Teaching Domain Testing: A Status Report

Cem Kaner, J.D., Ph.D.
Florida Institute of Technology
150 West University Blvd.
Melbourne, FL 32901
321-674-7137
kaner@kaner.com

ABSTRACT

Domain testing is a stratified sampling strategy for choosing a few test cases from the near infinity of candidate test cases. The strategy goes under several names, such as equivalence partitioning, boundary analysis, and category partitioning. This paper describes a risk-focused interpretation of domain testing and some results (experiential and experimental) of helping students learn this approach.

1. INTRODUCTION

Domain testing [6, 7, 22, 23, 26, 28] is a stratified sampling strategy for choosing a few test cases from the near infinity of candidate test cases [17]. The strategy goes under several names, such as equivalence partitioning, boundary analysis, and category partitioning.

Domain testing is probably the most widely described, and one of the most widely practiced, software testing techniques. For example, according to Richard Craig, a well known teacher and consultant to testers in industry, "equivalence partitioning is … intuitively used by virtually every tester we've ever met."[9, p. 162]

Domain testing seems easy to explain and teach but students can *appear* knowledgeable long before they achieve competence. This article provides an overview of the nature of domain testing and then discusses some of the challenges and approaches to teaching the technique.

2. THE NATURE OF DOMAIN TESTING

The essence of domain testing is that we *partition* a *domain* into subdomains (*equivalence classes*) and then select *representatives* of each subdomain for our tests. I'll expand on each of these points below.

2.1 Scope of the Technique

Some authors restrict their consideration of the scope of this technique to linearizable input variables to mathematical functions [*e.g.,* 1, 7, 28]. A linearizable variable is one whose values can be mapped to a number line. The analysis is simpler and more obvious in these cases.

Some authors relax these restrictions. For example, Ostrand & Balcer [23] describe a general method for any type of input parameter or environmental condition associated with any function. Kaner et al. [20] describe how domain testing is applied in industry to compatibility testing, working through printer compatibility testing in detail. This variable's values (types of printer) cannot be ordered and a particular printer might be a member of more than one class.

2.2 Partitioning

Partitioning a set normally means splitting it into nonoverlapping subsets. Disjunction is an important attribute of some mathematical models associated with domain testing, but some authors [26, 20] note that practitioners work with overlapping sets.

2.3 Domains

Domain testing is a type of functional testing [13, 14]: we view the program as a function and test it by feeding it interesting inputs and evaluating its outputs. A function has an input domain (the set of all possible values that could be input to the program) and an output domain. Many of the discussions focus on input domains [1, 2, 15, 23, 28] but essentially the same analysis can be done on output domains [5, 20]. Along with considering traditional outputs (such as reports or screen displays), it can be useful to consider output to devices and to the file system [27].

2.4 Equivalence Classes

How we define *equivalence class* carries implications for theoretical interpretation and practical application of domain testing. In principle, the definition is straightforward: "All elements within an equivalence class are essentially the same for the purpose of testing" [23, p. 676]. However, the notion of "essentially the same" has evolved in different ways.

- ***Intuitive equivalence:*** Descriptions intended for practitioners often define equivalence intuitively, relying primarily on examples. [e.g. 9, 21, 25]. For example, "Basically we are identifying inputs that are treated the same way by the system and produce the same results" [9, p. 162] and "This concept, in which different input values yield similar results, is known as equivalence class partitioning" [25, p. 88]

  Along with providing a loose definition and an example, these books and courses often list some heuristics for identifying easy-to-analyze variables and picking appropriate test cases [5,20,21,22].

  When I worked as a testing supervisor, test manager, or consultant, I gained experience with testers who were trained this way (sometimes by me, sometimes by other teachers, peers or books). My impression is that this training is too vague. Testers trained this way often don't know how to generalize their knowledge to handle simultaneous testing of several variables. They also don't have much guidance as to how to find (or abstract out) the necessary information to partition variables if they don't have it in an immediately understandable form. Additionally, they may not generalize well to cases not directly covered by the heuristic examples. Finally, I often see examples of inclusion of unnecessary cases. For example, to test an integer input field that accepts values from 0 to 99, it's most common to test –1, 0, 99 and 100. Testers who are uncomfortable with domain testing or don't understand it often add such cases as 1, 2, 97, 98, and sometimes also –2 and 102.
- ***Specified equivalence:*** Another approach commonly presented to practitioners defines equivalence in terms of the specification. Variables have values that are valid (according to the specification) or invalid. The tester's task is to choose cases (typically boundaries) from the valid set and from each of the invalid sets. [5,10,22]. As

with the intuitive approach, this guidance is often accompanied by detailed examples and heuristics for identifying likely variables and selecting test cases.

Kaner, Falk & Nguyen [20] point out that in commercial software, detailed specifications are often not developed. In my experience, testers trained to design tests primarily from specifications often find it hard to adapt to less formal environments. Particularly troubling is the advice given to practitioners (usually at conferences or by short-term (aka drive-by) consultants) that they should refuse to test until they obtain a specification that provides sufficient equivalence class and boundary information. In companies that choose not to produce such documents, the tester who insists on them often loses credibility and goodwill without gaining much documentation.

- ***Analysis of the variables:*** Both the intuitive approach and the specified-equivalence approach focus on the program's variables, primarily from a black box perspective. Some authors [*e.g.* 13] take this further and suggest ways to partition data or identify boundary values by reading the code.
- ***Same path:*** Descriptions that appear primarily in the research literature often say that two values of a variable are equivalent if they cause the program to take the same branch or the same (sub-)path. [1,7,28]

  In practice, this viewpoint makes sense if the tester is also the author of the code or is intimately familiar with the source code. For a black box tester, this approach is uninspiring because application of it depends on information the tester doesn't have available.
- ***Subjective equivalence:*** Kaner [16] concluded that equivalence is in the eye of the tester. "If you expect the same result from two tests, you consider them equivalent. A group of tests forms an equivalence class if you believe that: they all test the same thing; if one test catches a bug, the others probably will, too; and if one test doesn't catch a bug, the others probably won't either. … Two people analyzing a program will come up with a different list of equivalence classes. This is a subjective process. It pays to look for all the classes you can find. This will help you select tests and avoid wasting time repeating what is virtually the same test." Kaner also presented heuristics for identifying equivalence classes, essentially an expansion of Myers' list [22].

  In my experience, the subjective approach [17, 18] poses the same teaching and training problems as intuitive equivalence.
- ***Risk-based equivalence***: Several of the early authors identified domain testing as a method that targeted specific types of errors [4,6,28]. Whittaker describes domain testing as an *attack* (essentially a software testing pattern) that arises out of the need to test for a class of commonly occurring faults.

  Myers opened a different door for risk analysis in his proposed solution to a classic example, *the triangle problem*. In the triangle problem, the computer reads three inputs which are supposed to be sides of a triangle. It is to evaluate them as scalene, isosceles, equilateral, or not a triangle. Myers' answer included test cases with non-numeric input and the wrong number of inputs.

  In a similar vein, Binder [2] listed 65 tests for the triangle problem, addressing several new dimensions of risk, such as potential errors arising if you try to repeat the test more than once.

  Collard [8] listed 4 tests that might be sufficient. He noted Kent Beck's claim that 6 tests would suffice for his implementation of the triangle program. Contrasting with Jorgensen's [15] list of 140 tests, Collard [8, p. 4] said, "This sounds like a bad joke – ask four different experts, get four different answers which range from four to 140 test cases – and four consulting bills."

  Collard proceeds from this contrast to teach a risk-based approach to domain testing. He phrases this in terms of his students' assumptions, pushing them to introspect about the assumptions they're making when they analyze the problem. For each assumption, he asks what tests you could design to explore the program's behavior if that assumption is in fact not true. From these possible tests, the tester should pick those that seem plausible

  The key to understanding the wide variation of responses to the triangle problem lies in recognizing that many of the tests have nothing to do with mishandling of a numeric input, such as taking the wrong path because the programmer mistyped an inequality. Instead, they lie on different dimensions from the number line that shows which number inputs are too large or too small. Each of these dimensions is subject to a different equivalence analysis. Here are three examples:

  (a) The valid number of sides you can enter for the triangle is three. Two and four sides are members of the too-few-sides and too-many-sides equivalence classes.

  (b) If input is ASCII encoded, the valid range (digits) runs from 48 ("0") to 57 ("9"). Characters 47 ("/") and 58 (":") are the interesting boundary values for non-digit input.

  (c) If input is interactive, a long delay between entry of two numbers might trigger a timeout failure. A delay just briefer than the timeout limit and one just longer than the limit lie in different equivalence classes.

  Consider the following three test cases:

  (i) Side 1 = 1, side 2 = 2, side 3 = 3, delay between entries = 1 second

  (ii) Side 1 = 2, side 2 = 2, side 3 = 3, delay between entries = 1 hour

  (iii) Side 1 = <space>2, side 2 = 2, side 3 = 3, delay between entries = 1 second.

  Cases (i) and (ii) are equivalent with respect to risks that the program won't process digits correctly or won't accept three entries as the right number, but not equivalent relative to the risk that the program (i) would accept three sides that don't define a triangle or (ii) won't recognize the inputs because of timeout. Similarly, cases (ii) and (iii) are equivalent with respect to the size of the numbers entered, but in different classes relative to the risk that the program will not sensibly interpret a leading space.

In this paper, I adopt a subjective, risk-based definition of equivalence:

- Two tests are equivalent, relative to a potential error, if both should be error-revealing (both could trigger the error) or neither should be revealing.

- The same test might be error-revealing relative to one potential error and not revealing relative to another.
- Given the same variable under analysis, two testers are likely to imagine different lists of potential errors because of their different prior experiences.

See Kaner & Bach [18] for an example of teaching materials that adopt this approach.

2.5 Selecting Representatives

We might expect that any member of an equivalence class should be as good as any other member, in the sense that if one member of the class would reveal an error, so should the others. [12,22,23] However, random sampling from equivalence class members isn't a very effective way to test [12]. When dealing with numeric input fields, programs tend to fail more at boundaries because some errors (off-by-one errors) are specific to the boundaries.

Boundary values are representatives of the equivalence classes we sample them from. However, they are better representatives than other members of their class because they are more likely to reveal an error.

Several discussions of domain testing stress optimal choice of test values within equivalence classes. [1,20,21,25]

"A *best representative* of an equivalence class is a value that is at least as likely as any other value in the class to expose an error." [18, p. 37]

The generalization of boundary cases to best representatives is helpful to support domain analysis of nonordered spaces, such as printers. Imagine testing a mass-market program's compatibility with printers on the market. Failure to operate with any printer can result in technical support costs even if the underlying error is more fairly attributed to an oddity in the printer than to an error in the software under test. If the projected costs are high enough, the publisher is likely to revise the program to deal with this special case [20]. There are so many types of printers that it is impractical to test all of them. Applying a stratified sampling approach, we might group as equivalent all 100 printer models advertised as Postscript Level 3 compatible. However, there is no smaller-to-larger ordering within or between groups of printers, so how should we decide which individual printers to test?

If one printer is more prone to memory management problems, it is more likely to fail tests that require lots of memory or run long sequences of tasks. For those types of tests, this printer might be a best representative. A second printer might be more prone to timing bugs that cause papers jams; this might be a best representative for tests that push the printer into multitasking.

2.6 Combination Testing

Tests often involve multiple variables. Even if we reduce the number of tests per variable, we face combinatorial explosion when we test them together. Most discussions of domain testing suggest mechanical ways to test variables together [1,7,14,27]. These can be useful, but in practice, testers have several opportunities to exercise judgment. For example,

- If a test combines more than two variables, an out-of-bounds value for one variable may cause the program to reject the test without even processing the other variables' values. If this is the only test that combines those particular values of those other variables, that combination has been effectively masked—not tested.
- A test might be made more powerful by combining out-of-bounds values for several variables, but only if the program will process the values of each variable. If the program checks each variable in turn and stops processing on the first bad value, the combination actually tests only one out-of-bounds value at a time.
- Some combinations are more interesting because they reflect common uses of the product. Others are based on previous failures.

3. TEACHING EXPERIENCES

Over the past ten years, I've taught several dozens of testing classes to commercial and academic students.

3.1 Basic Knowledge

I introduce students to domain testing through some very simple examples (such as the triangle problem [22] or adding two 2-digit numbers [20]). In the process of working through this main example, I provide further examples (stories) from industrial experience or examples (often presented as puzzles) that illustrate that we analyze variables along many different dimensions. I also map the example onto Myers' boundary analysis table. Presentation of this material is typically lecture format, with an in-class class brainstorming session. Academic students do a homework assignment that has them brainstorm further about Myers' triangle.

At this point in the course, commercial students tell me that they now understand domain testing and believe they can do it. Commercial and academic students can define equivalence classes and give a basic description of the method.

During my first two or three years of teaching this class, I believed that commercial students actually did understand this material well enough to apply it, but in later years, I gave students exercises (in-class for commercial students; assignments for academic students) and discovered that their knowledge was more superficial.

In Bloom's [3] terms, at this point, students appear to have Level 1 knowledge of some terminology, of some specific facts, of a way of organizing test-related information and of a methodology. That is, the student can describe them, but doesn't yet know how to apply them. Students can also read and explain a boundary analysis table, so in Bloom's scale, they are also operating at Level 2 (Comprehension).

3.2 Higher Level Knowledge

In the academic course, I give students one or two additional domain testing assignments. We apply all test techniques to a sample application, such as Open Office's word processor.

An assignment might require the student to pick a feature of the word processor, pick a variable associated with the feature, identify ten risks associated with the variable and, for each risk, create a test specifically for that risk that is designed to maximize the chance of finding a defect associated with this risk and explain what makes this a powerful test. The student is told to use boundary values and that in explaining power, "it might help you to compare [your test] to a less powerful alternative." The student reports results in a table (format specified in the assignment instructions).

Variants of this assignment might require the student to pick several variables or to identify a class of error-revealing tests associated with the risk.

In addition to the assignments, students receive a set of practice questions well before the exam. The exam questions are draw from the practice set. Students develop answers to these questions, comparing results in study groups. Here are two examples from the 2002 study guides:

- Imagine testing a file name field. For example, go to an Open File dialog, you can enter something into the file name field. Do a domain testing analysis: List a risk, equivalence classes appropriate to that risk, and best representatives of the equivalence classes. For each test case (use a best representative), briefly explain why this is a best representative. Keep doing this until you have listed 12 best-representative test cases.
- Imagine testing a date field. The field is of the form MM/DD/YYYY (2 digit month, 2 digit day, 4 digit year). Do an equivalence class analysis and identify the boundary tests that you would run in order to test the field. (Don't bother with non-numeric values for these fields.)
- I, J, and K are signed integers. The program calculates K = I / J. For this question, consider only cases in which you enter integer values into I and J. Do an equivalence class analysis from the point of view of the effects of I and J (jointly) on the variable K. Identify the boundary tests that you would run (the values you would enter into I and J).

Students gain skill and insight throughout the semester, but the following problems are common to most students at some point in the class; some students finish the course with them:

- ***Poor generalization.*** Students often pick inappropriate variables for analysis. For example, a binary variable can take on two values. You gain nothing by saying that each value is the sole member of an equivalence class and is the best representative of its class.
- ***Missing values.*** Students are initially likely to miss error-handling cases; most students do this well by end of term.
- ***Excess values.*** For example, in a numeric input field that accepts two-digit positive numbers, some students include 97 and 98 in their test set, along with 99 and 100. Most students seem to get past this.
- ***Failure to spot a boundary.*** In a field that can be treated as numeric, the student doesn't identify a boundary case.
- ***Failure to spot a dimension.*** For example, consider a field that is supposed to accept two-digit numbers. How many characters can we actually enter into this field?
- ***Failure to articulate a risk.*** Rather than state how the program might fail, the tester reiterates the test case in the risk field. Another failure is the excessively general statement, like "failure to process this value correctly."
- ***Failure to explain how a test case relates to a stated risk.*** When I ask for this type of explanation in an assignment, several students either leave this field blank or say something inarticulate or irrelevant. Many, but not all, students improve with feedback.

These errors all reflect problems in higher-order thinking about the domain testing task. In terms of Bloom's taxonomy:

- Following a standard procedure, requires Level 3 (Application) knowledge of domain testing.
- Figuring out (and explaining the choice of) boundary values within a given equivalence class is a Level 4 (Analysis) task.
- Identifying risks and error-revealing classes associated with them is a Level 6 (Evaluation) task.

Note that domain testing, as I expect students to learn it (risk-based) requires a deeper level of understanding (Bloom taxonomy's analysis and evaluation) than the primarily procedural approaches we often see [*e.g* in 15 and 22].

3.3 Padmanabhan's Experiment

Sowmya Padmanabhan's [24] thesis research involved developing a more effective set of teaching materials for domain testing. Her course mixed explanatory and procedural information, drilling students in several tasks, such as identifying variables and associated dimensions.

The final task in Padmanabhan's course was an open-book performance test. Students were presented with a screen shot of the Microsoft Powerpoint *Page Setup* dialog. They were to list the variables involved, provide an equivalence class table and combination tests. In her course, the equivalence table included five columns: Variable, Dimensions, Equivalence Classes, Test Cases and Risks. This is an *authentic performance*, in Wiggins' [29] sense, a realistic, meaningful task. Evaluation of the results is being done by three people who have significant experience testing, managing and hiring testers. Their standard for evaluation is the performance they would expect from a tester who had been trained in domain testing and has had a year's commercial testing experience.

I have seen Padmanabhan's data. The final scoring is not complete, but there are some obvious and noteworthy trends:

- These students' tables—especially the specific identification of dimensions along which data entered into the variable can vary—look unusually sophisticated for students who have no prior test experience or education and only 15 hours of instruction.
- The 23 students' analyses were remarkably consistent. Their tables were structured the same way and almost all identified the same dimensions. For example, for the Page Width variable (a floating point field that can range from 1 to 56 inches), students consistently identified three dimensions: page width (in inches), number of characters that can be entered into the field (from 0 to 32) and the allowed characters (digits and decimal point).
- The students were also remarkably consistent in what they missed. Examples: (a) if you enter a sufficiently large paper size, PowerPoint warns that "the current page size exceeds the printable area of the paper in the printer". No one identified a boundary related to this message. (b) when you change page dimensions, the display of slides on the screen and their font sizes (defaults and existing text on existing slides) change. No one appeared to notice this. And (c) change the page size to a large page and try a print preview. You only see part of the page.

Padmanabhan provided her students with detailed procedural documentation and checklists. From the consistency of the students' results, I infer that students followed the procedural documentation rather than relying on their own analyses. It's interesting to note that, assuming they are following the

procedures most applicable to the task at hand, the students not only include what the procedures appear to tell them to include, but they also miss the issues not addressed by the procedures. In other words, the students aren't doing a risk analysis or a dimensional analysis; they are following a detailed set of instructions for how to produce something that looks like a risk analysis or dimensional analysis.

I think we'll find Padmanabhan's materials (especially the practice exercises) useful for bringing students to a common procedural baseline. However, we have more work to do to bring students' understanding of domain testing up to Bloom's Level 6 (Evaluation) or Gagne's [11] development of cognitive strategies.

4. ACKNOWLEDGMENTS

This research was partially supported by NSF Grant EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers." Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

This is a draft paper, submitted to the Conference on Software Engineering Education & Training 2004.

I acknowledge the contributions of Jia Hui Liu, Sabrina Fay, Sowmya Padmanabhan, James Bach, and Pat McGee to the work reported in this paper.

5. REFERENCES

[1] Beizer, B. Black-Box Testing, John Wiley & Sons, 1995

[2] Binder, R.V. Testing Object-Oriented Systems, Addison-Wesley, 2000

[3] Bloom, B.S. Taxonomy of Educational Objectives: Book 1 Cognitive Domain, Longman, 1956

[4] Boyer, R.S., Elspas, B., & Levitt, K.N. SELECT—A formal system for testing and debugging computer programs by symbolic execution. Proceedings of the International Conference on Reliable Software, Los Angeles, CA, p. 234-245, April 1975

[5] Burnstein, I. Practical Software Testing, Springer, 2003

[6] Clarke, L.A. A system to generate test data and symbolically execute programs, IEEE Transactions on Software Engineering, v.2, p. 208-215, 1976

[7] Clarke, L.A., Hassel, J. & Richardson, D.J. A close look at domain testing, IEEE Transactions on Software Engineering, v.2, p. 380-390, 1982

[8] Collard, R. Developing Software Test Cases, book in preparation, manuscript draft February 16, 2003

[9] Craig, R.D. & Jaskiel, S.P. Systematic Software Testing, Artech House Publishers, 2002

[10] DeMillo, R.A., McCracken, W.M., Martin, R.J., & Passafuime, J.F. Software Testing & Evaluation, Benjamin/Cummings,1987

[11] Gagne, R.M. & Medsker, K.L The Conditions of Learning: Training Applications, Wadsworth, 1996

[12] Hamlet, R. & Taylor, R. Partition testing does not inspire confidence, Proceedings of the Second Workshop on Software Testing, Verification & Analysis, Banff, Canada, July 1988, pp. 206-215

[13] Howden, W.E. Functional testing and design abstractions, Journal of Systems & Software, v.1, p. 307-313, 1980

[14] Howden, W.E. Functional Program Testing & Analysis, McGraw-Hill, 1987

[15] Jorgensen, P.C. Software Testing: A Craftsman's Approach, 2d ed. CRC Press, 2002

[16] Kaner, C. Testing Computer Software, 1st ed., McGraw-Hill, 1988

[17] Kaner, C. The impossibility of complete testing, Software QA, v.4, #4, p. 28, 1997. http://www.kaner.com/pdfs/imposs.pdf

[18] Kaner, C. & Bach, J. Black Box Testing: Commercial Course Notes, 2003, http://www.testingeducation.org/coursenotes

[19] Kaner, C., Bach, J. & Pettichord, B. Lessons Learned in Software Testing, John Wiley & Sons, 2001

[20] Kaner, C., Falk, J., Nguyen, H.Q. Testing Computer Software, 2d ed. Van Nostrand Reinhold 1993, reprinted John Wiley & Sons, 1999

[21] Kit, E. Software Testing in the Real World, Addison-Wesley, 1995

[22] Myers, G.J. The Art of Software Testing, John Wiley & Sons, 1979

[23] Ostrand, T.J. & Balcer, M.J. The category-partition method for specifying and generating functional tests, Communications of the ACM, v.31 #6, p. 676-686, 1988

[24] Padmanabhan, S. Domain Testing: Divide and Conquer, M.Sc. thesis in Software Engineering, Florida Institute of Technology, in preparation. To be published at http://www.testingeducation.org/articles

[25] Tamres, L. Introducing Software Testing, Addison-Wesley, 2002

[26] Weyuker, E.J. & Jeng, B. Analyzing Partition Testing Strategies, IEEE Transactions on Software Engineering, v.17, p. 703-711, 1991

[27] Whittaker, J. How to Break Software, Addison-Wesley, 2002

[28] White, L.J., Cohen, E.I., & Zeil, S.J. A domain strategy for computer program testing, in B. Chandrasekaran & S. Radicchi (eds), Computer Program Testing, North Holland Publishing, Amsterdam, p 103-112, 1981

[29] Wiggins, G. Educative Assessment, Jossey-Bass, 1998