An introduction to the theory and practice of domain testing

VISTACON
HCMC, VIETNAM
September, 2010

Cem Kaner

Sowmya Padmanabhan

Overview

Domain testing is the most widely taught (and perhaps the most widely used) software testing technique.

Practitioners often study the simplest cases of domain testing under two other names, "boundary testing" and "equivalence class analysis." That simplified form applies

- Only to tests of input variables
- Only when tested at the system level
- Only when tested one at a time
- Only when tested in a very superficial way

In competent practice, and in the research literature, all of these limitations are often set aside, but there is very little practical teaching of the more general approaches.

Sowmya Padmanabhan did her M.Sc. thesis research on this. Sowmya, Doug Hoffman and I are co-authoring the ***Domain Testing Workbook*.**

This talk will present a worksheet that we've developed for planning and creating domain tests of individual variables or multiple variables that are independent or linearly or nonlinearly related. I'll brush on some of the theory underlying the approach, but mainly I want to present some of the lessons that this work brought home to me about skilled (contrasted with inexperienced or unskilled) domain testing.

Understanding domain testing

In domain testing,
we partition a domain
into sub-domains
(equivalence classes)
and then test using
values from each
sub-domain.

Notice how I format this

Variable	Valid case equivalence class	Invalid case equivalence class	Boundaries & special cases	Notes
X	0 – 100		0	
			100	
		< 0	-1	
		> 100	101	

- One line per test

Domain Testing

Variable	Valid case equivalence class	Invalid case equivalence class	Boundaries & special cases	Notes
X	0 – 100		0	
			100	
		< 0	-1	
		> 100	101	

- Each test maps clearly to a description of the class that it represents
- I start by focusing on classes directly tied to the variable's ***primary dimension***.
  - I'll talk more about primary and secondary dimensions later, but for now note that
    - my first-draft chart works ***on X's number line***,
    - There are no letters, no blank fields, no timeouts, no X-input-too-long overflows.

Domain Testing

Variable	Valid case equivalence class	Invalid case equivalence class	Boundaries & special cases	Notes
X		< 0	-1	
	0 – 100		0	
			100	
		> 100	101	

- I prefer "valid" values first, but some people prefer this order instead.

Boundary table as a test plan component

- Makes the reasoning obvious.
- Makes the relationships between test cases fairly obvious.
- Expected results are pretty obvious.
- Several tests on one page.
- Can delegate it and have tester check off what was done. Provides some limited opportunity for tracking.
- Not much room for status.

Building table like these (in practice)

- Relatively few programs will come to you with all fields fully specified. Therefore, you should expect to learn what variables exist and their definitions over time.
- To build an equivalence class analysis over time, put the information into a spreadsheet. Start by listing variables. Add information about them as you obtain it.
- The table should eventually contain all variables. This means, all input variables, all output variables, and any intermediate variables that you can observe.
- In practice, most tables that I've seen are incomplete. The best ones that I've seen list all the variables and add detail for critical variables.

These tables are limited

- Most of these tables consider only input variables
  - People don't buy programs to enter data
  - They buy programs to analyze data and give results
  - ***Therefore, we need to test <u>output</u> variables***
- These tables focus on individual variables
  - Individual variables are a start, but we need to test combinations of several variables together

Exercise 1

- S is an input string. It will hold someone's name
- You can enter letters, numbers or spaces into S
- The program doesn't care what S is but it cannot be more than 30 characters

Do an equivalence class / boundary analysis on S.

- Ignore secondary dimensions, for now

When you are done, please pass the chart to the person on your left, and grade the chart of the person on your right.

Exercise 2

- A, B, and C are integers.
- You can enter data into A and B, but not C
- C is calculated from A and B
- Do an equivalence class analysis on C, showing what values of A and B you need to test with, to test C.

- When you are done, please pass the chart to the person on your right, and grade the chart of the person on your left.

Research

Cem Kaner & Sowmya Padmanabhan, "Practice and transfer of learning in the teaching of software testing," *Conference on Software Engineering Education & Training*, Dublin, July 2007.

- http://www.kaner.com/pdfs/kanerPadmanabhanPractice.pdf
- http://www.kaner.com/pdfs/CSEETdomain2007.pdf (slides)

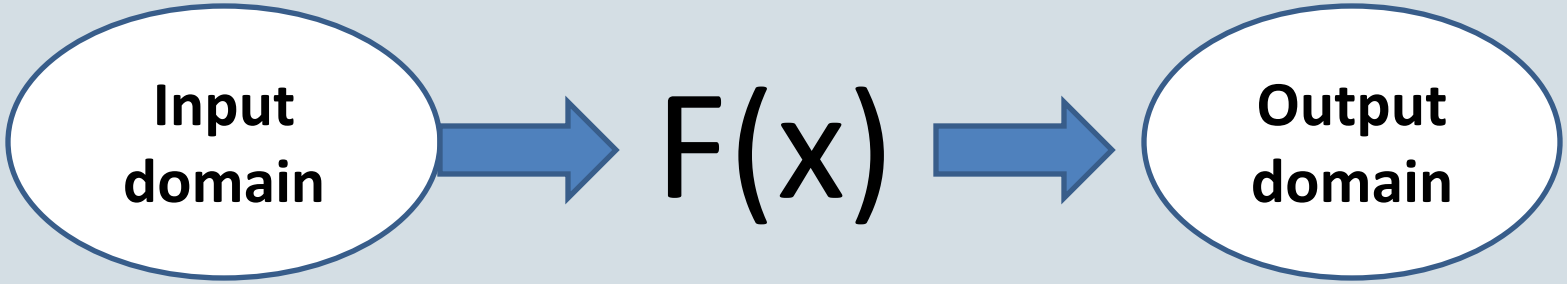
Padmanabhan, S. (2004). Domain Testing: Divide & Conquer. Unpublished M.Sc. Thesis., Florida Institute of Technology, Melbourne, FL. http://www.testingeducation.org/a/DTD&C.pdf.

- We tried teaching this technique via exemplars but had mixed results.
- We ultimately concluded that we needed to embed the exemplars in a stronger cognitive structure.

Four central questions

- What domain are we testing?
- How do we determine how to group values of a variable(s) into equivalence classes?
- How do we determine which members of each class to test?
- How do we determine whether the program passed or failed the test?

Domains



Functional testing:

We treat the program as a function that transforms inputs to outputs.

(Howden)

Domains

- Single-input tests check input filters:
  - Is this necessary at the system level?
  - Should we test the filter at the unit level?

- Output domain is more challenging

  K = I * J

  I, J, K are integers

  I, J are input variables. K is the output.

  Test K.

  We'll look at this example again later

Four central questions

- What domain are we testing?
- **How do we determine how to group values of a variable(s) into equivalence classes?**
- How do we determine which members of each class to test?
- How do we determine whether the program passed or failed the test?

Partitioning

How do we determine how to group variables into equivalence classes?

- Partitioning a set means splitting it into nonoverlapping subsets.
- Disjunction (nonoverlapping) is important for some models, but practitioners often work with overlapping sets (Kaner, et al., 1993; Weyuker & Jeng, 1991). For our practical purposes, partitioning means dividing a set of possible values of a variable into subsets that don't overlap (or not much).
- Partitioning usually splits a set into equivalence classes.

Partitioning

Intuitive or subjective equivalence: two test values are equivalent if they are so similar to each other that it seems pointless to test both.

Intuitive definitions ***(e.g., Craig & Jaskiel,*** 2002; Kit, 1995) appear obvious. For example,

- "Basically we are identifying inputs that are treated the same way by the system and produce the same results" Craig & Jaskiel (2002, p. 162).
- "If you expect the same result from two tests, you consider them equivalent. A group of tests forms an equivalence class if you believe they all test the same thing … Two people analyzing a program will come up with a different list of equivalence classes. This is a subjective process" Kaner (1988, p. 81).

Partitioning

Specified equivalence: two test values are equivalent if the specification says that the program handles them in the same way.

- Burnstein, 2003; DeMillo, McCracken, Martin, & Passafiume, 1987; Myers, 1979.
- Challenges:
  - Testers complain about missing specifications may spend enormous time writing specifications
  - Focus is on things that were specified, but there might be more bugs in the features that were un(der)specified

Partitioning

- ***Analysis of the code that defines or uses the variables:*** The intuitive and specified-equivalence approaches focus on the program's variables, primarily from a black box perspective. Some authors (e.g., Howden, 1980) suggest ways to partition data or identify boundary values by reading the code.
- ***Same path:*** Descriptions that appear primarily in the research literature say that two values of a variable are equivalent if they cause the program to take the same branch or the same (sub)path. (Beizer, 1995; Clarke, et al., 1982; White, 1981)

Partitioning

Equivalent Paths: two test values are equivalent if they would drive the program down the same path (e.g. execute the same branch of an IF). (Beizer, 1995; Clarke, et al., 1982; White, 1981)

- Tester should be a programmer
- Tester should design tests from the code
- Some authors claim that a complete domain test will yield a complete branch coverage.
- No basis for picking one member of the class over another.
- Two values might take program down same path but have very different subsequent effects (e.g. timeout or not timeout a subsequent program; or e.g. word processor's interpretation and output may be the same but may yield different interpretations / results from different printers.)

Partitioning

Risk-based equivalence:

- Several early authors pointed out that domain tests target specific types of errors (Binder, 2000; Boyer, 1975; Clarke, 1976; White, 1981).
- Weyuker & Ostrand (1980; *see also Jeng & Weyuker, 1989, p. 38)* proposed that the ideal partitioning should be *revealing*, *meaning that it*

  "divides the domain into disjoint subdomains with the property that within each subdomain, either the program produces the correct answer for every element or the program produces an incorrect answer for every element."

Partitioning

We 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 (Weyuker & Ostrand, 1980; Weyuker & Jeng, 1991).
- 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 experience history.

Four central questions

- What domain are we testing?
- How do we determine how to group values of a variable(s) into equivalence classes?
- **How do we determine which members of each class to test?**
- How do we determine whether the program passed or failed the test?

Selection

Suppose that our program design is:

- INPUT < 10 result: Error message
- 10 <= INPUT < 25 result: Print "hello"
- 25 <=INPUT result: Error message

Some error types

- Program doesn't like numbers
  - Any number will do
- Inequalities mis-specified (e.g. INPUT <= 25 instead of < 25)
  - Detect only at boundary
- Boundary value mistyped (e.g. INPUT < 52, transposition error)
  - Detect at boundary and any other value that will be handled incorrectly

Boundary values (here, test at 25) catch all 3 errors

Non-boundary values (consider 53) may catch only 1 of the 3 errors

Selection

How do we determine which members of each class to test?

- Boundaries are representatives of the equivalence classes we sample them from. They're more likely to expose an error than other class members, so they're *better representatives.*
- "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."* (Kaner & Bach, 2003, p. 37)
- Generalizing from boundaries to ***best representatives*** is useful for analyzing nonordered spaces, such as printers (Kaner, et al., 1993).

Selection is a stratified sampling problem

Domain testing is an instance of stratified sampling

> In statistics, **stratified sampling** is a method of sampling from a population.
>
> When sub-populations vary considerably, it is advantageous to sample each subpopulation (stratum) independently. **Stratification** is the process of grouping members of the population into relatively homogeneous subgroups before sampling. The strata should be mutually exclusive: every element in the population must be assigned to only one stratum. The strata should also be collectively exhaustive: no population element can be excluded. Then random or systematic sampling is applied within each stratum. This often improves the representativeness of the sample by reducing sampling error.
>
> Wikipedia: Stratified Sampling

- Polling looks for typical representatives
- Testing looks for "best" representatives (most likely to yield failure or other new information)

Four central questions

- What domain are we testing?
- How do we determine how to group values of a variable(s) into equivalence classes?
- How do we determine which members of each class to test?
- **How do we determine whether the program passed or failed the test?**

Evaluation

How do we determine whether the program passed or failed the test?

- Stunningly little guidance on this in the domain testing literature
- The answer is generally treated as obvious, but it is not
- We did not buy the program in order to segregate 100 from 101 for the variable X. We did not buy the program to obtain an input filter.

Evaluation

How do we determine whether the program passed or failed the test?

- We bought the program to obtain benefits.
- We enter an X value to obtain benefits.
- We test a value of X
  - Against the filter
  - ***Against any use of X later in the program.***

Failure to consider consequences with this technique is our students' most common failure. It is widespread among practitioners too.

Our structure for domain testing

Here is a list of several tasks that people often do as part of a domain test. We organized the book's chapters around this list because it seems to us to put the tasks into a logical order.

For any particular product or variable, you might skip several of these tasks. Or, you might do tasks in a different order than we list here. This is an inventory, not a control structure.

A. Identify the potentially interesting variables.
B. Identify the variable(s) you can analyze now.
C. Determine the variable's primary dimension.
D. Determine the variable's type or scale.
E. Determine whether the variable's values can be ordered (smallest to largest)
F. Partition (create equivalence classes):
    - If the dimension is ordered, determine the sub-ranges and transition points.
    - If the dimension is not ordered, determine what "similar" means for this variable, and base partitioning on that.
G. Lay out the analysis in a table that shows the partitions and best representatives (or boundary cases) for each partition.
H. Identify secondary dimensions and analyze each in the classical way.
I. Generalize the analysis to multidimensional variables.
J. Linearize the domain (if possible).
K. Identify constraints among the variables
L. Identify and test variables that hold results (output variables).
M. Evaluate how the program uses the value of this variable.
N. Identify additional potentially-related variables for combination testing.
O. Create combination tests for multidimensional or related variables.
P. Imagine risks that don't necessarily map to an obvious dimension.
Q. Identify and list unanalyzed variables. Gather information for later analysis.
R. Summarize your analysis with a risk/equivalence table.

Our structure for domain testing

A. **Identify the potentially interesting variables.**
B. **Identify the variable(s) you can analyze now**.
   - Frequently, we notice more variables than we can analyze today.
   - Identify the ones you're going to work with *now*.
   - Note the others and add details when you get them (or when you go looking for them).

Exercise 3

SunTrust issues Visa credit cards with credit limits in the range of $400 to $40000. A customer is not to be approved for credit limits outside this range. A customer can apply for the card using an online application form in which one of the fields requires that the customer type in his/her desired credit limit.

- Identify the variables
- Do the domain analysis on as many of these variables as you think are appropriate.
- Explain why you restricted your analysis to the variable(s) that you did.

Our structure for domain testing

C. **Determine the variable's primary dimension.**

- If we're trying to put a number into X between 0 and 100, the primary dimension is a number line that includes [0, 100].
- To figure out the primary dimension of a variable, ask what the variable is for. What range of values will it contain if we only give it useful values? That range lies along the primary dimension.
- Sometimes, this is the *only* dimension
  - If X is stored in a byte and is read by a function that will interpret the bit pattern in that byte as a signed integer, then no matter what we put into X, what gets read is an integer between -128 and 127.
- Other times, there are possibilities outside of this dimension. We'll talk about these as *secondary dimensions*.

Our structure for domain testing

D. Determine the variable's type or scale.

We will analyze

- integers

  very differently from

- strings.

Exercise 4

What are the **primary dimension and scale** of:

The page setup function of a text editor allows a user to set the width of the page in the range of 1 to 56 inches. The page width input field will accept (and remember) up to 30 places after the decimal point.

What are the boundaries between valid and invalid input?

Exercise 5

What are the **primary dimension and scale** of:

An ATM allows withdrawals of cash in amounts of $20 increments from $20 to $200 (inclusive).

What are the boundaries between valid and invalid input?

Exercise 6

What are the **primary dimension and scale** of:

> **A StudentLastName field must start with an alphabetic character (upper or lower case). Subsequent characters must be letters, numbers, or spaces.**

What are the boundaries between valid and invalid input?

Examples of ordered sets

- ranges of numbers
- character codes
- how many times something is done
  - (e.g. shareware limit on number of uses of a product)
  - (e.g. how many times you can do it before you run out of memory)
- how many names in a mailing list, records in a database, variables in a spreadsheet, bookmarks, abbreviations
- size of the sum of variables, or of some other computed value (think binary and think digits)
- size of a number that you enter (number of digits) or size of a character string
- size of a concatenated string
- size of a path specification
- size of a file name
- size (in characters) of a document

Examples of ordered sets

- size of a file (note special values such as exactly 64K, exactly 512 bytes, etc.)
- size of the document on the page (compared to page margins) (across different page margins, page sizes)
- size of a document on a page, in terms of the memory requirements for the page. This might just be in terms of resolution x page size, but it may be more complex if we have compression.
- equivalent output events (such as printing documents)
- amount of available memory (> 128 meg, > 640K, etc.)
- visual resolution, size of screen, number of colors
- operating system version
- variations within a group of "compatible" printers, sound cards, modems, etc.
- equivalent event times (when something happens)
- timing: how long between event A and event B (and in which order--races)
- length of time after a timeout (from JUST before to way after) -- what events are important?

Examples of ordered sets

- speed of data entry (time between keystrokes, menus, etc.)
- speed of input--handling of concurrent events
- number of devices connected / active
- system resources consumed / available (also, handles, stack space, etc.)
- date and time
- transitions between algorithms (optimizations) (different ways to compute a function)
- most recent event, first event
- input or output intensity (voltage)
- speed / extent of voltage transition (e.g. from very soft to very loud sound)

Our structure for domain testing

E. Determine whether the variable's values can be ordered (smallest to largest).

- The classic examples are of ordered sets

Exercise 7

What is the **proper ordering** for this variable?:

An ATM allows withdrawals of cash in amounts of $20 increments from $20 to $200 (inclusive).

What are the boundaries between valid and invalid input?

Exercise 8

What is the **proper ordering** for this variable?:

> **A StudentLastName field must start with an alphabetic character (upper or lower case). Subsequent characters must be letters, numbers, or spaces.**

What are the boundaries between valid and invalid input?

Our structure for domain testing

F. Partition (create equivalence classes):

- If the dimension is ordered, determine the sub-ranges and transition points.
- If the dimension is not ordered, determine what "similar" means for this variable, and base partitioning on that.

– We partitioned the values of X into three "equivalence classes:"
  - Too small
  - Just right
  - Too big

– For partitioning of non-orderable sets, see the Printers example in Kaner / Falk / Nguyen, Testing Computer Software

Examples of non-ordered sets

Here are examples of variables that don't fit the traditional mold for equivalence classes but which have enough values that we will have to sample from them. What are the boundary cases?

- Membership in a common group
  - Such as employees vs. non-employees.
  - Such as workers who are full-time or part-time or contract.
- Equivalent hardware
  - such as compatible modems, video cards, routers
- Equivalent output events
  - perhaps any report will do to answer a simple the question: Will the program print reports?
- Equivalent operating environments
  - such as French & English versions of Windows

Our structure for domain testing

G. Lay out the analysis in a table that shows the partitions and best representatives (or boundary cases) for each partition.

Variable	Valid case equivalence class	Invalid case equivalence class	Boundaries & special cases	Notes
X	0 – 100		0	
			100	
		< 0	-1	
		> 100	101	

Note that this shows only the primary dimension.

Exercise 9

Partition this variable and lay out the analysis in a table that shows the partitions and the best representatives for each partition:

The page setup function of a text editor allows a user to set the width of the page in the range of 1 to 56 inches. The page width input field will accept (and remember) up to 30 places after the decimal point.

Exercise 10

Analyze this using steps A through G:

(a) FoodVan delivers groceries to customers who order food over the Net.

- **To decide whether to buy more vans, FoodVan tracks the number of customers who call for a van.**
- **A clerk enters the number of calls into a database each day.**
- **Based on previous experience, the database is set to challenge (ask, “Are you sure?”) any number greater than 400 calls.**

Exercise

A. Identify the potentially interesting variables.
B. Identify the variable(s) you can analyze now.
C. Determine the variable's primary dimension.
D. Determine the variable's type or scale.
E. Determine whether the variable's values can be ordered (smallest to largest)
F. Partition (create equivalence classes):
   - If the dimension is ordered, determine the sub-ranges and transition points.
   - If the dimension is not ordered, determine what "similar" means for this variable, and base partitioning on that.
G. Lay out the analysis in a table that shows the partitions and best representatives (or boundary cases) for each partition.

Exercise 10 (b)

Continued from Exercise 10(a).

Analyze this using steps A through G:

FoodVan schedules drivers one day in advance. To be eligible for an assignment, a driver must have special permission or she must have driven within 30 days of the shift she will be assigned to.

Exercise

A. Identify the potentially interesting variables.
B. Identify the variable(s) you can analyze now.
C. Determine the variable's primary dimension.
D. Determine the variable's type or scale.
E. Determine whether the variable's values can be ordered (smallest to largest)
F. Partition (create equivalence classes):
   - If the dimension is ordered, determine the sub-ranges and transition points.
   - If the dimension is not ordered, determine what "similar" means for this variable, and base partitioning on that.
G. Lay out the analysis in a table that shows the partitions and best representatives (or boundary cases) for each partition.

Understanding domain testing

As you just saw in the last example, one of the underlying risks addressed by domain testing is ambiguity. Interpretation of the specification is often most difficult for the boundary cases.

This is one of the key reasons that we test equivalence classes at their boundaries rather than at random “equivalent” points inside the set. (Read Hamlet & Taylor, 1988; Ostrand & Balcer, 1988.)

Our structure for domain testing

H. Identify secondary dimensions and analyze each in the classical way.

- This is where we consider other data types and other sources of variation that may affect the handling of this variable
  - Non-integer input
  - Non-numeric input
  - Buffer-attackingly large or small values
  - Time
- Weinberg/Myers included these dimensions in their presentations
- Many others (Binder, academics) ignored them completely (e.g. Binder's treatment of Triangle, over 100 tests, all on primary dimension)
- No other discussions that we've seen that distinguish primary from secondary (what the variable/feature is for versus what else might happen to it.)

Our structure for domain testing

I. Generalize the analysis to multidimensional variables.

- Copeland cites this as the breakpoint between equivalence class / boundary analysis & domain analysis because so many academic treatments (I include Beizer in these) present domain testing as a multidimensional (two or three variable) technique
- The sampling theory is the same: partition & select
- If variables are *independent*, we can use combination techniques for independent variables (e.g. combinatorial or random)
- When value of one variable constrains the available values for the other(s), combinatorial (e.g. all-pairs) fails. Instead, we have to map the multidimensional space as it is and sample from the actual boundaries.

Exercise 11

What are the **secondary dimensions** of:

> **The page setup function of a text editor allows a user to set the width of the page in the range of 1 to 56 inches. The page width input field will accept (and remember) up to 30 places after the decimal point.**

What are the boundaries between valid and invalid input—along these secondary dimensions?

Exercise 12

What are the **secondary dimensions** of:

An ATM allows withdrawals of cash in amounts of $20 increments from $20 to $200 (inclusive).

What are the boundaries between valid and invalid input—along these secondary dimensions?

Exercise 13

What are the **secondary dimensions** of:

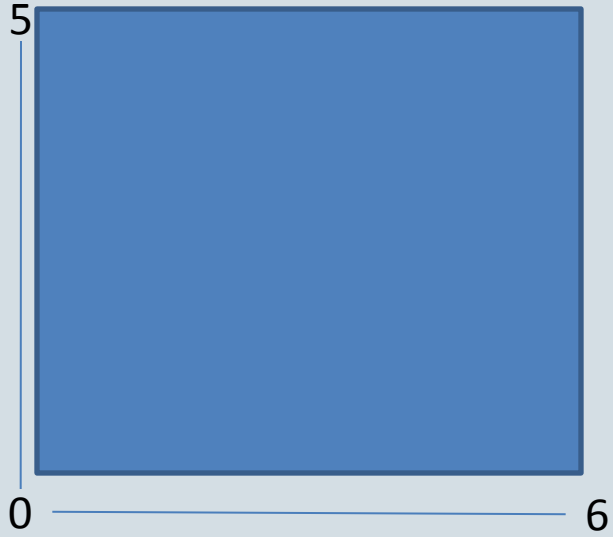
> **You are testing a program that includes an Integer Square Root function. The function reads a 32-bit word that is stored in memory, interprets the contents as an unsigned integer and then computes the square root of the integer, as a floating point number.**

What are the boundaries between valid and invalid input—along these secondary dimensions?

Our structure for domain testing

I. ***Independent*** multidimensional variables.

- Combine
  - X = 0 or 6 (its boundaries)
  - Y = 0 or 5 (its boundaries)



Our structure for domain testing

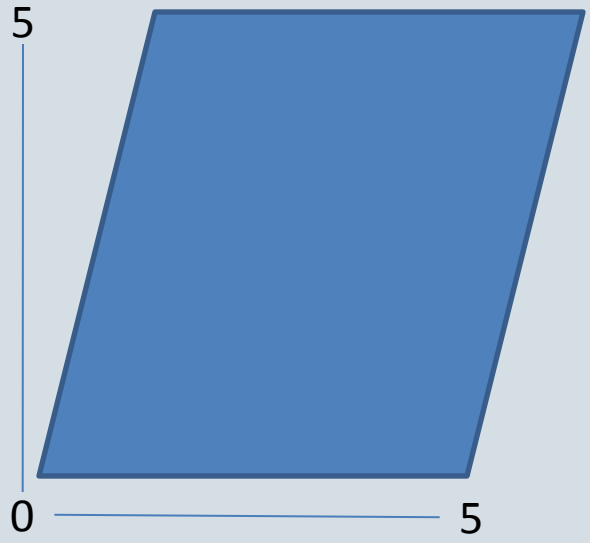
I. ***Non-independent*** multidimensional variables.

I. Combinations of

- X = 0 or 6 (its boundaries)
- Y = 0 or 5 (its boundaries)

are not useful:

- (0, 0) is on boundary
- (6, 0) is not near any boundary
- (5, 0) is on boundary
- (0, 5) is not near any boundary



Our structure for domain testing

J. Linearize the domain (if possible).

The goal is to describe the boundaries of the domain in terms of simple linear inequalities, like this:

$$0 \leq X \leq 1$$
$$X \leq Y \leq 5+X$$

Our structure for domain testing

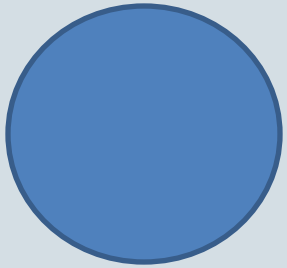
J. Linearize the domain (if possible).

Extensive academic writing about simple linear inequalities:

$$0 \leq X \leq 1$$
$$X \leq Y \leq 5+X$$

- On points, off points, in points, out points
- Special cases that have confused students for years
- The heuristics fall apart for simple nonlinear cases, like:

$$X^2 + Y^2 \leq 10$$



Our structure for domain testing

K. Identify constraints among the variables

- 30 days in September but not August or October
  - (month, day)
- How much is sales tax in this city?
  - (tax rate, city)
- Oh, *that's* against the law *here*?
  - (laws, country)

Our structure for domain testing

L. Identify and test variables that hold results (output variables).

- Simple example: Mailing labels
  - Inputs: Your first, middle and last names
  - Output: A label that your name can't fit on

Exercise 14

- I, J, and K are (signed) integers.
- You can enter data into I and J. You cannot enter data directly into K.
- The program calculates K = I * J.

a. What are the constraints among I, J, and K?
b. What invalid values are possible for K?
c. How would you create a table to show tests for the boundary cases of K?

Our structure for domain testing

M. Evaluate how the program uses the value of this variable.

- Every variable that can be affected by this variable can be tested with boundaries (and other interesting values) of this variable
- 1st through Nth order data flows
- Program slicing addresses this in maintenance

Lecture example: resizing the width of PowerPoint slides

Our structure for domain testing

N. Identify additional potentially-related variables for combination testing.

O. Create combination tests for multidimensional or related variables.

- Earlier, we considered an individual variable that was inherently multidimensional (a point)
- Now consider testing a bunch of variables together that are not inherently dimensions of one larger variable.
- We can reduce the enormous set of tests via stratified sampling
  - All-pairs is one stratified sampling approach (sample with a coverage criterion)
  - Cause-effect graphing is similar but for constraining variables
  - Other possibilities …

Our structure for domain testing

P. Imagine risks that don't necessarily map to an obvious dimension.

- Domain analysis is the start, not the end of test design

Our structure for domain testing

Q. Identify and list unanalyzed variables. Gather information for later analysis.

- You will always have more research to do
- You will never have a complete test plan

R. Summarize your analysis with a risk/equivalence table

For more examples, see workbook handout p. 62, 68, 135

Variable	Risk (potential failure	Classes that should not trigger the failure	Classes that might trigger the failure	Test cases (best representatives)	Notes
Variable 1	mishandles values that are too small	> 0	< 1	0	
				-1	
				-999999999999etc	buffer over-flow
	mishandles values that are too large	< 101	> 100	101	
				9999999999etc	
	misclassifies valid values	< 1, > 100	1 - 100	1	
				100	
	fails on nondigits	digits	non-digits	/	ASCII 47
				:	ASCII 58
				A	

Our structure for domain testing

R. Summarize your analysis with a risk/equivalence table.

- Classical domain analysis starts with the primary dimension, partitions it, and selects "best representative" tests for each partition.

  Underlying this is a theory of error: programmers may misclassify values of the variable, especially at boundaries between partitions.

- Many authors extend this analysis to values of the variable that are "off" the primary dimension, such as letters in a number field or entering only two values into a 3-value dialog.

  These extensions to secondary dimensions are interesting when the user error (or other source of bad data) is plausible. In these cases, we pick a best representative of this class of error—an example of this type of error that is most likely to cause the program to fail.

Our structure for domain testing

R. Summarize your analysis with a risk/equivalence table.

- Risk-based testing starts from a possible error and designs tests that should expose the error if it is there.
- Domain testing is a type of risk-based testing that is focused on classification errors and their consequences.
- ***Risk-based domain testing*** starts from the potential error:
  - We identify how the variable we are working with might be involved with that error
  - We partition values of the variable into classes
    - We ignore the classes that cannot expose the error
    - We test best representatives from classes that can expose the error if it is in the code.

Our structure for domain testing

R. Summarize your analysis with a risk/equivalence table.

We prefer the classical table for simple, academic examples because the risk-oriented table is so much more complex.

The weakness of the very simple examples is that they are divorced from real-life software:

- You analyze a variable, but you don't know why a program needs it, what the program will do with it, what other variables will be used in conjunction with it.
- As soon as you know the real-life information, many risks (should) become apparent, and these are very difficult to represent in the classical table because it is best used to highlight one or a few primary dimensions of a variable.

The risk-oriented table helps us organize the testing that is based on this broader knowledge of the application.

- Any time you are thinking beyond the basic "too big / too small" tests, this style of table might be more helpful than the classical one.

Notes on the FoodVan Exercise

Even these simple specifications are ambiguous:

- Does “within 30 days” mean “less than 30” or “less than or equal to 30” ?
- When does the special permission have to have been issued?
- If you can work tomorrow morning on the basis of permission, can you work tomorrow afternoon on the basis of experience? Is tomorrow morning within 30 days of tomorrow afternoon?
- Do we compute 30 days in days or hours (minutes / seconds)?
- What result if the last day you worked was 28 days ago? 29 days ago? 30 days ago?

Even if you are clear on the answers to these, do you believe that the programmer and the specification writer will come to the same answers?

References

For a list of references, see our file on disk—excerpt from the Domain Testing Workbook