Software Testing, Quality Assurance and Maintenance Winter 2010

Lecture 34 — March 31, 2010

Patrick Lam *version 0*

Usage Models

We have alluded to the complexity of software systems, but all of the systems that we've seen in this course have been tiny compared to, say, Google's infrastructure. In particular, here are two assumptions:

- Hardware doesn't randomly fail on you;
- Management can adequately prioritize which bugs to fix (and you can fix them.)

Consider this quote:

> The Canadian Government wants to know from Toyota:
>
> How did this happen?
> How did the defect go undetected?
> How can this be prevented?
>
> As someone who's worked support for Enterprise software, I'd like to try my hand at answering the government:
>
> You have a complex system. You QA it to death using user workload simulations. You correct any faults that occur. But at some point, you have to stop testing. You put the software out into the world and then, six months later, the customer reports a problem that QA never ran into ever. Why? Because no QA is absolutely thorough and no system is absolutely fault-free. The bug may only materialize itself after 6 more hours of operation than QA tested or only if it's run on 3 more servers than QA tested it on. Is the solution to test for 6 more hours or on 3 more servers? No. Because there will always be a bug that is just beyond QA's grasp. This is how all complex systems work: there are defects you can never predict. This is also why people suddenly keel over from heart defects they never knew they had.

(Sandra)

To reason about the reliability of more complicated systems, we turn to usage models. I'll first discuss the foundations behind usage models, and then discuss complications (alluded to above).

Basic Idea

So far, we've talked about constructing test cases, which are basically deterministic, and put them together to get test suites. Underlying article of faith: good things happen if you have enough coverage in your test suite.

Statistical reasoning, in general, abandons the notion of coverage over the entire input space, and instead tries to determine properties of the underlying distribution by sampling. (Consider opinion polls versus censuses).

We will therefore create a usage model and sample tests from this usage model. If the usage model is good, it'll (1) identify the most likely failures and (2) allow QA to certify that the system will not fail under the tested workloads.

Examples

Usage models could be grammars, FSMs, or Markov chains. Here are two simple usage models from [1]. Our example will be a "simple interactive system", with 2 screens: Main Menu (where you can Display Screen or Exit), and a Display Screen, which always returns to the Main Menu.

FSM. We can represent the usage model by a finite state machine:

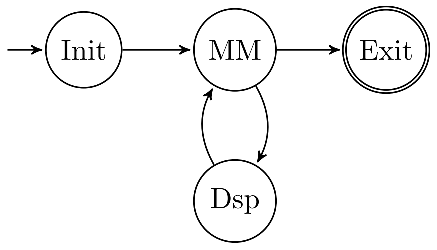

Figure 1: Finite State Machine usage model.

This representation of the usage model can help testers understand and verify the usage model, since it hides some extra details that we'll add in the Markov Chain usage model.

Markov Chain. The problem with the FSM is that it doesn't tell us how likely each transition between states might be. Markov Chains augment automata by including transition probabilities, which enable statistical testing.

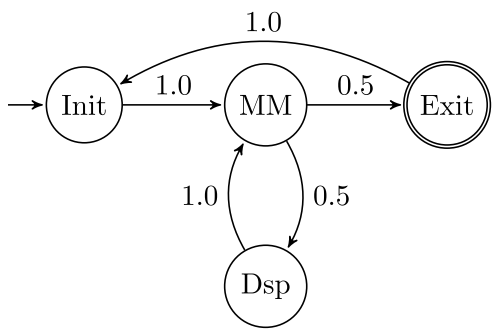

Figure 2: Markov Chain usage model.

Basically, the probabilities come from observations, or more frequently, guesswork (see below). We can run workload simulations using Markov Chains, and if the probabilities are correct, these simulations can tell us about the reliability of the system (by counting the number of failures) and can help pinpoint failures.

Splitting States. Markov Chains have no memory: the next state is determined completely from the current state and a random number. Workloads, however, do exhibit dependencies on history. We can hack up the Markov Chain to encode some memory:

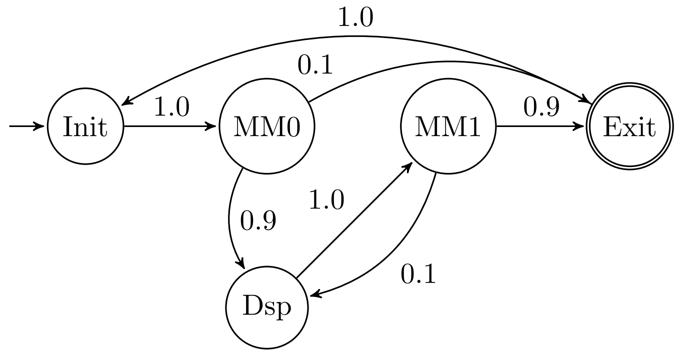

Figure 3: Hacked Markov Chain usage model.

Creating a Usage Model

Creating a FSM will get you most of the way to having a Markov Chain, and we've seen how to create FSMs. It remains to compute probabilities. You can do that by observing workloads, when possible, or by guessing. Given a Markov Chain, you can verify its correctness by comparing it against available information. As the software evolves, the usage model needs to evolve as well.

Using the Usage Model

Testing with usage models requires an additional step: statistical analysis, which happens after running the models. Otherwise, the process is the same as usual: define stopping criteria for software to exit the testing phase; generate test cases and decide what success means; do the testing; analyze the results; fix the software; iterate as needed.

Experiences from Industry

Here are some notes from someone works for a provider of grid and cloud software doing support. (I'm told that QA is much less precise in the real-world, which shouldn't be news to you.) She is familiar with three types of testing: feature testing, longevity testing, and workload simulation.

Software Architecture. Multiple layers, including lots of legacy code. Problems when legacy code meets new code.

Feature Testing. Individual thorough testing of each new feature compared to their designs. Automatic regression testing of older features. No interaction testing.

Sometimes customer requirements don't match designs. "QA tries to catch these things prior to software release by asking Support if they think this 'makes sense'. However, when they get to this stage, it's usually too late to redesign the feature. Usually the feature gets tweaked or, more often than not, a disclaimer gets put in the release notes, e.g. 'Please note that Feature X will only work when used in J-Mode. Feature X should never be used in conjunction with Feature Z.' "

Their software also has to work with the operating system. Linux patches have often caused havoc.

Longevity Testing. Provides support for statements like "Our software is certified to maintain X connections from Y servers for Z hours without cleanup or restart." See [2] for more information on longevity testing. Designed to exceed expected usages; customers typically shut down their systems periodically, resetting the longevity clock.

Platform doesn't actually know how the customer will use the product in practice: "We had one customer call in with some random complaint about things going bonkers. When we looked into it, we discovered that they were running about 500 more connections than we supported – 1000 more than they had used with the previous release!" They estimate workloads from any information they can get from the customers. Customers aren't always willing to describe their expected usage.

Workload simulation / Usage Patterns. QA relies on information from Support about usage patterns and configurations. This stage includes feature interaction tests.

Course Summary

This course had a theoretical side and a practical side. We defined software faults and failures, tests and test suites, and coverage criteria and subsumption. On the theoretical side, you learned to evaluate test suites against various notions of coverage. In particular, you used graph coverage, logic coverage, syntax-based coverage (and mutation), and input-space coverage to guide test suite construction. On the practical side, you applied tools for unit testing (particularly JUnit), how to write good unit tests, learned about regression testing, tools for testing web applications, test plans, bug reports, and usage models.

References

[1] Gwendolyn H. Walton, J. H. Poore and Carmen J. Trammell. Statistical Testing of Software Based on a Usage Model. *Software—Practice and Experience, 25(1):97–108, 1995.*

[2] Steven Woody. Software Longevity Testing. *Better Software Magazine, September/October 2009.* `http://www.stickyminds.com/BetterSoftware/magazine.asp?fn=cifea&id=121`.

Appendix

Notes courtesy an anonymous source.

So I looked over your notes quickly (it's hard to have quiet study time with a 7-month-old!) and I can assure you that in the real world QA is hardly as precise a science. :)

Just a quick disclaimer: I actually work in Support. My dealings with QA are usually in the User Experience/test- & feature-creation stage and the OMG-the-customer-is-*pissed* stage.

Platform makes grid and cloud software. The software is made up of different "layers" that sit atop the OS. Each component layer has its own daemons and controls. Some of the layers are thick—THICK!—with legacy code. That stuff doesn't get modified anymore, so it doesn't get QAed anymore, really. I hate to say it, but the real big problems usually happen when the legacy code meets the New Code.

Because I only get involved at the creation stage and the after-the-fact stage, the only testing I'm really aware of is the longevity testing, the user workload simulation testing and the feature testing.

Feature testing: At each new release, new featured are QAed thoroughly to make sure that they work as designed. The features are tested individually, not in conjunction with other features. Testing of older features is done through an automated system, to just check that nothing was broken in creating the new release.

Longevity testing: I'm not sure what goes into this, but I know that the majority of CYA statements comes from this testing. This is where they determine the operating parameters. So this is the "Our software is certified to maintain X connections from Y servers for Z hours without cleanup or restart." The longevity criterion are designed to go over and above what we typically see in the field. Customers will usually shut down their systems for maintenance once a month or so and they all have budget constraints/whatnot that dictate how many servers/applications/whatever they can have at any time. We just make sure that we run everything to go over and above what we expect.

Workload simulation/Usage patterns: This is when QA works with Support. We tell QA how the customers have been using the software, what their configurations are like, etc. Then QA uses this info to test the features together. In the event of a new feature, they ask us how the customers might use the new features and test that.

If you note up at "feature testing" I said that they check that the features work "as designed"? That is one of the biggest sources of problems at the end-user level because what the software was *designed* to do and what the customer *expects* it to do are two different things. So you get a situation where the customer wants to use a feature in what they think is a reasonable way, but ends up giving them totally unexpected results. We actually had a situation where a bug in v3 of the product was thought to be a feature by the customer. When version v3.1 came out and we fixed the bug, the customer was *pissed*. It took 3 years (!) to sort out the mess. This was one of the first issues of customer management I had to deal with and I finally got this resolved on my last day of work before mat leave!

The other issue with "as designed" is that the design isn't always ... shall we say ... good? Like it makes sense and works on its own, but doesn't work well in real usage terms. QA tries to catch these things prior to software release by asking Support if they think this "makes sense". However, when they get to this stage, it's usually too late to redesign the feature. Usually the feature gets tweaked or, more often than not, a disclaimer gets put in the release notes. Something like, "Please note that Feature X will only work when used in J-Mode. Feature X should never be used in conjunction with Feature Z."

Re. workload simulation and longevity: we're working with priors here. We don't know how the customer will actually use the product. We had one customer call in with some random complaint about things going bonkers. When we looked into it, we discovered that they were running about 500 more connections than we supported – 1000 more than they had used with the previous release! Turned out that they had gotten some extra budget and gone whole hog! It took a lot of finessing to get them to scale back. So everything is speculation.

In addition to all this, because the software is a layer above the OS, we have to work with Sun, Linux and Microsoft to make sure that their software and our software don't clash. 99% of the time, this works out and all is well. What we can't predict is what will happen when the OS gets patched. We've had myriad issues with Linux patches wreaking havoc with our software. And these are totally and utterly unpredictable. There is no way that QA can ever, in a million years, predict these things. The customers will patch the OS and then Bad Things will happen and it will literally take months to sort the whole thing out. In these cases, you have to work with Linux to get the issue resolved. It's no fun.