugs, overall network outage bugs, and for determining the duration of an issue by pinpointing it down to when it started.

Will the test always run on schedule?

Running scheduled automated tests is inherently a delicate process. Not only is the script testing a living and breathing application, its also being controlled by some kind of scheduling process that can be error prone.

If the test fails to start or if an unaccounted for obstacle such as a browser warning pop-up halts progress past a crucial point, the system health snapshot may be blurred, under exposed, or totally dark. If a crashing system, or an unaccounted for failure occurs outside the application under test causing the whole suite to fail, then it's quite possible that your camera is totally broken.

This is where analysis and design play the biggest role. Some things to consider when scheduling tests include disabling browser warnings in the web browser configuration settings. Disabling the screen saver in the operating system properties, and implementing robust shutdown and restart techniques to counteract a hanging test. Beware of overlooking a hanging test that was killed before its time since other scheduled downstream tests may not have run.

Debugging scripts takes about two thirds of the total development time. If it doesn't then the script is designed tremendously well, very short, or extremely simple. There's nothing worse than developing a new script, introducing it to the main set of scripts, and watching it fail over and over from home where you can't do anything. Test redundancy is important. Running scripts on more than one computer will increase the chances of at least one machine capturing an image of how the system is reacting to certain stimuli.

Distributing test results

How long does it take a test to run? I've heard of some organizations with runtimes of their automated suites lasting over 12 hours or more. Half a day! That's a long time for a computer to be working on testing. SilkTest and Quick Test Professional both have a nice summary of results that only appears at the end of the runtime. This didn't quite make running automated tests very efficient when you have to wait till several hours later to know whether or not an error occurred, and you have to interpret them at the machine that ran the test.

There's a general curiosity factor that happens when a test is running. When you can see it on the screen doing tasks, how can you track its progress without sitting there and watching? How can you tell if it's caught in a loop without sitting there and watching? Who has time to watch automated test scripts run? Even during development and debugging of a test script there's no information available about a test until it's completed its task. It becomes apparent that a more robust method of tracking progress and errors during runtime needs to be devised.

Some tools have database connectivity for outputting results during runtime. This is a great solution if adequate resources to maintain a database exist, and if all your testing tools can interact with that database. Databases can cause constrictions upon results, and be hard to sort through without serious planning and reporting strategies. An elegant solution to results capturing can be constructed using a database, but it also takes a little more skill and attention to detail than simply outputting text log files. Text based log files are easy to use since they can easily be parsed by scripts looking for "tokens" signifying certain meaning. They can also be read by a human line by line, which makes them useful right away.

Reporting test results via emailed summaries

We all remember the little boy who cried wolf. Don't be like that boy. Credibility is important in any business, especially in quality assurance. So how does one go about actually crying wolf in software test automation? Try a sloppy for loop, or an untested recursive function that was written late on a Friday night before the weekend that's dumping error garbage all over the logs, sending bad messages to the network monitoring system, and eventually causes your boss to get a call at 3am Sunday morning.

Reporting errors automatically can be defined in more ways than one. Simply putting a line in a log file is not reporting an error. Those get missed unless someone explicitly looks at the log file, interprets, and reports the error. Processing the log file for errors and emailing the appropriate group or individual is automatically reporting an error. Interfacing with a network monitoring system tied into a paging system that reports critical errors to human monitors is reporting errors automatically. There are certainly pros and cons to reporting errors automatically. Leading the pro charge is early defect detection, followed closely by ease of finding bugs in the application. If the scripts are stable enough, and there is time to design an elaborate paging system based on tests script failures, by all means, take advantage of paging tools. However, there is a level of tolerance

networking professionals, and quality assurance professionals have when faced with the con of interpreting a reported error that isn't really an error. These kinds of tests need constant attention.

A safe and somewhat easier solution is concocting a results summary to convey script information quickly via email. This way nobody is paged, yet the report is still available to people interested in the information. They are given a choice to follow up on information found in the nice clean reports of pass or fail statistics.

Distributing test results via the web

Have you ever left your glasses on the top of your head and looked around for them everywhere and felt silly when you put your hand on your head in frustration only to find them there? Yeah, well neither have I. That only happens to other people. As testers, we have mechanical brains that keep track of things as well or better than a database index. When we need a file from five days ago, we just hit it up instantly and regurgitate it to our boss. Yeah, right. I am forgetful. In fact I forgot my dad's birthday this year and was so upset about it! It's a pattern that I've identified. Therefore I use the computer to do the remembering for me. I can't remember if a test passed or failed three days ago when it's been run every day several times, and has failed sporadically once or twice in the last few days. The way I make sense of this is by keeping log files available on the web. All the logs since the last time I analyzed them are kept available. Once they are read and analyzed, I archive them and remove them like a read email. However, if need be, I can always go back and read through them if there's some trend analysis needed. Admittedly this is where a database would excel, but writing tests that have a line by line progress report helps to determine where the test failed. It's more granular. Appendix B shows an example of a test script log file output.

One of the biggest advantages this gives a tester is to be able to watch what their scripts are doing in the background during runtime using simple print statements to assist in developing and debugging. This type of analysis capability makes historical analysis easy from any computer within the network. If a tester is at a developer's desktop showing him or her how to replicate an issue, and a database index or account name is needed, this info can be retrieved easily from the log file displayed on the web. Replicating a bug becomes a matter of repeating line by line exactly how and where the test script failed.

Another benefit of online results display is that bugs may vanish into thin air. Proactive developers may fix an error before a tester has time to report it! And last but not least, management loves to take a peek at the test results to see if the production website is operating properly, or to find out when a certain failure occurred. This allows everyone the benefit of seeing the test results from the comfort of their desktop computer. If the logs are clear enough, human interpreters probably won't even have to ask questions about how the tests work. But that's up to the test architect.

Using open source technologies to distribute test results

Say your solution to distributing automated results information is to do it online. How would you do it? What tools would you use? Do you have a machine that can be a web server? Does the machine have enough processing power to handle the amount of traffic the site may get? Where do you get the web server software? What if you have no idea how to install a web server? How much can you spend on such an initiative? Is the benefit of having results available for historical analysis on a website worth the trouble in making it happen? These are all great questions that can be solved by using open source technologies. The Apache web server can run on Windows, and some of the distributions out there come preconfigured with MySQL database and PHP web scripting language all bundled up in an executable that can be easily installed on a desktop computer. Most Linux distributions have useful tools for distributing results, like mail programs and scripting languages. Free is a great price, and the documentation is available through the web complete with free scripts and open discussion forums. That's the beauty of using open source software for the results distribution. A lot of work gets completed quickly with relatively no cost. The only drawback is there needs to be a skilled operator, but these skills can be learned quickly via the documentation found on the web.

As of now, there is a market for a well-written functional testing tool. But in terms of the supporting software, like languages for parsing results files and displaying them on the web, there are advanced, very free, and well documented languages like PERL and PHP, and tools like Apache and MySQL available via the open source software development community. A little bit of investigation in these areas goes a long way to finding out what's out there, and getting started with using reliable free software to further the testing initiatives at any software development shop.

Conclusion

Bugs in the system can and will cause users to get frustrated and quit. Avoiding a disruption to the user experience is possible by designing and implementing a quality assurance methodology that includes both manual and automated testing. By using teamwork to get the job done, automated tests, their results, and the benefit of historical and real time trend analysis fosters confidence and a timely understanding of system health. To reliably capture an application wellness snapshot, it's necessary to design a test system that's flexible, maintainable, and modular. Plainly a relationship exists where if more attention is given to quality by the application vendor, the more attention users will give to that application. It's about providing users with a good experience the first time, and retaining them for years to come. Automated testing contributes to improving the user experience, and helps build user confidence and loyalty in the product for many happy returns.

Appendix A: Track Speaker Announcement

Announcement printed in STARWEST 2003 brochure:

TEST AUTOMATION
Thursday, October 30, 2003
1:30 PM . 2:30 PM

Rapid Automated Test Script Development in a Fast Paced Web Shop
Clay Givens, HotJobs, a Yahoo! Company

Before you start writing automated test scripts, you need an automation framework that matches your application and environment. Not surprisingly, Web applications at HotJobs, a Yahoo! Company are constantly changing and need a fast, flexible methodology for test automation. Clay Givens illustrates the importance of designing this flexibility into automated tests with an emphasis on modularity for speed. He describes how to design automated tests, run them efficiently, and share test results using both commercial and homegrown open-source testing tools.

- Design end-to-end automated tests for the Web that hit core functionality
- Automated test scheduling pitfalls and challenges
- Distribute test results on the Web

Appendix B: Sample Log File

19:40:07:---------- Starting suite_a_testcases.t
USCreateEmployerJobSyncSearchUpdateSyncDeleteSyncSearch on http://hotjobs.yahoo.com
19:40:21:After clicking member login, page [Welcome to HotJobs.com - Member Home] was loaded.
19:40:21:Logged into US employer account [account1]
19:40:52:Inserted job J681073BB on US site, in account [account1]
19:40:57:Sleeping for 300 seconds to wait for sync.
19:46:12:Executing job search by KW[Job1suite_a_c3_d1.ini] CITY[Salem] STATE[OR]

19:46:13:Found job after sync via basic job search. Title was updated
19:46:32:Updating job J681073BB TITLE[Job2suite_a_c3_d1.ini] CITY[New York] STATE[New York]
19:46:40:Updated job J681073BB in account [account1]
19:46:40:Sleeping for 300 seconds to wait for sync.
19:51:56:Executing job search by KW[Job2suite_a_c3_d1.ini] CITY[New York] STATE[NY]
19:51:57:Found job after sync via basic job search
19:52:10:Sleeping for 300 seconds to wait for sync.
19:57:25:Executing job search by KW[Job2suite_a_c3_d1.ini] CITY[New York] STATE[NY]
19:57:26:Could not find job after deletion and sync via basic job search
19:57:45:Logout of US Employer account hjus_blamar1 successful.
19:57:45:---------- Exiting suite_a_testcases.t USCreateEmployerJobSyncSearchUpdateSyncDeleteSyncSearch on http://hotjobs.yahoo.com
...
20:19:19:---------- Starting suite_a_testcases.t USCreateCPResumeSearchUpdateSyncSearch on http://hotjobs.yahoo.com
20:19:33:Logged into US job seeker account: account1
20:19:33:Logged into US seeker account account1
20:19:45:Deletion of resume 1 successful.
20:20:06:Deleted all resumes in job seeker account account1
20:23:10:Created resume [r1337494vc] with title [CP 08/01/03 20:20:06].
20:23:10:Inserted a copy paste resume [r1337494vc] on US site
20:23:10:Sleeping for 300 seconds to wait for sync.
20:28:25:After clicking member login, page [HotJobs.com - Member Logins Exceeded] was loaded.
20:28:25:*** Error: Login page [HotJobs.com - Member Logins Exceeded] returned. Will override and attempt to login again.
20:28:25: Page Title: HotJobs.com - Member Logins Exceeded
20:28:25: Page URL: http://member.hotjobs.com/cgi-bin/authcached-login
20:28:25: Occurred in USMemberLogin at suite_a_functions.inc(313)
20:28:25: Occurred in USCreateCPResumeSearchUpdateSyncSearch at suite_a_testcases.t(367)
20:28:32:Login to US Employer account [account1] successful.
20:28:50:Found resume r1337494vc in US Employer account [account1] search.
20:29:09:Logout of US Employer account [account1] successful.
20:30:12:Successfully updated resume r1337494vc in account account1
20:30:12:Sleeping for 300 seconds to wait for sync.
20:35:27:After clicking member login, page [Welcome to HotJobs.com - Member Home] was loaded.
20:35:27:Login to US Employer account [account1] successful.
20:35:46:Found resume r1337494vc in US Employer account [account1] search.
20:36:05:Logout of US Employer account [account1] successful.
20:36:17:Deletion of resume 1 successful.
20:36:38:Deleted all resumes in job seeker account account1
20:36:44:Clicking the sign out link from home page.
20:37:03:Logged out of US seeker account account1
20:37:03:---------- Exiting suite_a_testcases.t USCreateCPResumeSearchUpdateSyncSearch on http://hotjobs.yahoo.com

Bibliography

Craig, R.D, and Jaskiel, S.P., "Systematic Software Testing", STQE Publishing, 2002
Beck, K, "Extreme Programming Explained: Embrace Change", Addison-Wesley, 2000
Bennatan, E.M., "Software Project Management", 2nd edition, McGraw-Hill International, 1992