developer.*

Test Smarter, Not Harder

by Scott Sehlhorst

Introduction: Complexity Leads to Futility

Imagine we are developing a web page for customizing a laptop purchase.

If you've never configured a laptop online before, take a look at Dell's "customize it" page for an entry level laptop. The web page presents eleven questions to the user that have from two to seven responses each. The user has to choose from two options in the first control, two in the second, and so on. The user has seven possible choices for the last control.

When we look at all of the controls combined, the user has to make (2,2,2,2,2,3,2,2,3,4,7) choices. This is a simple configuration problem. The number of possible laptop configurations that could be requested by the user is the product of all of the choices. In this very simple page, there are 32,256 possibilities. At the time of this writing, the page for customizing Dell's high-end laptop has a not dissimilar set of controls, with more choices in each control: (3,3,3,2,4,2,4,2,2,3,7,4,4). The user of this page can request any of 2,322,432 different laptop configurations! If Dell were to add one more control presenting five different choices, there would be over ten million possible combinations!

Creating a test suite that tries all two million combinations for a high end laptop could be automated, but even if every test took one tenth of second to run, the suite would take over 64 hours! Dell changes their product offerings in less time than that.

Then again, if we use a server farm to distribute the test suite across ten machines we could run it in about 6 hours. Ignoring the fact that we would be running this type of test for each customization page Dell has, 6 hours is not unreasonable.

Validating the two million results is where the really big problem is waiting for us. We can't rely on people to manually validate all of the outputs–it is just too expensive. We could write another program, which inspects those outputs and evaluates them using a rules-based system ("If the user selects 1GB of RAM, then the configuration must include 1GB of RAM" and "The price for the final system must be adjusted by the price-impact of 1GB of RAM relative to the base system price for this model.")

There are some good rules-based validation tools out there, but they are either custom software, or so general as to require a large investment to make them applicable to a particular customer. With a rules-based inspection system, we have the cost of maintaining the rules. The validation rules are going to have to be updated regularly, as Dell changes the way they position, configure, and price their laptops.

Since we aren't Dell, we don't have the scale (billions of dollars of revenue) to justify this level of investment. The bottom line for us is that we can't afford to exhaustively test every combination. Dell's shareholders require them to grow their business, and these configuration pages are the vehicle by which Dell generates billions of dollars in revenue. They have to test it. The cost of errors (crashes, lost sales, mis-priced items, invalid combinations of features) is too high. With this level of risk, the cost of *not testing* (the cost of poor quality) is extremely high.

We Can't Afford to Test It

I was able to attend a training session with Kent Beck a few years ago. I was also honored to be able to enjoy a large steak and some cold beer with him that night after the training. When asked how he responds to people who complain about the cost of quality, Kent told us he has a very simple answer: "If testing costs more than not testing then don't do it."

I agree. There are few situations where the cost of quality[1] exceeds the cost of poor quality. These are situations where the needed infrastructure, test-development time, and maintenance costs outweigh the expected cost of having a bug. (The "expected cost"[2] is the likelihood (as a percentage) of the bug manifesting in the field, multiplied by the cost of dealing with the bug.)

The techniques described in this article are designed to *reduce* the cost of quality, to make it even less likely that "not testing" is the best answer.

Just Test Everything, It's Automated!

Two "solutions" that we have to consider are to test nothing and to test everything. We would consider testing nothing if we can't afford to test the software. When people don't appreciate the complexities of testing or the limitations of automated testing, they are inclined to want to "test everything." Testing everything is much easier said than done.

[1] http://tynerblain.com/blog/2006/02/22/software-testing-series-measuring-the-cost-of-quality/
[2] http://tynerblain.com/blog/2006/02/03/definition-of-expected-value/

Have you ever been on a project where the manager said something like, “I demand full testing coverage of the software. Our policy is zero tolerance. We won’t have bad quality on my watch.”?

What we struggle with here is the lack of appreciation for what it means to have “full coverage” or any other *guarantee* of a particular defect rate.

There are no absolutes in a sufficiently complex system—but that’s ok. There are statistics, confidence levels, and risk-management plans. As engineers and software developers, our brains are wired to deal with the *expected*, *likely*, and *probable* futures. We have to help our less-technical brethren understand these concepts—or at least put them in perspective.

We may get asked, “Why can’t we just test every combination of inputs to make sure we get the right outputs? We have an automated test suite—just fill it up and run it!”

We need to resist the urge to respond by saying, “Monkeys with typewriters will have completed the works of Shakespeare before we finish a single run of our test suite!”

Solving the Problem

There are a lot of applications that have millions or billons of combinations of inputs. They have automated testing. They have solutions to this problem. We just finished discussing how impractical it is to test exhaustively, so how do companies test their complex software?

In the rest of the article, we will explore the following approaches to solving the problem.

- Random sampling
- Pairwise testing
- N-wise testing

We will also explore the impact that changing the order of operations has on our testing approach, and the methods for testing when the sequence matters.

RANDOM SAMPLING

Early on in the software testing world, someone realized that by randomly checking different combinations of inputs, they would eventually find the bugs. Imagine software that has one million possible combinations of inputs (half as complex as our previous example). Each random sample would give us 0.000001% coverage of all possible user sessions. If we run 1,000 tests, we would still only have 0.001% coverage of the application.

Thankfully, statistics can help us make statements about our quality levels. But we can't use "coverage" as our key measurement of quality. We have to think about things a little bit differently. What we want to do is express a level of confidence about a level of quality. We need to determine the sample size, or number of tests, that we need to run to make a statistical statement about the quality of the application.

First we define a quality goal–we want to assure that our software is 99% bug free. That means that up to 1% of the user sessions would exhibit a bug. To be 100% confident that this statement is true, we would need to test at least 99% of the possible user sessions, or over 990,000 tests.

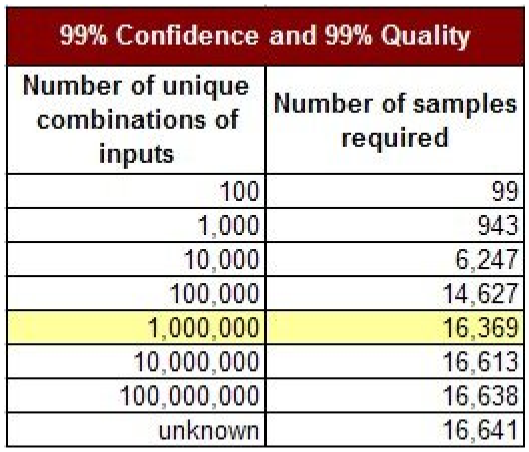
By adding a level of confidence to our analysis, we can use sampling (selecting a subset of the whole, and extrapolating those results as being characteristic of the whole) to describe the quality of our software. We will leverage the mathematical work that has been developed to determine how to run polls.

We define our goal to be that we have 99% confidence that the software is 99% bug free. The 99% *level of* confidence means that if we ran our sample repeatedly, 99% of the time, the results would be within the margin of error. Since our goal is 99% bug free code, we will test for 100% passing of tests, with a 1% margin of error.

How many samples do we need, if there are one million combinations, to identify the level of quality with a 99% confidence, and a 1% margin of error? The math for this is readily available, and calculators for determining sample size are online and free. Using this polling approach, we find that the number of samples we require to determine the quality level with a 1% error and 99% confidence is 16,369.

If we test 16,369 user sessions and find 100% success, we have established a 99% confidence that our quality is at least at a 99% level. We only have 99% quality, because we have found 100% quality in our tests, with a 1% margin of error.

This approach scales for very large numbers of combinations. Consider the following table, where our goal is to establish 99% confidence in a 99% quality level. Each row in the following table represents an increasingly complex software application. Complexity is defined as the number of unique combinations of possible inputs).

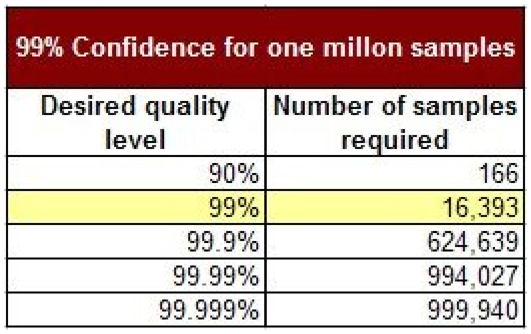


99% Confidence and 99% Quality	
Number of unique combinations of inputs	Number of samples required
100	99
1,000	943
10,000	6,247
100,000	14,627
1,000,000	16,369
10,000,000	16,613
100,000,000	16,638
unknown	16,641

We can see that the very few additional tests have to be run to achieve the same level of quality for increasingly complex software. When we have a modest quality goal, such as 99/99 (99% confidence in 99% quality), this approach is very effective.

Where this approach doesn't scale well is with increasing levels of quality. Consider the quest for "five nines" (99.999% bug free code). With each increase in the desired level of quality, the number of tests we have to run grows. It quickly becomes an almost exhaustive test suite.

Each row in the following table represents an increasingly stringent quality requirement, with the complexity of the software staying constant at one million possible input combinations.



99% Confidence for one millon samples	
Desired quality level	Number of samples required
90%	166
99%	16,393
99.9%	624,639
99.99%	994,027
99.999%	999,940

The random sampling approach does not provide a benefit over exhaustive testing when our quality goals are high.

Pairwise Testing of Input Variables

Studies have shown that bugs in software tend to be the results of the combination of variables, not individual variables. This passes our "gut-check" since we know that conscientious developers will test their code. What slips through the cracks is overlooked combinations of inputs, not individual inputs.

Consider a very simple laptop configuration page, having three selectable controls: CPU, Memory, and Storage. Each control has three possible values as shown in the table below.

Available user selections		
CPU	**Memory**	**Storage**
Bargain	Minimal	Large
Consumer	Average	Very Large
Power-user	Excessive	Huge

We successfully pass tests of each of the different values available in the CPU control. However, we discover that our test fails if the user selects a CPU value of "Consumer" *and* selects a Storage value of "Huge". This highlights an unknown dependency between the CPI and Storage controls.

Pair-wise testing[3] is designed to get coverage of every possible combination of two variables, without testing every possible combination of all the variables. For this example, there are 27 unique combinations of all of the selections. The following table shows the first 9 combinations. An additional 9 combinations are needed for each of the other CPU selections.

Available user selections		
CPU	**Memory**	**Storage**
Bargain	Minimal	Large
Bargain	Minimal	Very Large
Bargain	Minimal	Huge
Bargain	Average	Large
Bargain	Average	Very Large
Bargain	Average	Huge
Bargain	Excessive	Large
Bargain	Excessive	Very Large
Bargain	Excessive	Huge
[…]	[…]	[…]

Exhaustive pair-wise testing will make sure that every unique combination of any two variables will be covered. The next table shows the combinations for this example.

[3] http://tynerblain.com/blog/2006/03/18/software-testing-series-pairwise-testing/

Available user selections		
CPU	**Memory**	**Storage**
Bargain	Minimal	Large
Consumer	Average	Very Large
Power-user	Excessive	Huge
Bargain	Average	Huge
Consumer	Excessive	Large
Power-user	Minimal	Very Large
Bargain	Excessive	Very Large
Consumer	Minimal	Huge
Power-user	Average	Large

With just 9 tests, we are able to exhaustively cover every unique pair of CPU and Memory, CPU and Storage, and Memory and Storage. Pair-wise testing allows us to get full coverage of the combinations of every two variables, with a minimal number of tests.

Pair-wise testing not only gives us full coverage of every pair of values, it also gives us (redundant) coverage of every single value for each control.

If we look back at our previous examples of laptop-configuration, we can calculate the numbers of tests required to get full pair-wise coverage. For the entry level laptop configurator, there are 32,256 possible unique combinations of inputs. We can test every unique combination of two variables with 31 tests. For the high-end laptop configurator, there are 2,322,432 unique combinations of inputs. We can test every unique combination of two variables with 36 tests.

N-wise Testing

The concept of pair-wise testing can be extended to N-wise testing–looking at every combination of N possible inputs. This is a simple extension of the idea behind pairwise testing. Good developers will catch the bugs caused by the combination of two variables. Even the best developers will overlook the three-variable (or four or more variable) combinations.

The following table shows how many tests are required to get full coverage of each N-wise combination of inputs for both the low-end and high-end laptops configurators.

N-wise test coverage		
N	**Low-end**	**High-end**
1	7	7
2	31	36
3	110	179
4	318	749
5	814	2812

This is a much more manageable situation. Exhaustive coverage required us to use 2.3 million tests, where using N-wise testing with N=3, yields only 179 tests! Existing studies have consistently shown that N=3 creates on the order of 90% code coverage with test suites, although the number will vary from application to application. We will use N=3, based on practical experience that N=4 tests rarely uncover bugs that were missed with N=3.

This approach only works when users are forced to enter values in a proscribed sequence and in cases where the sequence of entry is irrelevant. This set of tests won't give us representative coverage of what the users will do when they are allowed to make selections in arbitrary but relevant order. If order doesn't matter for us (for example, most API signatures have a fixed order, and many websites will process multiple inputs in a batch), then we have our desired methodology.

Order Relevance and Statistical Testing

There's been an assumption implicit in all of our calculations so far: that the order of selection in the controls is irrelevant. The available N-wise test calculation tools do not incorporate order of selection in their permutations–explicitly, they assume a fixed order of operations. When we test an API we have control over the order of processing–there are a fixed number of arguments, in a fixed order. People, however, do not always interact with the controls in a fixed order. And web service architectures may not be able to depend upon a predetermined sequence of events.

With 5 controls in an interface, we have 5! (factorial[4]) or 120 possible sequences in which selections can be made by a person. Although the user interface may incorporate dynamic filtering that prevents some subsets of out-of-sequence selection, N-wise testing is blackbox testing[5], and will not have access to that information.

For an interface with M possible controls, each script created by an N-wise test generator will have to be tested in M! sequences to get exhaustive coverage. If the controls are split across multiple screens, then we can reduce the number of sequences. For example, if there are 5 controls on the first screen, and five controls on the next screen, instead of considering 10! (3.6 million sequences), we can consider all first-screen-sequences in combination with all second-screen sequences (5! * 5! = 120 * 120 = 14,400 script sequences).

[4] http://mathworld.wolfram.com/Factorial.html
[5] http://tynerblain.com/blog/2006/01/12/foundation-series-black-box-and-white-box-software-testing/

In our example laptop configurators, there are 11 and 13 controls (all on the same page) for the low-end and high-end laptops respectively. This would imply 11! and 13! possible sequences (40 million and 6 billion).

We do not need to do exhaustive coverage of the sequencing permutations. An N-wise test is specifically analyzing the interdependence of any combination of N controls. As a lower-bound, we would only need N! sequences for each generated script. So our 179-script suite for the high-end laptop (with N=3) would need 3! (6) * 179 = 1,074 scripts to cover the product.

Here's the table for the lower-bound of scripts required to account for different values of N for both laptop-configurators.

N-wise test coverage when varying sequence of operations		
N	Low-end	High-end
1	7	7
2	62	72
3	660	1,074
4	7,632	17,976
5	97,680	337,440

This is a lower bound, because it assumes a perfect efficiency in combining unique sequences of each group of N controls. Existing N-wise testing tools do not (to the author's knowledge) take order of operations into account. For N=2, this is trivial–just duplicate the set of tests, in the exact reverse order.

We can take order of operations into account by treating the sequence as an additional input. We use the mathematical formula "X choose Y" which tells us the number of different combinations of Y values from a set of X values. The formula for calculating "X choose Y" is X!/(Y!*(X-Y)!) where X is the number of inputs and Y is the dimension of the desired N-wise test.

Here's the table of the number of combinations for each N, for both the low-end and high-end laptop configuration screens we've been discussing.

X Choose Y - number of unique sequences for given N-wise run		
N	Low-end	High-end
1	11	13
2	55	78
3	165	286
4	330	715
5	462	1,287

Here are the values, generally, for varying numbers of inputs.

Number of unique sequences for given N-wise run versus # of controls										
N	Number of controls									
	3	4	5	6	7	8	9	10	15	20
1	3	4	5	6	7	8	9	10	15	20
2	3	6	10	15	21	28	36	45	105	190
3	1	4	10	20	35	56	84	120	455	1,140
4		1	5	15	35	70	126	210	1,365	4,845
5			1	6	21	56	126	252	3,003	15,504

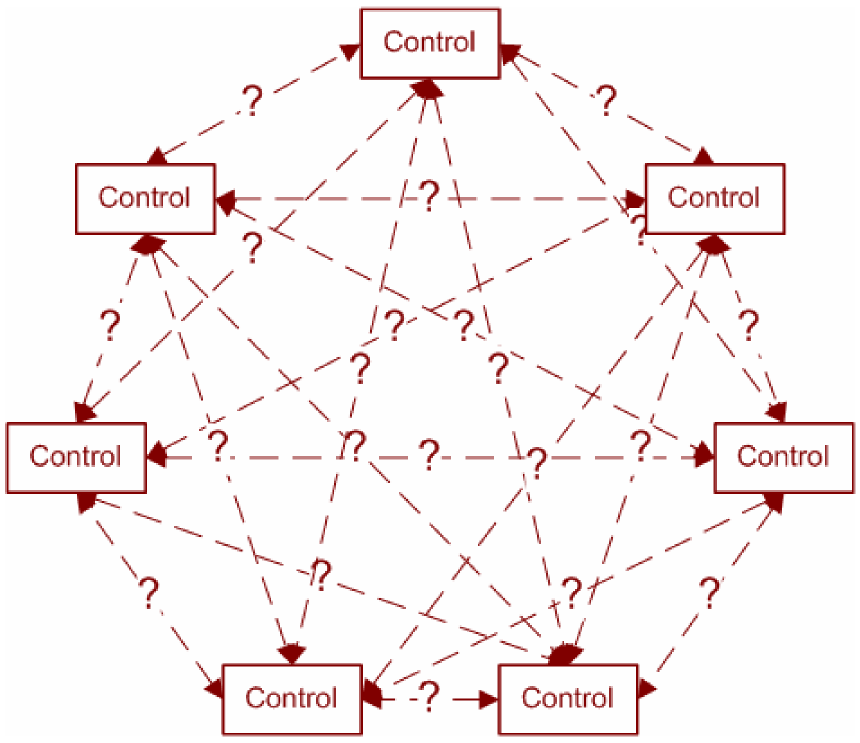
We would then calculate the N-wise testing using a value of N+1 as an input to the test-generation tool, and include the number of unique sequences as if it were a control input.

Unfortunately, we don't have a solver capable of handling single dimensions larger than 52. This limits our ability to create a test suite for N=3 to a maximum of 7 controls.

To show the impact of sequencing on the test suite, consider an interface with 7 controls, each having 5 possible values. N=3 would require 236 tests if order is irrelevant. We then include sequence of selection as a parameter (by adding an 8th control with 35 possible values, and testing for N=4), In this case, N=3 (with sequencing) requires 8,442 scripts. Our theoretical lower bound would be 236 * 35 = 8260.

How to Make it Even Better

When we don't know anything, or don't apply any knowledge about our application to our testing strategy, we end up with far too many tests. By applying knowledge of the application to our test design we can greatly reduce the size of our test suite. Tests that incorporate knowledge of the application being tested are known as whitebox tests[6].



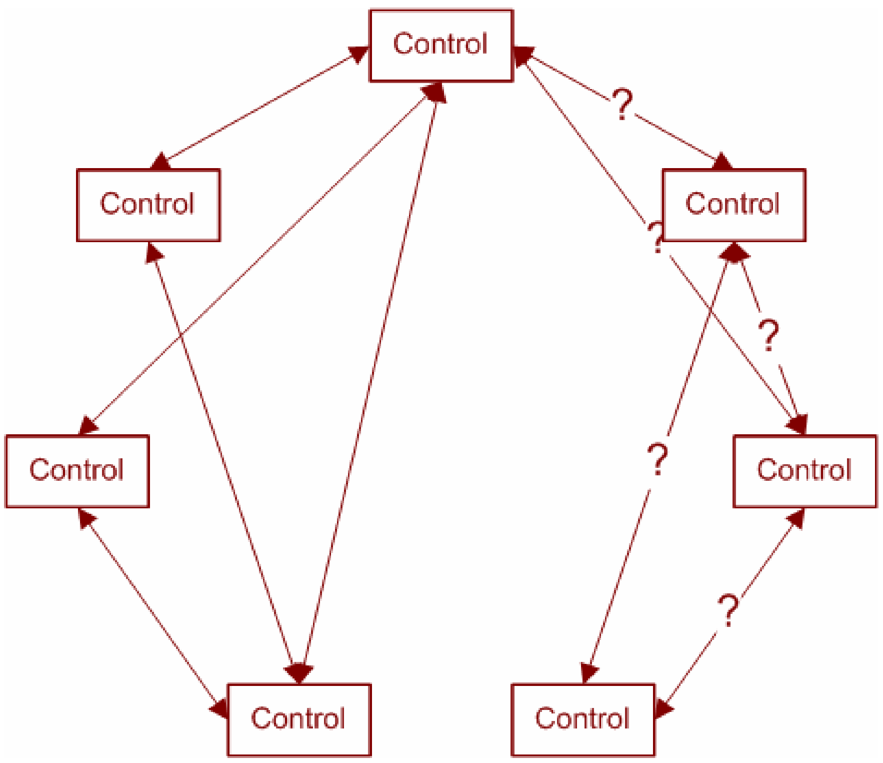
[6] http://tynerblain.com/blog/2006/01/13/software-testing-series-black-box-vs-white-box-testing/

MAP OUT THE CONTROL DEPENDENCIES

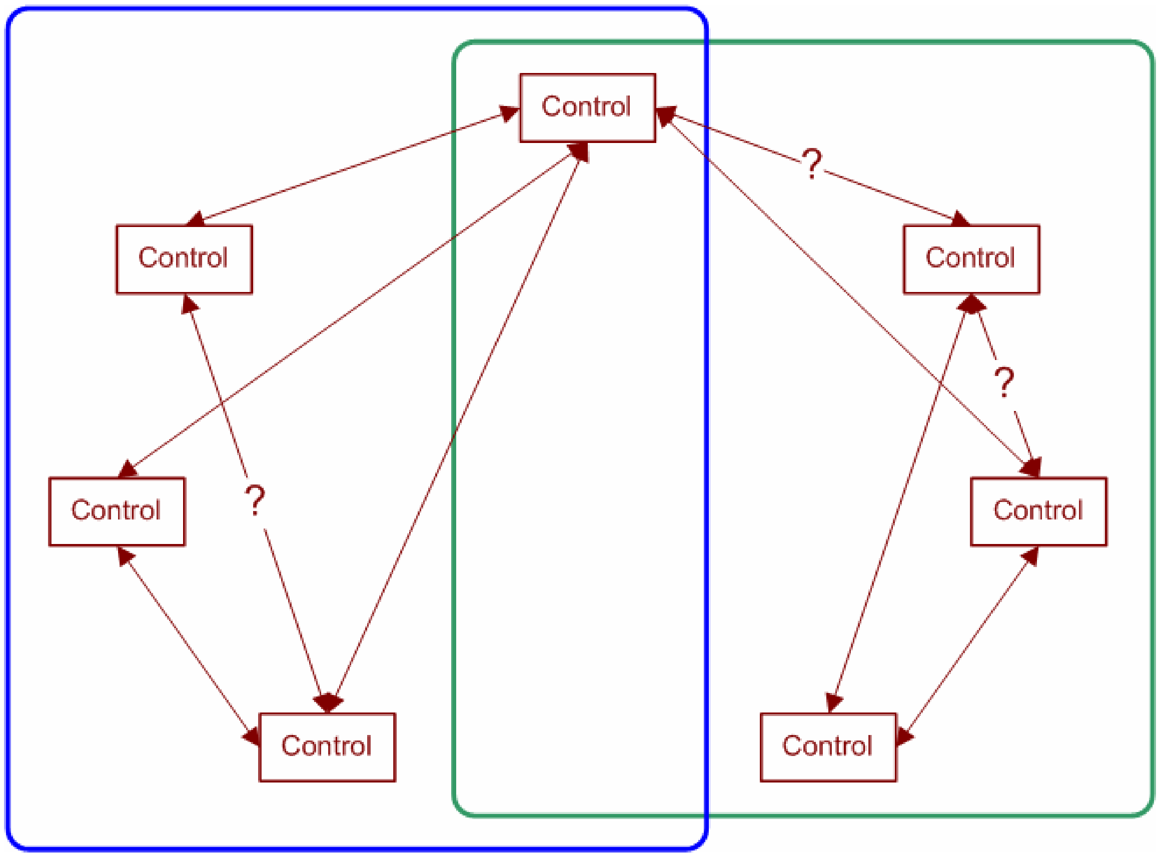
In our previous examples, we applied no knowledge of the interactions of controls, or the interactions within the program of having made selections in the controls. If we consider a visual map of the controls and their possible relationships, it would look like the following diagram.

There is a possibly-relevant connection between the selections in every pair of controls. We have designed our testing around the lack of knowledge that is clearly visible in the diagram.

It is likely that we can rule out some of the dependencies, but possibly not all of them. Our approach should be conservative; only remove those dependencies that we know don't exist. This knowledge comes from an understanding of the underlying application. Once we remove these links the diagram will look like this:



This clarified mapping allows us to reduce the size of our test suite dramatically, because we've identified the independence of many controls. In an ideal case, the result will be two or more completely disconnected graphs, and we can build a set of tests for our suite around each separate graph. As the diagram above shows, we do not have two completely independent graphs. We can take a testing approach as shown in the following diagram:



We've grouped all of the controls on the left in a blue box. These controls will be used with the N-wise generation tool to create a set of tests. The grouping of controls on the right will also be used to generate a set of tests.

In this example, we reduce the number of tests required by a significant amount when order matters.

Number of unique sequences for given N-wise run versus # of controls				
N	7 controls		4 controls + 4 controls	
	order independent	order matters	order independent	order matters
3	236	8,260	306	1,440

Also note that we increase the number of tests required when order doesn't matter, if we have any overlapping controls (if the graphs can't be separated). When the graphs can be separated, this reduces the amount of testing even if order is irrelevant.

The key to separating the graphs is to make sure that all controls only connect to other controls within their region (including the overlapping region).

Eliminate Equivalent Values from the Inputs

When we know how the code is implemented, or have insights into the requirements, we can further reduce the scope of testing by eliminating equivalent values. Consider the following example requirements for an application:

Requirements
Silver status is available to accounts with 10 or more orders
Gold status is available to accounts with 100 or more orders
Platinum status is available to accounts with 500 or more orders
Accounts are set to the highest status for which they qualify

The next table shows two variables that we are evaluating in our testing–imagine that they are controls in a user interface (or values imported from an external system).

All values for two controls	
# Orders	**Account Status**
1-9	Platinum
10-49	Gold
50-99	Silver
100-499	
500-999	
1000+	

If we did a pairwise test suite without knowledge of the requirements, we would have 18 tests to evaluate. We get 18 tests by finding all of the unique combinations of the two controls (6 order-quantity values * 3 account status values = 18 combinations). However, with knowledge of the requirements, we can identify that some of the values are equivalent. The highlighted regions represent equivalent values (with respect to the requirements).

All values for two controls	
# Orders	**Account Status**
1-9	Platinum
10-49	Gold
50-99	Silver
100-499	
500-999	
1000+	

Which we can collapse for testing purposes into:

All values for two controls	
# Orders	**Account Status**
1-9	Platinum
10-99	Gold
100-499	Silver
500+	

This consolidation of *equivalent* values reduces the number of tests we need to run. For our simple pairwise test, we reduce the number from 18 to 12. The number is reduced because now we have 4 order-quantity values * 3 account status values = 12 combinations. When there are more controls involved, and when we are doing N-wise testing with N=3, the impact is much more significant.

Conclusion

When we're testing any software, we are faced with the tradeoff of cost and benefit of testing. With complex software, the costs of testing can grow faster than the benefits of testing. If we apply techniques like the ones in this article, we can dramatically reduce the cost of testing our software. This is what we mean when we say test smarter, not harder.

Summarizing the techniques covered in this article:

- We can test very complex software without doing exhaustive testing.
- Random sampling is a common technique, but falls short of high quality goals–very good quality requires very high quantities of tests.
- Pairwise testing allows us to test very complex software with a small number of tests, and reasonable (on the order of 90%) code coverage. This also falls short of high-quality goals, but is very effective for lower expectations.
- N-wise testing with N=3 provides high quality capable test suites, but at the expense of larger suites. When the order of inputs into the software matters, N-wise approaches become limited in the number of variables they can support (fewer than 10), due to limitations of test-generation tools available today.
- We can apply knowledge of the underlying software and requirements to improve our testing strategy. None of the previous techniques require knowledge of the application, and thus rely on brute force to assure coverage. This approach results in conceptually redundant tests in the suite. By mapping out the grid of interdependency between inputs and subdividing the testing into multiple areas we reduce the number of tests in our suite. By removing redundant or equivalent values

from the test suite we also reduce the number of tests required to achieve high quality.

Testing smarter, not harder.

###

Continued Reading

The following articles from the author's Tyner Blain blog were referenced in this article.

"Software Testing Series: Measuring the Cost of Quality"
http://tynerblain.com/blog/2006/02/22/software-testing-series-measuring-the-cost-of-quality/

"Definition of Expected Value"
http://tynerblain.com/blog/2006/02/03/definition-of-expected-value/

"Software Testing Series: Pairwise Testing"
http://tynerblain.com/blog/2006/03/18/software-testing-series-pairwise-testing/

"Foundation Series: Black Box and White Box Software Testing"
http://tynerblain.com/blog/2006/01/12/foundation-series-black-box-and-white-box-software-testing/

"Software Testing Series: Black Box vs White Box Testing"
http://tynerblain.com/blog/2006/01/13/software-testing-series-black-box-vs-white-box-testing/

About the Author

Scott Sehlhorst has built nine years of software experience on a foundation of eight years as a mechanical engineer. He's worked as a consultant, developer, and technical preseller. Scott also operates the Tyner Blain blog (`tynerblain.com\blog`), and Tyner Blain LLC, which provides product management and process improvement services.