Science-based Test Case Design - Better Coverage, Fewer Tests

by Gary Gack, President, Process-Fusion.net
and Justin Hunter, CEO, Hexawise

Draft 4 - 5/13/2010

Synopsis

This chapter describes an approach to test case design using a proven statistical method known as Design of Experiments. Benefits and costs quantified by a study comparing effectiveness of a manual test case design approach to effectiveness of DoE-based test case selection methods are described. In the interest of balance we discuss counter-arguments and limitations. We illustrate use of this method with examples and a case study.

Design of Experiments (DoE) – an Introduction

DoE is by no means a new idea. Indeed, according to Wikipedia, an early example occurred in the British navy in the 18th century.

> "In 1747, while serving as surgeon on HM Bark *Salisbury*, James Lind carried out a controlled experiment to develop a cure for scurvy. Lind selected 12 men from the ship, all suffering from scurvy, and divided them into six pairs, giving each group different additions to their basic diet for a period of two weeks. The treatments were all remedies that had been proposed at one time or another. They were:
>
> - A quart of cider every day
> - Twenty five gutts (drops) of *elixir vitriol* (sulphuric acid) three times a day upon an empty stomach,
> - One half-pint of seawater every day
> - A mixture of garlic, mustard, and horseradish in a lump the size of a nutmeg
> - Two spoonfuls of vinegar three times a day
> - Two oranges and one lemon every day.
>
> The men who had been given citrus fruits recovered dramatically within a week. One of them returned to duty after 6 days and the other became nurse to the rest. The others experienced some improvement, but nothing was comparable to the citrus fruits, which were proved to be substantially superior to the other treatments."
> http://en.wikipedia.org/wiki/Design_of_experiments#History_of_development

As illustrated by Lind's application to scurvy, DoE is a "controlled experiment" that focuses on pairs (in this example pairs are defined by treatment and subject). In other examples pairings may involve 3, 4, or "n-way" combinations of factors. This method has been widely applied over the years to drug testing (e.g., clinical trials), engineering design (e.g., optimal wing design), and many other areas. More recently this approach has been adapted to software test case design and embodied in easy to use tools. The statistical methods and algorithms underlying tools that implement DoE are outside the scope of this article – knowledge of those methods is not required to use DoE tools

effectively. Think of DoE-based test case design tools as a “black box” – given appropriate inputs they generate optimized test cases – knowledge of internal workings is not required.

Benefits of DoE in Software Test Case Design

In essence, application of DoE to software test case design provides three principal benefits – first, it guarantees any desired (user specified) level of “coverage” of factor interactions – e.g., it ensures all 2-way, 3-way, or n-way interactions are included in the set of test cases generated by the method. In complex test situations, such as those found in many software systems, this method will always ensure full n-way coverage of huge numbers of potential interactions.

Second, it ensures coverage is achieved with a “minimum” number of tests, which means test execution effort will necessarily be reduced. DoE tools do not necessarily generate the absolute theoretical minimum number of cases as that may require excessive compute time, but they always come quite close and result in dramatically fewer test cases than are typically created by manual methods. A recent article[1] in IEEE Computer magazine reported the results of a 10-project empirical comparison of DoE based testing to “business as usual” methods. On average the DoE-based method found 2.4x more defects per tester hour, and 13% more defects in total.

Consider a real world test problem[2].

- 60 parameters
- each with 2 – 5 value choices that trigger business rules.

... would require 1,746,756,896,558,880,852,541,440 tests if we wanted to test all possible combinations of the inputs we’ve identified. To put that into perspective, at a rate of one test case per second, 6 billion people could execute “only” 189,216,000,000,000,000 tests in one year.

It takes just 36 tests to achieve 100% 2-way coverage

Achieving that with conventional (manual) test design approaches is extremely difficult and almost never certain.

Third, because the process used to generate necessary inputs is highly structured and repeatable, the effort necessary to provide the required inputs and generate the test cases is significantly less than that typically needed to create test cases manually. Pilot projects conducted by a major outsourcing firm on average realized 30-40% reduction in test planning and execution effort. Larger gains are expected with future releases of

software systems under test as the DoE test planning process leads to highly reusable test inputs.

Why Factor Interactions Matter

The IEEE article mentioned above[1] reports results of several retrospective studies of the characteristics of software defects that "escaped" into systems released to customers.

Defect "trigger"	Cumulative % of Defects		
	Minimum	Maximum	Average
Single parameter value	30%	70%	60%
2 parameter values	70%	95%	86%
3 parameter values	88%	99%	91%

Many fewer defects arise from 4 or 5 way interactions, and 6-way interaction defects are exceptionally rare. The key take-away: ***over 90% of all software defects are, on average, triggered by interactions between 3 or fewer parameter values***. Hence any sound test plan should guarantee at least 100% coverage of 2-way interactions. Applications with high reliability requirements, such as medical devices or avionics, should guarantee a minimum of 100% coverage of 3-way interactions! That is extremely difficult to achieve without DoE-based test case design tools.

Definitions – Pairwise, Combinatorial, Orthogonal

These terms are often used interchangeably. Pairwise implies only 2-way combinations, whereas combinatorial suggests n-way combinations. An Orthogonal Array has the balancing property that, for each pair of columns, all parameter-level combinations occur an equal number of times. This is a fine distinction vis-a-vis 2-way and will in practice generate slightly more test cases.

A Simple 2-way Combinatorial Example

Given these parameters and (values):

OpSys (XP, W7, Unix, Mac); Browser (Firefox, Opera); User Type (Admin, General)

We can ensure 100% 2-way coverage with 8 tests as illustrated here. 100% 3-way coverage in this example would require 16 tests. As the number of parameters and values grows the advantage of this approach grows exponentially as we will see in upcoming examples.

	OpSys	Browser	UserType
1	XP	Opera	Admin
2	XP	Firefox	General
3	W7	Opera	General
4	W7	Firefox	Admin
5	Unix	Opera	Admin
6	Unix	Firefox	General
7	Mac	Opera	Admin
8	Mac	Firefox	General

Counter Arguments and Limitations

James Bach has taken issue with designating DoE-based / combinatorial methods as "best practices"[3]. He points out, we think quite rightly, that this method is not a silver bullet sufficient to slay every werewolf hiding in the code. Michael Bolton[4], while generally positive, has also expressed reservations. However, most experts will agree that it is at least a "good practice" that has an important place in many testing scenarios. Lead bullets, which DoE-based methods certainly are, will kill most of the more common sorts of wolves more surely than will bows and arrows. We believe this method merits far wider adoption than currently is the case.

Specific limitations of DoE-based methods include the following:

1. As with any approach to test case design you need the input of intelligent and thoughtful test designers. Using DoE based methods will not turn incompetent, inexperienced testers into prodigious bug-killers.
2. Testers can and will forget to include significant test inputs. If a test designer fails to include inputs that should be tested, tool generated test cases will also fail to include them.
3. You should not rely exclusively on DoE-based methods as a single approach for all of your test cases. These tools may well generate the vast majority of tests that should be run on an application, but you should consult Subject Matter Experts to see if they would recommend supplementing with additional tests.
4. DoE-based methods are generally not appropriate for unit or white box testing. The degree of benefit you can expect rises dramatically within scope and complexity of the application under test.
5. DoE-based methods are generally not well suited to find "race" conditions in multi-processor configurations.
6. DoE-based methods won't necessarily find sequential dependencies – e.g., a failure occurs only when a certain combination of inputs occurs in a specific sequence (we find it only by chance).
7. DoE-based methods (and other methods as well) won't necessarily find unexpected side effects – e.g., when testing Word a combination of configuration parameters appear to achieve the expected result, but lead to document corruption.

The general characteristics of situations in which these methods are more or less likely to be cost-effective are illustrated by the following:

Testing Challenge	Larger Benefit	Lesser Benefit
Constraints	Simple Constraints handled by "invalid pairs" – e.g., *hardware = Mac + Browser = IE8*	Complex Constraints e.g., *if P1=B and P2=C then P3 cannot be x or y unless …*
Branches	Steps in a process flow are independent – i.e., *choices*	Many different branches – *may require a series of*

	in each step do not influence later selections	*different combinatorial test plans, rather than a single plan*
Number of parameters and values	Many parameters, relatively few values for each *Equivalence classes may be used to reduce length of the values list*	Few parameters, each with many essential-to-test values (few equivalence classes) e.g., *two parameters (state and product) to calculate shipping charges – 50 states x 20 products = minimum 1,000 2-way tests*

As with any approach to testing, limited experience and limited analytical skills in the test team present severe challenges.

Levels of Abstraction

DoE-based methods can be applied at a high level of abstraction in the early stages of testing to "explore" the application. One might think of this level as very similar to exploratory testing strategies advocated by a number of experts. The underlying thought process is quite similar – we think about what the application does at a very high level and we take a test drive through the more common paths the system under test may support. Let's explore this conceptually (actual navigation is not shown).

Often we begin by enumerating any relevant configuration options – e.g., PC and Mac; Firefox, IE 6-7-8, and Chrome. Next we think about the major functions of the system - e.g., in Amazon we can find a book or a CD, we can add or remove items to our cart, and we can checkout or cancel. If we wanted to explore all potential combinations of this limited set of parameters we would need 2*5*2*2*2 = 80 tests. That might be a bit more tests than we want to execute simply to explore. Alternately we can explore every potential 2-way pairing (e.g., PC+Chrome+find book+add item+checkout) with only 10 tests. When we do this we learn more about the application, we find things we want to explore more fully, and we get an initial 'smoke test' that may reveal any glaring problem areas (e.g., we find can't search successfully with Mac/Chrome). The 10 tests generated are the following:

Table 1: "Exploratory" Test Cases

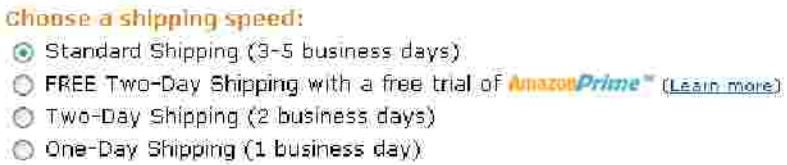
Hardware	Browser	Find	Cart	Finish
PC	Firefox	books	Add item	checkout
Mac	Firefox	CDs	Delete item	cancel
PC	IE6	CDs	Add item	cancel
Mac	IE6	books	Delete item	checkout
PC	IE7	books	Delete item	cancel

Mac	IE7	CDs	Add item	checkout
PC	IE8	books	Add item	checkout
Mac	IE8	CDs	Delete item	cancel
PC	Chrome	books	Add item	checkout
Mac	Chrome	CDs	Delete item	cancel

Using DoE at this level of abstraction differs from free-form exploration in that it involves a degree of formalization of our exploratory venture – we make a simple but explicit map of the territory we plan to explore. We might think of it as “structured exploration”. This has several advantages over a less-formal approach: (1) if we are interrupted we need not rely on memory to pick up where we left off (2) we cover a lot of ground with a minimum of effort, and we know exactly what has and has not been explored.

Often, exploratory testing leads us to recognize areas of the applications that are most “interesting” and deserving of a deeper dive. Taking this illustration one step farther, we might decide we want to test the checkout function more fully. To do that will need to navigate the following screens. In the sign-in screen we’ll assume we’re an existing customer, and we want to ship to the same address we used last time.

Next we choose a shipping method:



We then select a payment method, in this example one already on file.

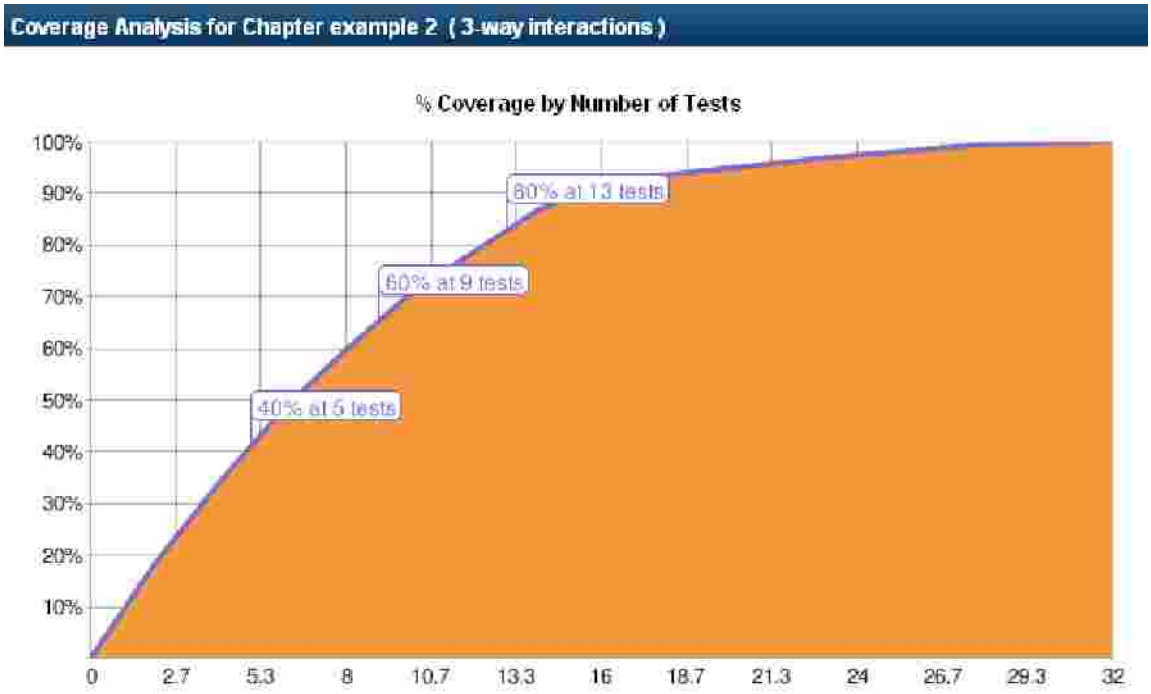
At this point we have several options – we can change the method of payment, we can change the shipping address, or we can place the order.

In the interest of brevity the above does not fully explore every possible branch but is perhaps sufficient to illustrate our point. This set of possible inputs (clearly only a small subset of the complete application) leads to 512 potential combinations if all possibilities are to be tested. 2-way coverage can be achieved in only 8 tests as illustrated below.

Table 2: "Deeper Dive" Test Cases

email address	password	Ship to	Ship method	payment method	change payment method	change billing address	place order
valid	valid	on file	standard	existing	yes	yes	yes
invalid	invalid	new	standard	new	no	no	cancel
valid	invalid	on file	free 2-day	new	yes	no	yes
invalid	valid	new	free 2-day	existing	no	yes	cancel
valid	valid	new	2-day	new	yes	yes	cancel
invalid	invalid	on file	2-day	existing	no	no	yes
valid	valid	on file	1-day	existing	no	no	cancel
invalid	invalid	new	1-day	new	yes	yes	yes

In this example 100% 3-way coverage can be achieved with 32 tests. The statistical methods used to generate these test cases are “smart” in that tests are generated in a sequence that leads to a large percentage of desired coverage with a comparatively small number of tests. As illustrated below, 80% coverage of 3-way combinations is achieved with only 13 tests. Obviously this is very valuable when schedule constraints force an early end to testing. In most cases the Pareto principle (the “80/20” rule) will apply – relatively few tests will find a disproportionate percentage of defects.



Combinatorial Coverage

When 2-way coverage is requested (as in Table 1 above) the test cases generated are guaranteed to provide 100% 2-way coverage but they also provide a certain percentage of higher order coverage. To illustrate that point, consider potential 3-way interactions among Hardware, Find, and Cart. As shown in table 3 below, 75% of 3-way interactions (6 of 8) are covered when we request 2-way coverage.

Hardware	Find	Cart	Test Case #
PC	Books	Add	1
PC	Books	Delete	5
PC	CDs	Add	3
PC	CDs	Delete	Not covered
Mac	Books	Add	Not covered
Mac	Books	Delete	4

Mac	CDs	Add	6
Mac	CDs	Delete	2

Similar analysis could be done in relation to interactions among other variables.

Case Study

Remittance Processing – Individual Retirement Account

126 test cases were identified using a "business as usual approach". An analysis of these cases identified four missing pairs – i.e., four instances of potential dual-mode defects were not tested in any of these 126 cases. The functionality being tested required 6 parameters, each with between 2 and 4 potential values as illustrated below:

IRA Remittance				
Plan (3)	Plan A	Plan B	Plan C	
Tax Year (2)	2009	2010		
Initial or Recurring (2)	Initial	Recurring		
Receipt Method (4)	Check	EFT	LBX	ACH-EFR
Bank (3)	"Bank A"	"Bank B"	"Bank C"	
Check Number (3)	Valid	Invalid	Blank	

Applying the DoE based method generated a total of 13 test cases to achieve 100% 2-way coverage or 48 tests to achieve 100% 3-way coverage.

The manually developed plan contained a great deal of redundancy and was hence quite inefficient – lots of wasted effort. The following comparison illustrates the extent of inefficiency in the manual plan.

Original Test Plan				
Test No	Plan	Tax Year	Initial or Recurring	Receipt Method
1	Plan A	2009	Initial	CHK
2	Plan A	2010	Initial	CHK
3	Plan A	2009	Initial	CHK
4	Plan A	2010	Initial	CHK
5	Plan A	2009	Initial	CHK
6	Plan A	2010	Initial	CHK
7	Plan A	2009	Initial	CHK

	Times Tested	
	Original Plan (out of 126)	Prioritized Plan (out of 13)
Potential single-mode fault: Plan A	43 X	5 X
Potential dual-mode fault: Plan A and Initial	21 X	2 X
Potential triple-mode fault: Plan A and Initial and CHK	7 X	1 X
Potential quadruple-mode fault: Plan A and Initial and CHK and 2009	4 X	1 X

When a test plan, such as this one, has 6 parameters each single test case can be decomposed into 15 potential dual mode defects being tested for. Efficient test plans will test for *as-yet-untested* 2-way defects with each new test case, whereas inefficient plans will test for the same potential defect again and again. DoE-based methods optimize plans generated to maximize efficiency.

In Summary

In our experience the DoE-based approach to software test case design has proven beneficial in every case we have encountered. It won't solve every conceivable test problem you will encounter, but it should certainly be one of the tools in every tester's bag of tricks. Give it a try – we think you'll like it!

To Learn More

The following articles and links provide additional perspectives on DoE-based test case selection methods.

1. Hexawise blog: http://hexawise.wordpress.com/
2. The Combinatorial Design Approach to Automatic Test Generation (published in IEEE Software September 1996, pp. 83-87): http://www.argreenhouse.com/papers/gcp/AETGissre96.shtml
3. All-pairs Testing: http://en.wikipedia.org/wiki/All-pairs_testing
4. Orthogonally Speaking (Elfriede Dustin): http://www.stickyminds.com/getfile.asp?ot=XML&id=5031&fn=Smzr1XDD3130filelistfilename1.pdf

5. Planning Efficient Software Tests (Madhav Phadke): http://www.stsc.hill.af.mil/crosstalk/1997/10/planning.asp

---

[1] Kuhn, Kacker, Lei, and Hunter - "Combinatorial Software Testing", *IEEE Computer, August, 2009* http://csrc.nist.gov/groups/SNS/acts/documents/kuhn-kacker-lei-hunter09.pdf

[2] http://hexawise.com/examples/telecom_example.pdf

[3] James Bach http://www.testingeducation.org/wtst5/PairwisePNSQC2004.pdf

[4] Michel Bolton http://www.developsense.com/pairwiseTesting.html