2008

PACIFIC NW
SOFTWARE QUALITY CONFERENCE

Collaborative Quality

October 13-15, 2008

Conference Paper excerpt
From the

CONFERENCE PROCEEDINGS

Non-Regression Test Automation

Douglas Hoffman
Software Quality Methods, LLC.
Doug.Hoffman@acm.org
www.SoftwareQualityMethods.com

8/3/2008

Experience and qualifications:

Douglas Hoffman has over twenty-five years experience in software quality assurance. He has degrees in Computer Science, Electrical Engineering, and an MBA. He has been a participant at dozens of software quality conferences and has been Program Chairman for several international conferences on software quality. He architects test automation environments and automated tests for systems and software companies.

He is an independent consultant with Software Quality Methods, LLC., where he consults with companies in strategic and tactical planning for software quality, and teaches courses in software quality assurance and testing. He is a Fellow of the ASQ (American Society for Quality), founding member of SSQA (Silicon Valley Software Quality Association) and AST (Association for Software Testing), and is also a long time member of the ACM and IEEE. He is Past Chair of the Santa Clara Valley Software Quality Association (SSQA) and Past Chair of the Santa Clara Valley Section of the ASQ. He has also been an active participant in the Los Altos Workshops on Software Testing (LAWST) and dozens of its offshoots. He was among the first to earn a Certificate from ASQ in Software Quality Engineering, and has an ASQ Certification in Quality Management.

Non-Regression Test Automation

Introduction

In my experience, most automated tests perform the same exercise each time the test is run. They are typically collected and used as regression tests, and are unlikely to uncover bugs other than very gross errors (e.g., missing modules) and the ones they were specifically designed to find. Testers often think of test automation as GUI based scripted regression testing, using scripts to mimic user behavior. Tool vendors actively sell the automating of manual tests. These are very narrow views of the potentially vast possibilities for automating tests because they are limited to doing things a human tester could do. When we think of test automation we should first think about extending our reach – doing things that we can't do manually. This topic describes getting past the limitations of automated regression suites and generating more valuable kinds of test automation.

The difficult part of automation is determining whether or not the software under test (SUT) responds correctly. Automated tests can easily feed huge numbers of inputs to the SUT. Variation of inputs in automated tests can use data driven approaches or random number generators. In the absence of an excellent mechanism for recognizing expected SUT behavior (an oracle), verification is time consuming and extremely difficult. With an oracle, automated tests can be designed using potentially huge numbers of variable inputs to evaluate the responses of the SUT – without doing exactly the same test exercise each time. This is not to say that the tests are not repeatable, as described in the section **Mechanisms for Non-Regression Automation** below. This type of automated test has the chance of uncovering previously undiscovered bugs.

Regression Testing Defined

According to IEEE Standard 610.12-1990, "**regression testing** [is] *Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements*." Mathur defines "*the word regress means to return to a previous, usually worse, state. Regression testing refers to that portion of the test cycle in which a program P' is tested to ensure that not only does the newly added or modified code behave correctly, but also that code carried over unchanged from the previous version P continues to behave correctly*."[i] Wikipedia describes the idea as applied to software testing: "**Regression testing** *is any type of software testing which seeks to uncover regression bugs. Regression bugs occur whenever software functionality that previously worked as desired, stops working or no longer works in the same way that was previously planned. Typically regression bugs occur as an unintended consequence of program changes. Common methods of regression testing include re-running previously run tests and checking whether previously fixed faults have re-emerged*."[ii] Although the definitions are consistent with one another, common usage in software testing is the rerunning of previously run tests and getting the same results.

Sources of Regressions

From a practical standpoint, regressions occur due to poor source code management (SCM), incomplete bug fixes, or unintended consequences (side effects) of code changes and bug fixes.

The first two should not occur or should be immediately discovered. Unintended consequencs are difficult, if not impossible to eliminate because in this case the change in one place causes a bug in some remote part of the system. Unintended consequencs, by definition cannot be anticipated.

Arguably, regression tests were initially created as a response to SCM problems. However, bugs should not be introduced through poor SCM today because the principles and techniques were mastered in the 1970's and many tools exist which make SCM straightforward. Some organizations fail to follow well established source code management techniques, with the expected consequence of introduction and re-introduction of bugs. In these instances regression tests can find reintroduced bugs. This first category of regressions are virtualy eliminated by the use of current SCM tools. Regression tests that find this type of regressions are really finding development process problems.

Incomplete bug fixes may occur for many reasons; e.g., because of misunderstanding of the bug's scope, fixing of a different bug, resource constraints, or ineffective developers. These are generaly caught when every bug fix is verified in the initial build containing it. The likelihood of getting a complete fix is high when changes are fresh and the developers are most familiar with the code.

Although the regression test case may be used to verify the veracity of a fix, bug fix verification is usually done specifically for each fixed bug, not as a general part of running the entire regression suite.

Advantages of Manual Tests

Automation of a manual test reduces its variability. Even though a manual tester may be attempting to follow the same steps each time, humans are prone to making mistakes and the 'mistakes' sometimes lead to the discovery of new defects. Mis-keying an input, clicking on the wrong spot on the screen, typing "Yes" in a date field, or pasting a file's contents instead of its name are all examples that might turn up a bug. Even though the human may quickly correct for the error without apparent changes in SUT behavior, there is some chance that an error may be exposed by this activity. Variation leads to unexpected behavior, and in these situations validation and recovery are very difficult problems to solve in an automated environment. Automated regression tests always provide exactly the values programmed and expect the same result each time.

A regression test does the same thing over and over. It covers the same conditions and inputs each time it is run. Since software doesn't break or wear out, we would not expect to find a functional error the second (or subsequent) time we run a test on a build. [1] An automated version of the test becomes progressively less likely to find errors each time it is run.

[1] In some respects, the second and subsequent times we run a test we are looking for different kinds of errors – e.g., variable initialization, test design, interference problems, etc. The question we asked the first time the test was run has been answered, and each subsequent time it runs the software is starting from a slightly different set of conditions.

Limits of Regression Tests

As defined above, a regression test runs to test current versus previous behavior of the SUT. The majority of existing automated tests are automated versions of manual regression tests. These are not the most powerful tests we could make (in terms of the likelihood and types of bugs to be found) and there are important and valuable automation we could be creating. Even so, there are many circumstances where automating regression tests is well justified.[iii]

However, regression tests are usually not an effective way to look for new defects. The vast majority of defects ever found by automated regression tests are actually found by the manual running of the test prior to automating it.[iv] We further reduce the chance of finding bugs by doing *exactly* the same thing each time.

To better understand the likelihood of finding bugs with automated regression tests, try the following [mental] exercise. Think of how a [manual] regression test is created and compare that with creating a demo script.

Creating a Manual Regression Test

The most common regression test creation technique:

- Conceive and write the test case
- Run it and inspect the results
- If the program fails, report a bug and try again later
- If the program passes the test, save the resulting outputs as expected results
- In future tests, run the program and compare the output to the expected results
- Report an exception whenever the current output and the saved output don't match

What we do for a demo is try to minimize the chance of encountering a bug. Now, think of how a demonstration exercise might be created (e.g., for a trade show).

Creating a Demo

A demo creation technique for new products (finding a "happy path"):

- Conceive and create the demo exercise
- Run it to see that it demonstrates the desired attributes
- If the program fails, report a bug and either wait for a fix or find a different way to do the exercise that doesn't fail
- Remember the specific steps and values that work when the program passes
- In future demos, do not deviate from the steps or try new values to minimize the chance of failure
- Be at risk for embarrassment if the program fails to do the demo correctly

The two processes are nearly the same. The demo developer finds a path that works and then sticks to it. A regression test developer is doing virtually the same thing. Once the regression test has been created, the likelihood of encountering a bug has been minimized. The only bugs likely to be found are those reintroduced through poor source code management or gross errors (so

obvious that any action will expose them). *Any* other exercise of comparable length will probably find more bugs because in addition to finding the gross errors there is some chance of finding legacy bugs that regression tests have no chance of finding.

Non-Regression Automation

There are many alternatives to automating regression tests. In addition to looking for more bugs by varying the tests, as mentioned in the Introduction, automated tests can extend testing into seeking new types of errors that can't be practically found through manual testing. Non-regression automated tests can do something differently each time they are run and most may not be performed manually at all..

Some types of errors sought through testing are only practical using non-regression automated tests:

- Buffer overruns, some types of security issues (e.g., found through potentially massive numbers of variations on input, huge input files, large data sets, etc.)
- Non-boundary special cases (e.g., internal boundaries, divide by zero, state machine errors)
- Memory leaks, stack overflows (e.g., accumulation errors)
- Memory corruption, stack corruption (e.g., in memory errors)
- Resource consumption/exhaustion (e.g., system effects)
- Timing errors (e.g., errors with small windows of opportunity)

Mechanisms for Non-Regression Automation

There are many types of non-regression automated tests. Most use a computer's pseudo-random number generator to introduce variation from run to run. Pseudo-random numbers are interesting because the series of values is statistically random but repeatable given a seed value. A test can generate a new random sequence or use a seed to rerun a sequence for fault isolation. Pseudo-code for the typical mechanism (for 100,000 possible seed values) is shown below:

```
IF (SEED parameter not present) /* Use SEED if it's present */
   /* Generate random seed if SEED not present */
   SEED = INT( 100000 * RND() ) /* SEED between 0 and 99,999 */
   PRINT SEED /* Print out SEED so the series can be repeated if necessary */
ENDIF
FIRST = RND(SEED) /* Use SEED to generate the first random number */
/* RND( ) will generate the next random number in the sequence after the first one */
```

The list below provides some of the types of non-regression tests in use today. Each is described briefly in the Appendix.

Nine examples of non-regression automation:

- Data driven / data configured
- Model based
- Random walks (both stochastic & non-stochastic)
- Function equivalence (using random input)
- A/B comparison (using potentially massive numbers of input values)
- Statistical models
- Real-time external monitors (e.g., memory leaks or data base corruption)
- Cooperating processes
- Duration testing, life testing, load generation (combining existing automated tests)

There are several approaches to automating non-regression tests based upon test style and how the test communicates with the SUT.

Test case style examples:

- Data driven commercial program
- Real-time monitoring utility
- Driver/stub combination
- Configurable/data driven custom program
- Individual program/test

There are also many touch-points where an automated test can interface with the SUT:

- Public API based
- GUI API based
- Non-GUI API based
- Individual program/tests
- Trusted objects

An example may serve to explain how the various characteristics work together. In the article, *Heuristic Test Oracles*[v] I describe testing a sine function in a system library. The test generates randomly selected sequences of inputs and checks to see that the sine function returns sequences of monotonically increasing or decreasing results. It looks for non-boundary special conditions (discontinuities) using a statistical model (monotonic increases and decreases) in an individual program (custom written test) that interfaces through the public API.

Test Oracles

The key element required to make any tests worthwhile is the oracle (method to check for expected/unexpected behavior). A person runs manual tests, using five senses and an untold number of creative oracles to observe and check SUT behavior. Whether using a specification or intuition, a person notices time between events, feels or hears clicks on a disk, sees sparkles on a

screen, or smells over-heated wiring. Automated tests only check elements identified for it, and some kind of an oracle must be available to provide the individual values to determine whether or not the behavior is expected. Oracles for automated tests are critically important and can be quite varied.[vi],[vii],[viii]

Oracle examples:
Reference Functions – equivalent function or saved values

- Previous values or previous version
- Competitor's product
- Standard function
- Custom model

Computational or Logical Modeling

- Inverse Functions – "round tripping"
  - Mathematical inverse
  - Operational inverse (e.g., split a merged table)
- Useful mathematical rules –
  - e.g., sin2(x)+cos2(x) = 1
  - e.g., time of event order = event number order

Heuristic Functions[ix] – incomplete but usually right

- Almost-deterministic approach
  - Check only some of the outcomes
- Compare incidental but informative attributes
  - Durations
  - Orderings
- Check (apparently) insufficient attributes
  - ZIP Code entries are 5 or 9 digits
- Check probabilistic attributes
  - X is usually greater than Y
  - Statistical distribution (test for outliers, means, predicted distribution)

Conclusions

Regression testing is the rerunning of previously run tests and expecting the same results. Automated regression tests are the primary mechanism used in test automation. Although improvement in tools and software development processes have virtually eliminated the original cause of regressions, there are still circumstances under which regression testing makes sense. Yet, regression tests are weak at finding new bugs and do not extend the scope of bugs found. Like product demos, regression tests minimize the opportunity for finding defects by design. Automating regression tests further reduces the chance of finding defects.

There are a great number of alternatives to simple regression automation that provide powerful tests for finding bugs that manual testing cannot, such as memory leaks or subtle timing errors. Non-regression test automation gives us capabilities that a manual tester does not have.

Many different types of non-regression automated tests are possible and there are many mechanisms that can be used to build non-regression automation. They can be designed to look

for types of bugs that typical regression tests cannot. There are also several test architectural styles, from custom program/tests to data driven commercial programs. Many possible touch points and monitoring techniques can also be used for SUT stimulation and result monitoring.

The key element required to make any tests worthwhile is the oracle. Oracles for automated tests are critically important and can be quite varied as described in the referenced papers and slides.

What automated tests can't do for us (yet):

- Know the expected results in all cases
- Notice things that we haven't specifically told the test (or the test mechanisms) to look at
- Analyze boundary conditions, partitions, models, etc., to determine the best test conditions to cover [With the exception of some tools that will tell us how to test that the code does what the code does]
- Decide on new courses of action (that aren't specifically written into the test) based on detection of potentially interesting occurrences

Appendix

Types of non-regression automation described

Data driven tests read input values with corresponding expected results. In this way the values being used in the tests can be repeated or different each time. Data input is not restricted to specific arithmetic or alphabetic values but may include function calls, parameter names, and other application specific attributes to be tested.

Data Configured tests read input values to enable/disable or configure the test case code. (e.g., printer paper sizes or printer dot densities)

Model Based tests use a model of the SUT. (e.g., a state machine or menu tree)

Random Walks use a series of pseudo-random values as input.

Stochastic tests are ones where the sequence matters. (e.g., logging-in before beginning a financial transaction)

Non-Stochastic tests are ones that are [theoretically] independent of one another. (e.g., order of loading of printer fonts)

Function Equivalence is a form of random walk when a true oracle is available and therefore any random input can be used whether the function is stochastic or not. (e.g., there are two versions of the same product)

A/B Comparison is used when a very large number of results are recorded and compared from one run to the next (e.g., potentially massive numbers of result values)

Statistical Models

Real-time External Monitors are programs run at the same time as the test but are monitoring characteristics external to the actual SUT (e.g., memory leak or data base corruption detectors)

Cooperating Processes are tests in which the test or monitor is communicating as a peer process with the SUT.

Duration Testing is done by running a series of tests (or one test repeatedly) continually for a period of time.

Life Testing is done by running a series of tests (or one test repeatedly) continually until the system fails.

Load Generation is one or more series of tests run in the background when a specific test is run. (e.g., doing performance analysis or to test that a file can be accessed even though other processes are accessing it)

Statistical Model is based on the statistical characteristics of the data rather than the data values. For many functions, input with a specified mean and standard deviation will generate results with a corresponding mean and standard deviation. Many other statistical characteristics are available and may be used, especially when dealing with very large data sets and unpredictable expected results.

References

[i] Mathur, Aditya P., **Foundations of Software Testing** (2008, Dorling Kindersley (India) Pvt. Ltd) ISBN 81-317-1660-0

[ii] June 14, 2008 at http://en.wikipedia.org/wiki/Regression_testing

[iii] Bach, James; "*Reasons to Repeat Tests*" http://www.satisfice.com/repeatable.shtml

[iv] Kaner, Cem, "*Avoiding Shelfware*" http://www.kaner.com/pdfs/shelfwar.pdf

Merick, Brian; "*Classic Testing Mistakes*" http://www.exampler.com/testing-com/writings/classic/mistakes.pdf

[v] Hoffman, Douglas "*Heuristic Test Oracles*" STQE Magazine, April, 1999 http://www.softwarequalitymethods.com/Papers/STQE%20Heuristic.pdf

[vi] Hoffman, Douglas "*Using Test Oracles in Automation*" Software Test Automation Conference, Spring 2003 http://www.softwarequalitymethods.com/Slides/Oracle%20Auto.pdf

[vii] Hoffman, Douglas "*Advanced Test Automation Architectures*" Conference for the Association for Software Testing (CAST), 2007 http://www.softwarequalitymethods.com/Slides/TestAutoBeyondX2-CAST07.pdf

[viii] Hoffman, Douglas "*Using Oracles in Automation*" PNSQC, 2001 http://www.softwarequalitymethods.com/Papers/Auto%20Paper.pdf

[ix] Hoffman, Douglas "*Heuristic Test Oracles*" STQE Magazine, April, 1999 http://www.softwarequalitymethods.com/Papers/STQE%20Heuristic.pdf