Introduction to Domain Testing

Cem Kaner
January, 2018

What Is Domain Testing?

The most widely taught technique for designing software tests

What Is Domain Testing?

A risk-based sampling strategy for efficiently hunting bugs in a program's handling of data.

What's A Domain?

Here's a typical definition of a mathematical function:

$$z = f(x,y) = \begin{cases} 100 \times \frac{x}{y}, & when\ x,y > 0 \\ 0, & when\ x = 0 \\ & x,y,z\ are\ integers \end{cases}$$

This function expresses the relationship between two numbers as a percentage. For example, under this definition:

$$f(1,2) = 50$$
$$f(1,3) = 33$$

Domains Of A Mathematical Function

$$z = f(x,y) = \begin{cases} 100 \times \frac{x}{y}, & when\ x, y > 0 \\ 0, & when\ x = 0 \\ & x, y, z\ are\ integers \end{cases}$$

- The set of possible values of ***x*** and ***y*** is the function's ***input domain.***
- The set of possible values of *f(x,y)* is the ***output domain***.

Mathematical Domains & Subject Matter Domains

The word *domain* often refers to a subject matter. For example,

- An accounting program's domain is the field of accounting.
- An accountant who helps design or document the program is often called a *subject matter expert* or a *domain expert*.

It seems obvious and natural to say *domain testing* to mean *testing done by a domain expert.*

But the history of the term in software testing applies *Domain Testing* to mathematical domains rather than subject matter domains.

Back to … Domains Of A Mathematical Function

$$z = f(x, y) = \begin{cases} 100 \times \frac{x}{y}, & when\ x, y > 0 \\ 0, & when\ x = 0 \\ & x, y, z\ are\ integers \end{cases}$$

- This defines the function's ***input domain*** and ***output domain***.
- This function is ***undefined*** when $y = 0$ or when x or y is negative or is not an integer.
- We call values of x or y ***invalid*** for this function if the function is not defined to handle them.

Key difference between mathematics and programming

A program must respond appropriately to every value that could possibly be sent to any of its functions, even if the mathematical definition of that function does not include that value.

Math Functions Are Not Identical To Their Implementations

Input values	*Output of the math function*	*Output of the programmed function*
x = 1, y = 2	50	50
x = 0, y = 0	0	0
x = 1, y = 0	Undefined	Proper error handling
x = -1, y = 2	Undefined	Proper error handling
x = a, y = %	Undefined	Proper error handling

When we test programs, we have to test how they handle mathematically-invalid values.

When we test programs, we have to test how they handle mathematically-invalid values.

There are infinitely many invalid values for any function:

Too big

Too small

Too strange

We can't test them all,
we hope to find all the bugs,
so we need a good sampling strategy.

Testing Variables In The Programmed Function

Variables: If a program can put a value into something, that something is a variable.

Data Type: Programmers normally define the types of their variables. Mathematicians can say that ***x, y*** and ***z*** are integers, but programmers have to say what **type** of integer, so the program has the right amount of space for them.

Let's make *our variables* **short integers.** In Java, this means that $-32768 \leq x, y, z \leq 32767$.

Equivalence classes for inputs

$$z = f(x, y) = \begin{cases} 100 \times \frac{x}{y}, & when\ x, y > 0 \\ 0, & when\ x = 0 \\ x, y, z\ are\ short\ integers \end{cases}$$

- Valid values of ***x:*** 0 to 32767
- Valid values of ***y:*** 1 to 32767
- All other values are invalid
  - When testing values for ***x***, the program should handle all values in {0, 1, …, 32767} the same way. We call this an **equivalence class.**

What's an Equivalence Class?

An equivalence class is a set of values that we think the program will handle in the same way.

Some Equivalence Classes For *x*

- Valid:

  {0, 1, …, 32767}
- Too small

  {-32768, …, -1}

  { less than -32768} not short integers
- Too big

  {bigger than 32767} not short integers
- Too strange
  - letters, operators (like +), other non-numeric, non-operator characters
  - Floating point (like, 1.2)
  - Expressions (like 3-2) that yield positive short integers

Boundary values

Instead of testing all of the values of a set, test the biggest and smallest. For ***x***, we would test:

{0, 1, …, 32767}	--> Test 0 and 32767
{-32768, …, -1}	--> Test -1 and -32768
{less than -32768}	--> Test -32769
{bigger than 32767}	--> Test 32768

Boundary testing provides an efficient strategy for choosing values from each of these equivalence classes.

Efficiency:

- We test with a very small proportion of the total of available-to-test data values
- We learn (almost) everything that we could learn if we actually tested all of the data values

Boundaries Are Examples Of Best Representatives

- Some equivalence classes don't have obvious boundaries
  - ***characters that are not numbers or operators***
- To test them
  - identify values that are more likely to make the program fail, and test with those
    - ¼ Unicode character 00BC (one-quarter)
    - ² Unicode character 00B2 (superscript 2)
    - ₂ Unicode character 2082 (subscript 2)

A **best representative** of an equivalence class is a boundary value OR another value in the class that seems more likely than other values to cause a program failure

Non-numeric inputs

The sampling strategy for non-numeric values is harder, but the principle is the same:

For each equivalence class:

- Pick a few values to represent the full set.
- Pick values that are the most likely to cause failure if the program has a relevant bug.

This technique is entirely focused on data

Domain testing helps you decide what values of a variable to test

But it doesn't tell you how to run the test or how to evaluate the program's behavior

Like most other test techniques, it guides you in only a portion of the task of designing the test.

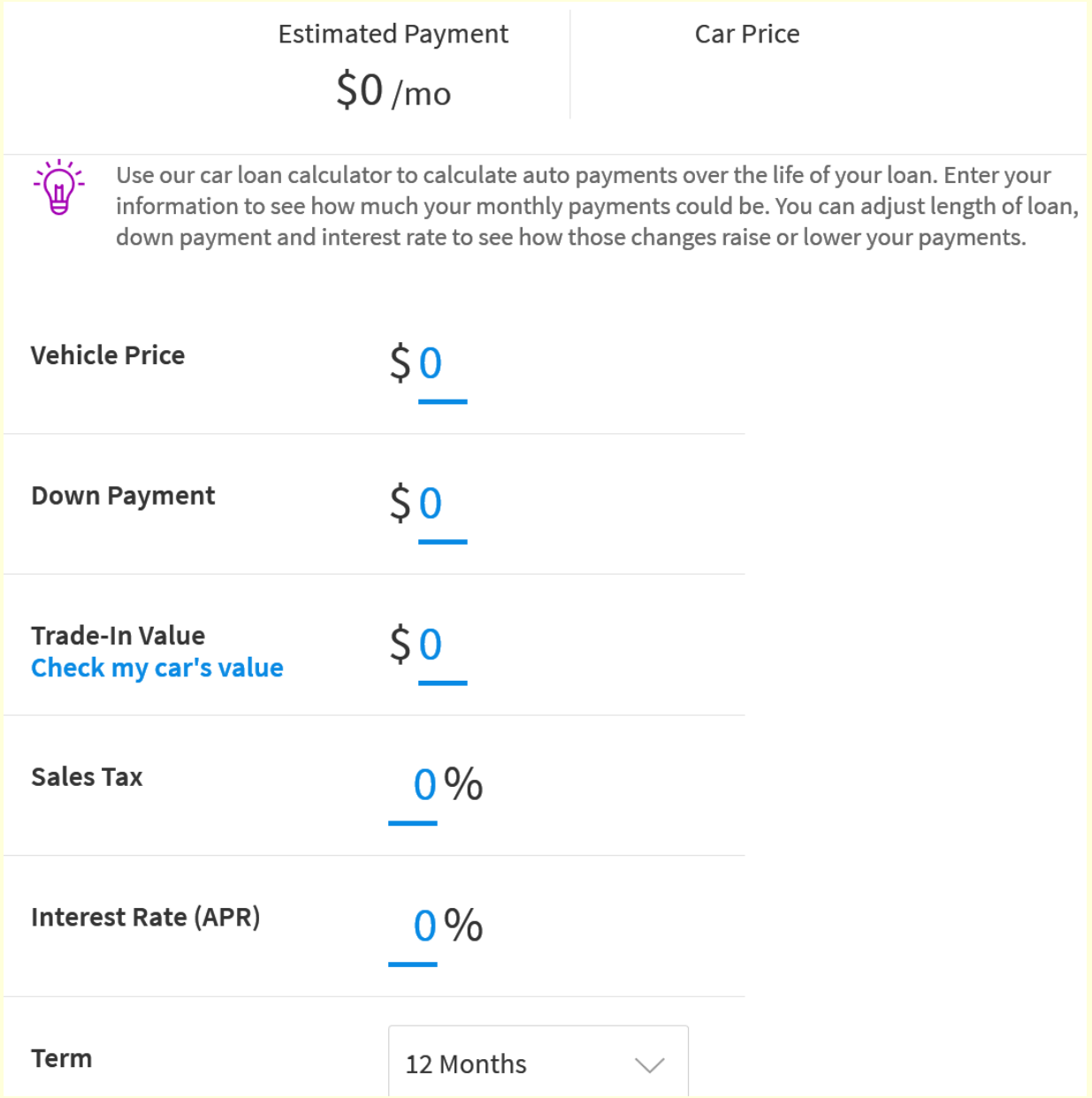
A working example

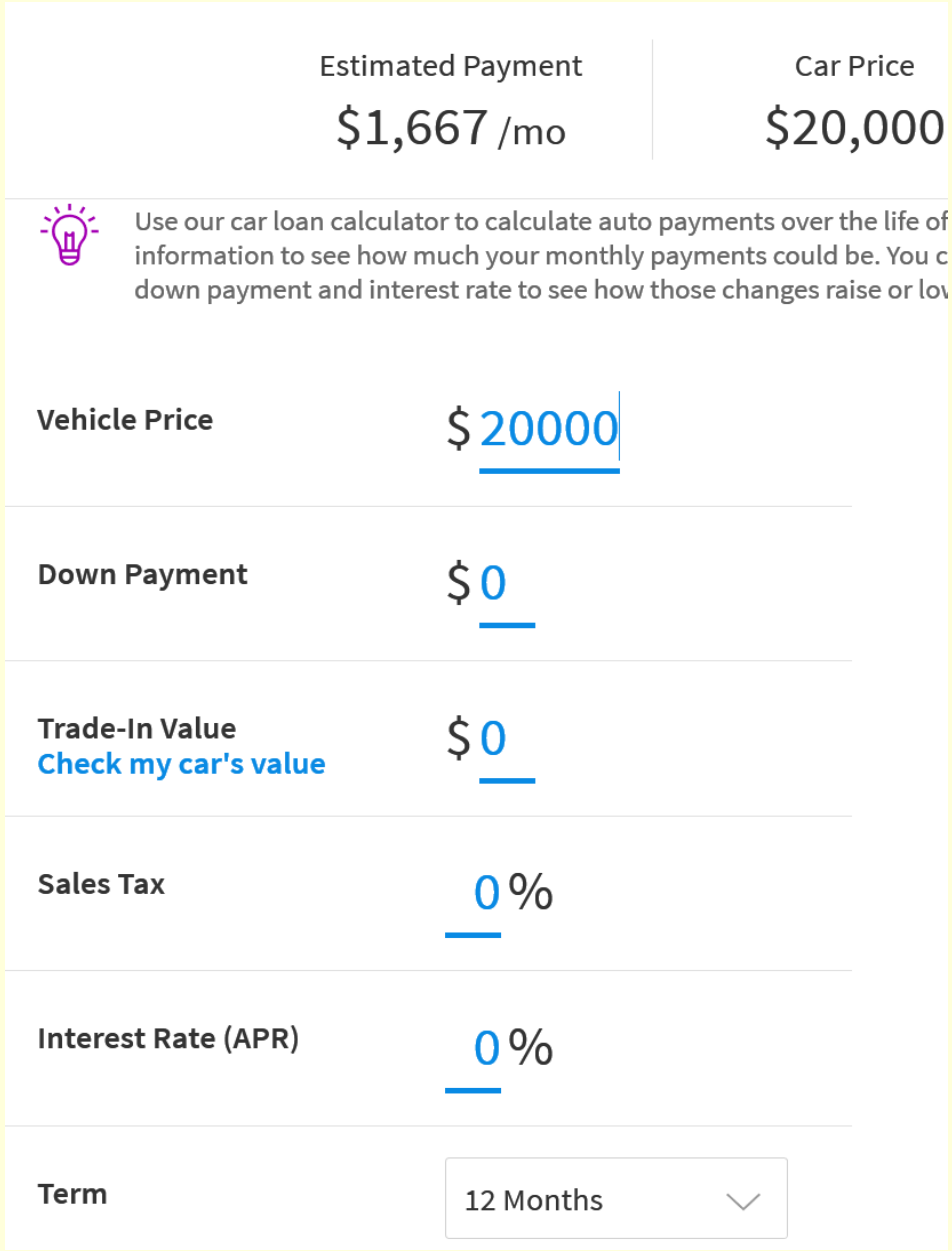
https://www.cars.com/car-loan-calculator/

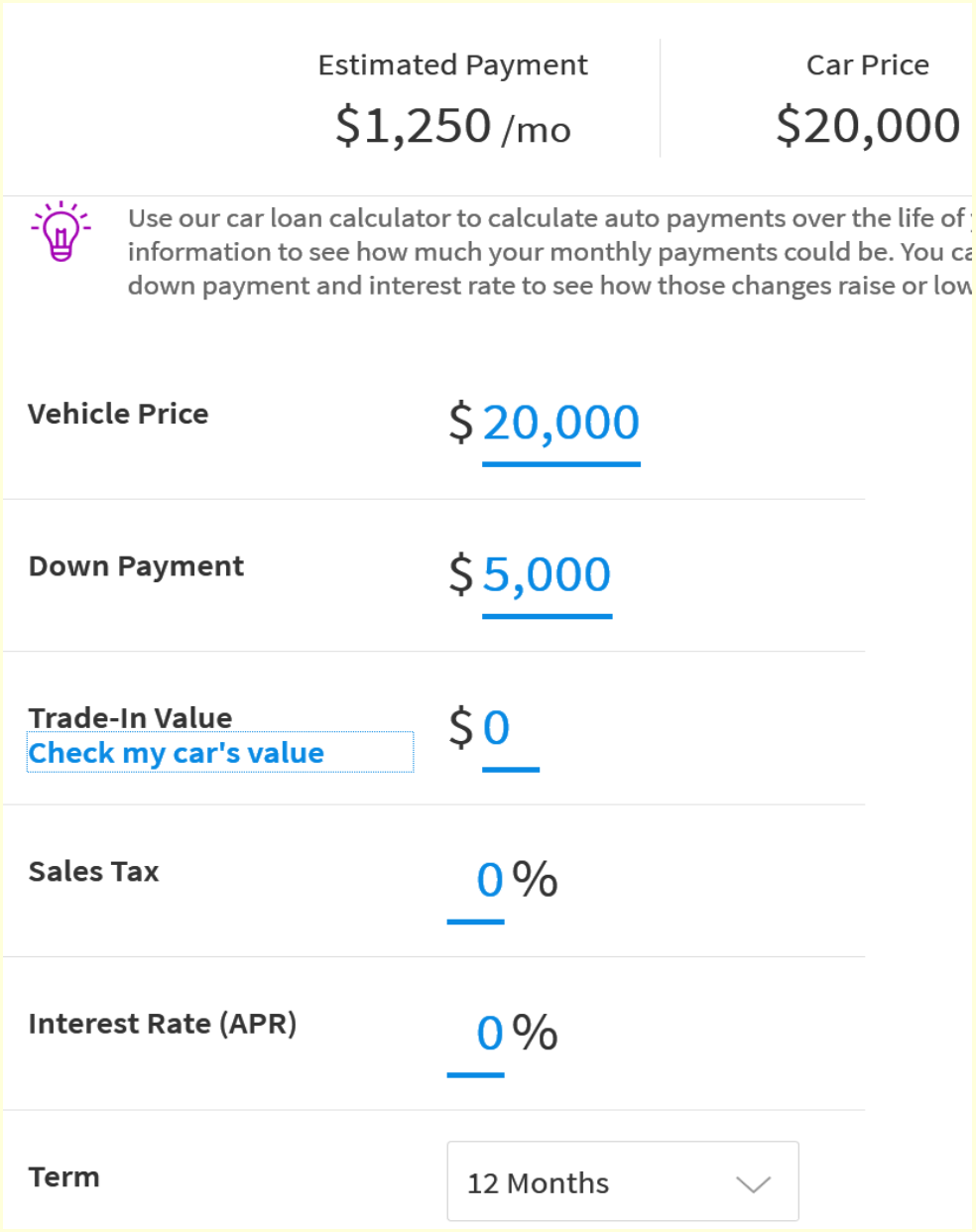
6 input variables

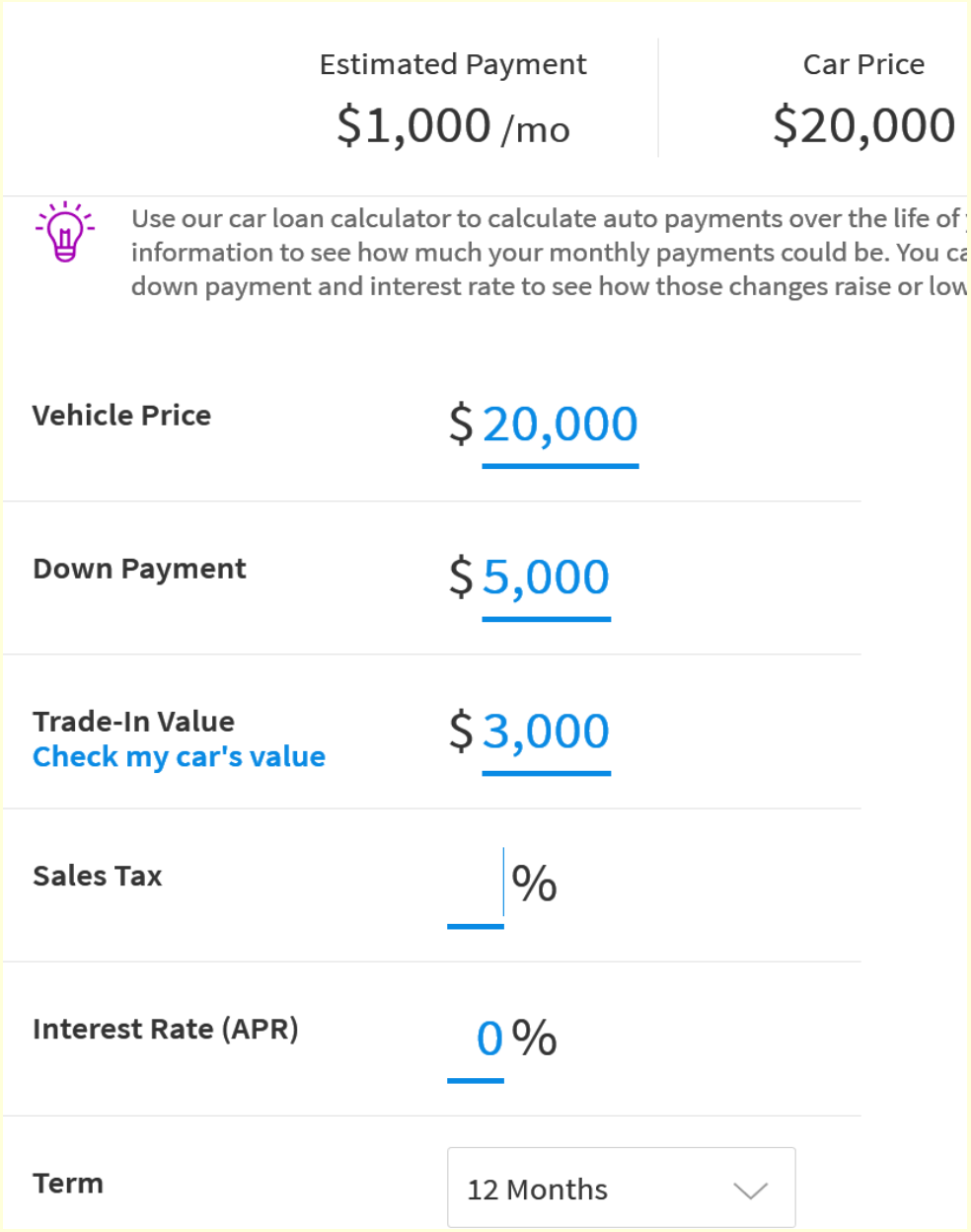
2 output variables

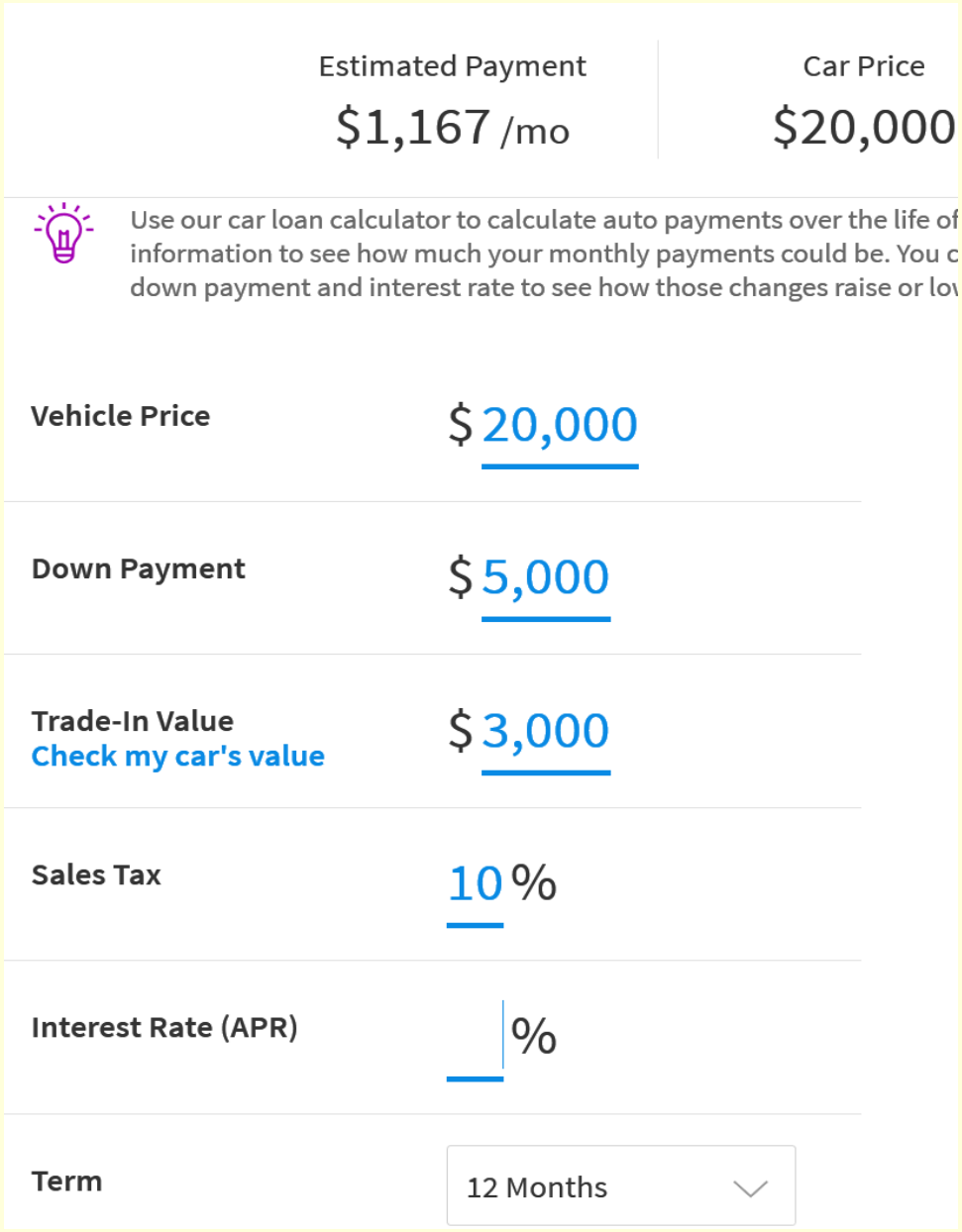
1 (calculated) result variable

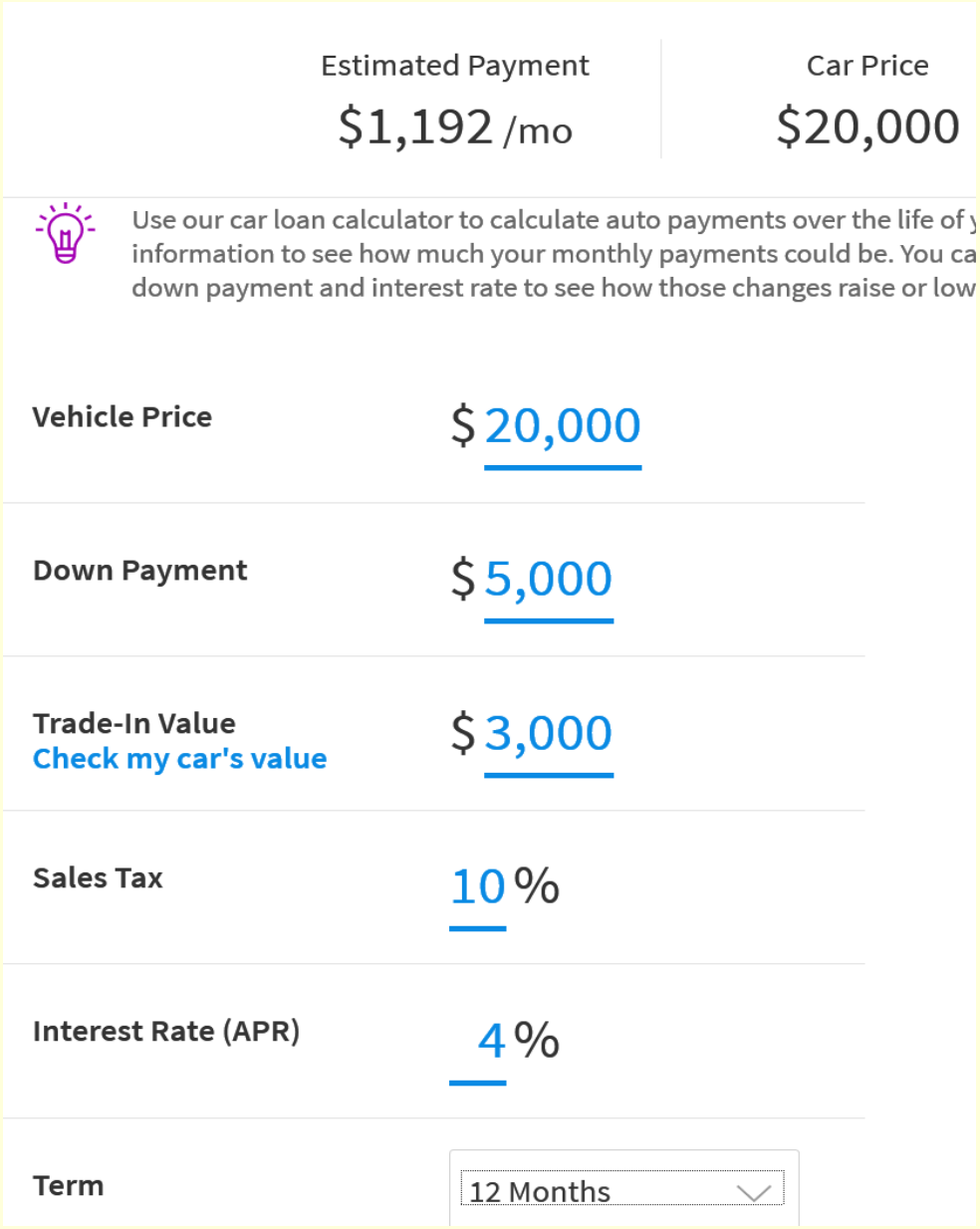


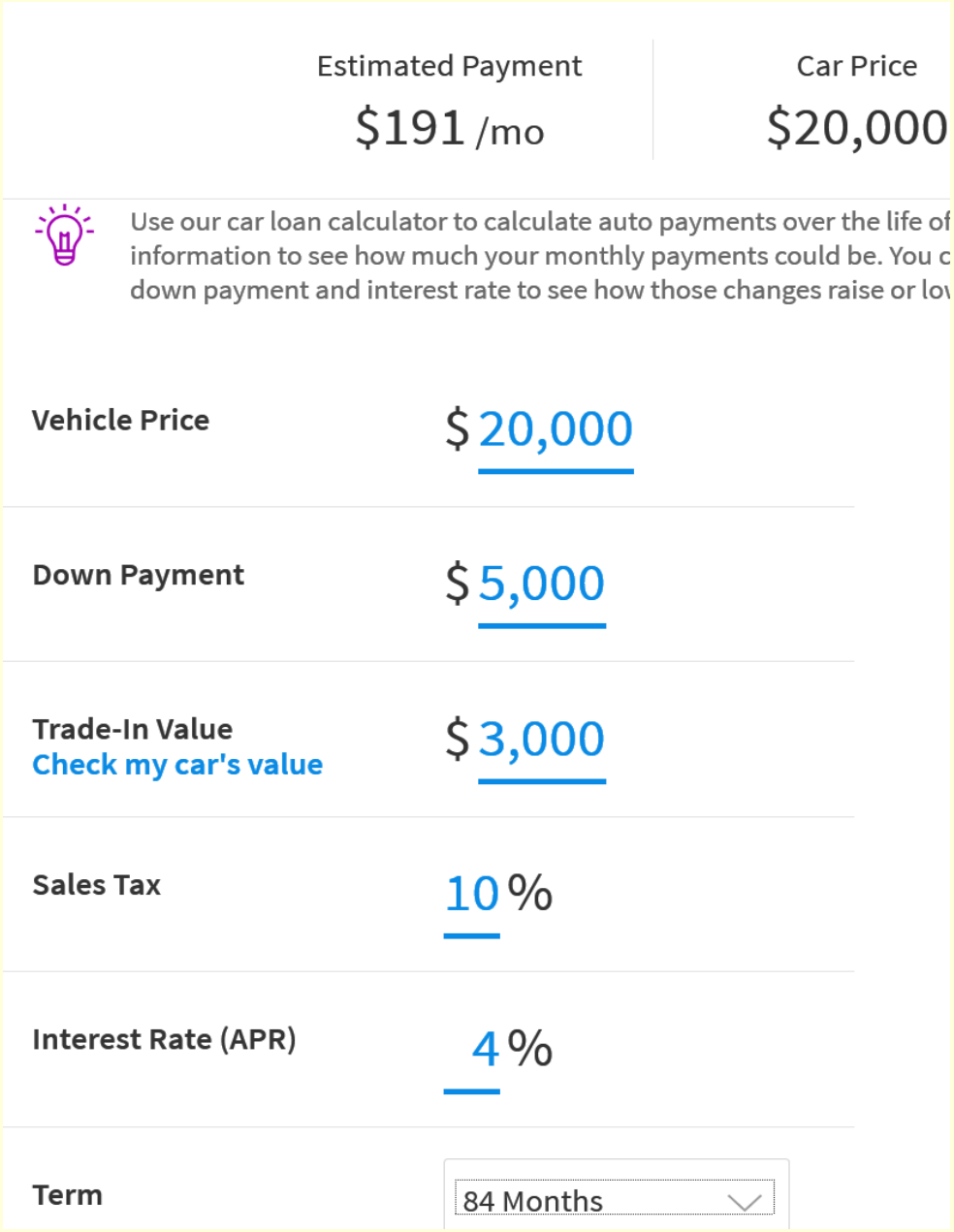

Estimated Payment
$1,667 /mo
Car Price
$20,000
Use our car loan calculator to calculate auto payments over the life of
information to see how much your monthly payments could be. You c
down payment and interest rate to see how those changes raise or lov
Vehicle Price
$ 20000
Down Payment
$ 0
Trade-In Value
Check my car's value
$ 0
Sales Tax
0 %
Interest Rate (APR)
0 %
Term
12 Months


Estimated Payment
$1,250 /mo
Car Price
$20,000
Use our car loan calculator to calculate auto payments over the life of
information to see how much your monthly payments could be. You ca
down payment and interest rate to see how those changes raise or low
Vehicle Price
$ 20,000
Down Payment
$ 5,000
Trade-In Value
Check my car's value
$ 0
Sales Tax
0 %
Interest Rate (APR)
0 %
Term
12 Months


Estimated Payment
$1,000 /mo
Car Price
$20,000
Use our car loan calculator to calculate auto payments over the life of
information to see how much your monthly payments could be. You ca
down payment and interest rate to see how those changes raise or low
Vehicle Price
$ 20,000
Down Payment
$ 5,000
Trade-In Value
Check my car's value
$ 3,000
Sales Tax
%
Interest Rate (APR)
0 %
Term
12 Months


Estimated Payment
$1,167 /mo
Car Price
$20,000
Use our car loan calculator to calculate auto payments over the life of
information to see how much your monthly payments could be. You c
down payment and interest rate to see how those changes raise or lo
Vehicle Price
$ 20,000
Down Payment
$ 5,000
Trade-In Value
Check my car's value
$ 3,000
Sales Tax
10 %
Interest Rate (APR)
%
Term
12 Months


Estimated Payment
$1,192 /mo
Car Price
$20,000
Use our car loan calculator to calculate auto payments over the life of y
information to see how much your monthly payments could be. You ca
down payment and interest rate to see how those changes raise or low
Vehicle Price
$ 20,000
Down Payment
$ 5,000
Trade-In Value
Check my car's value
$ 3,000
Sales Tax
10 %
Interest Rate (APR)
4 %
Term
12 Months


Estimated Payment
$191 /mo
Car Price
$20,000
Use our car loan calculator to calculate auto payments over the life of
information to see how much your monthly payments could be. You c
down payment and interest rate to see how those changes raise or lo
Vehicle Price
$ 20,000
Down Payment
$ 5,000
Trade-In Value
Check my car's value
$ 3,000
Sales Tax
10 %
Interest Rate (APR)
4 %
Term
84 Months

The cars.com example: INPUT VALUES

- What are the boundaries of all these variables?
- What numeric values will the program reject?
- What does it do with non-numeric values?
- Does it treat all non-numerics the same?

The cars.com example: RESULT VALUES

- Are the calculations of estimated payments correct?
- What are the biggest/smallest possible estimated payments?

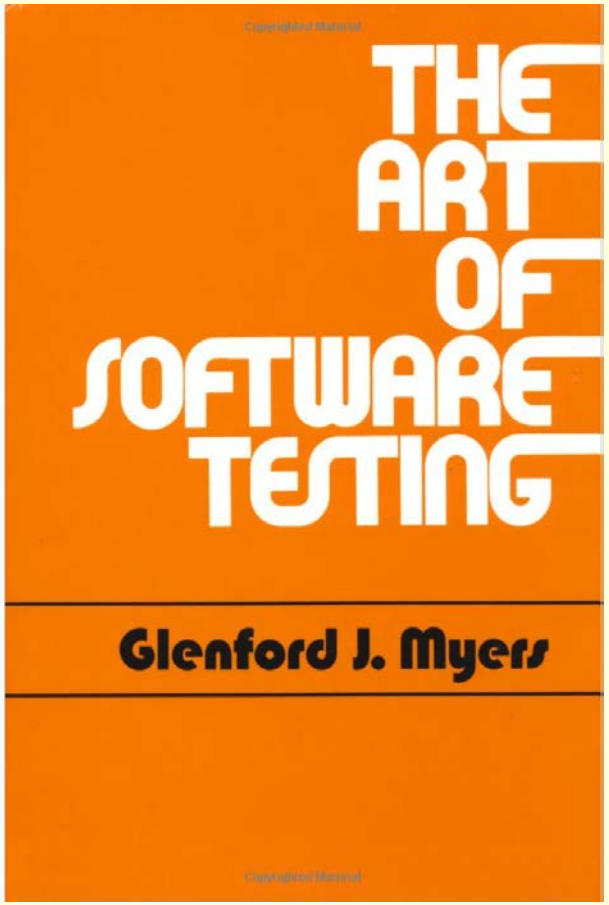
The cars.com example: OUTPUT VALUES

- Can you set a car price that is too hard (e.g. too big) to display?
- Can you calculate a payment value that is too hard to display?
- Will the program save these values or email them to someone? Are any values difficult to save or send correctly?

The cars.com example: EFFICIENCY

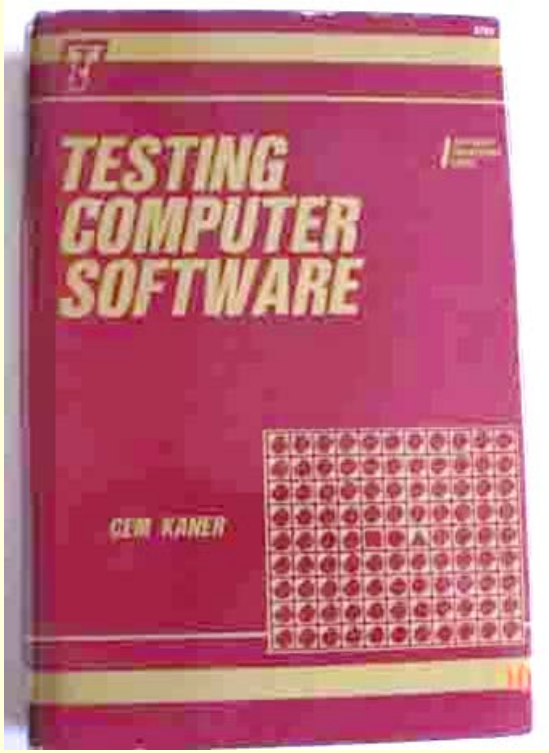
- How many tests would you have to run (and the program pass) before you would accept the program as correct?

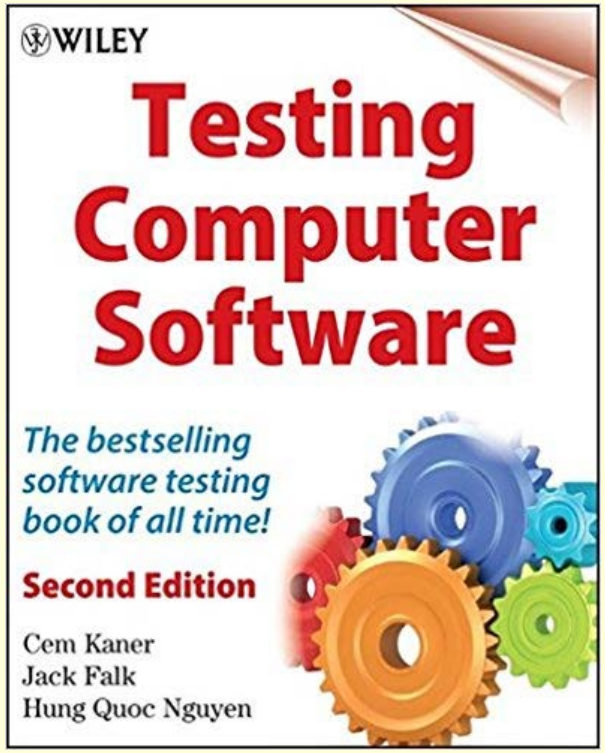
We've made it to 1979

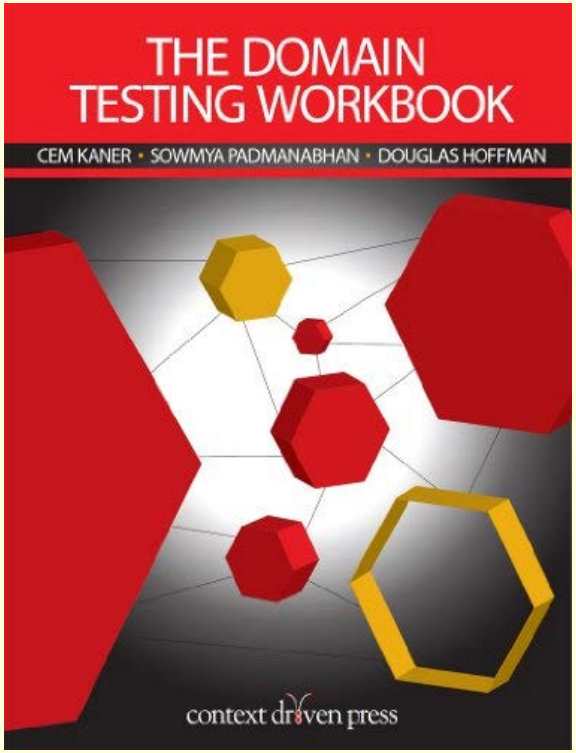


Myers was one of the first authors to present equivalence classes and boundaries in a way that highlighted the many types of invalid inputs

Testing invalid inputs







I extended Myers' work in the 1980's (as did many others). I think the *Domain Testing Workbook* (2013) is still the only general, systematic treatment of invalid and non-numeric inputs.

Limitations of the early approach

- Applied only to numeric variables
- Focused on single variables, tests of one variable at a time
- Published discussions focused on inputs:
  - whether the program was properly protected from bad input and
  - whether it appeared to properly handle acceptable input

The technique was generalized in many ways over the next 30 years.

First (Academic) Generalization: Multiple Variables

A program typically looks at the values of many variables when it is doing something to deliver value to the user, such as preparing a report or making a significant decision.

How does domain testing apply here?

Multiple Variables: The Easy Situation

All inputs are independent:

$$z = f(a,b,c,d,e,f,\ldots)$$

a is independent of the other variables if

- The value of ***a*** does not limit the values of any of the other inputs to ***f()*** and
- none of the values of the other inputs limit the value of ***a***.

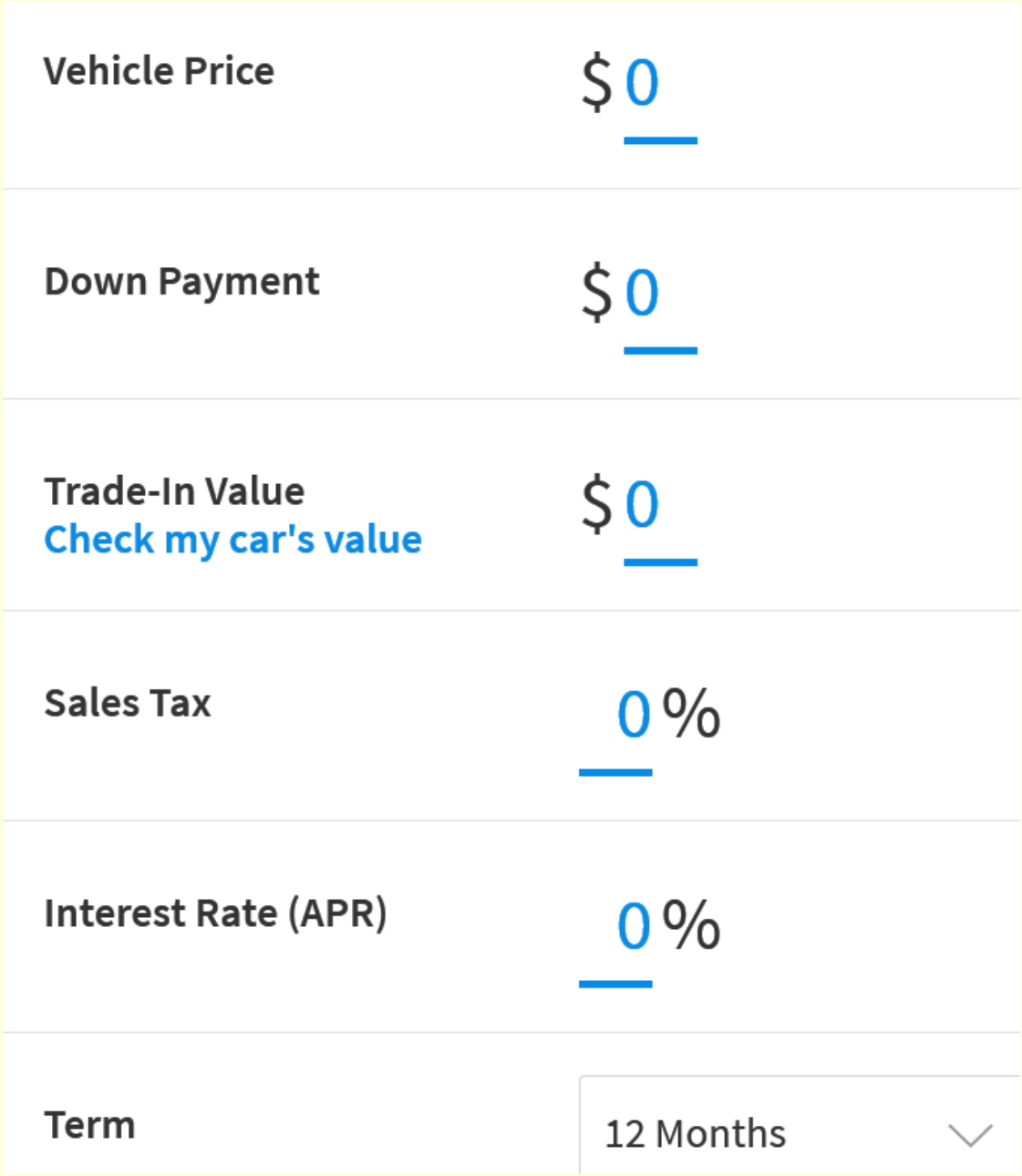
We can test independent variables together using combinatorial testing (such as all pairs). We can pick values for each variable using equivalence class and boundary analysis.

This is a straightforward way of limiting the number of tests.

6 input variables:

Down Payment ≤ Vehicle Price

Otherwise, they look independent



Multiple Variables: The Hard Situation

Harder when inputs are NOT independent:

$$z = f(a,b,c,d,e,f,\ldots)$$

The value of ***a*** **does** limit the value some other inputs to ***f()*** and/or some values of other inputs limit the value of ***a*** or of each other.

Classic example

$$z = f(x,y) = \begin{cases} x + y, & when\ x^2 + y^2 \leq 25 \\ x, y, z\ are\ floating\ point & \end{cases}$$

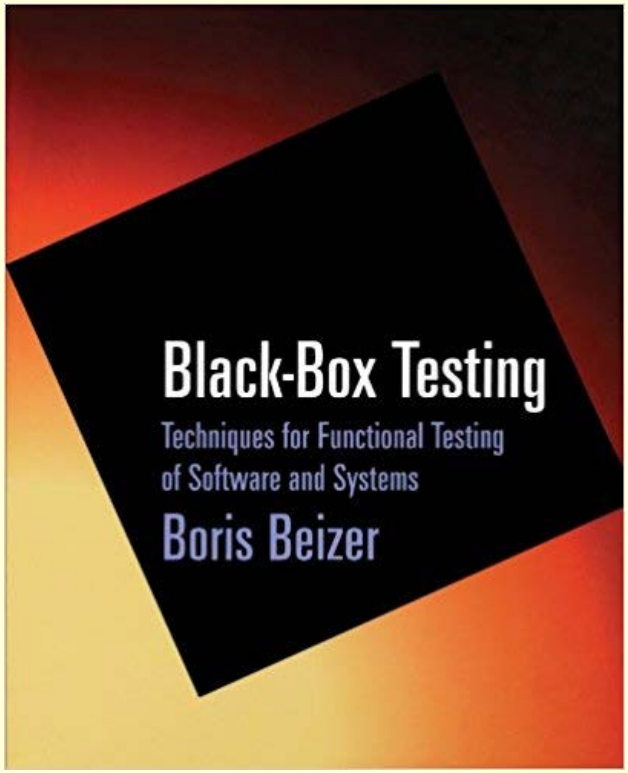
You can plot the valid values of ***x*** and ***y*** inside a circle that includes (0,5), (0, -5), (5,0), and (-5,0). It does not include (5,5).

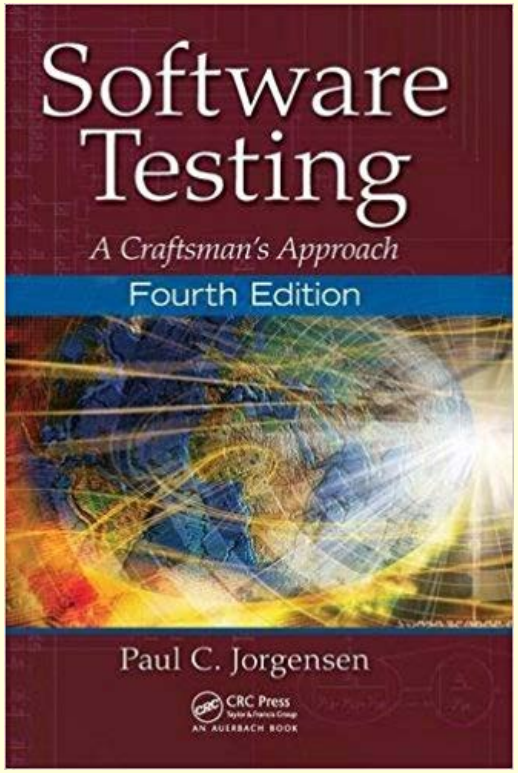
Multiple Variables: Non-Independent Variables

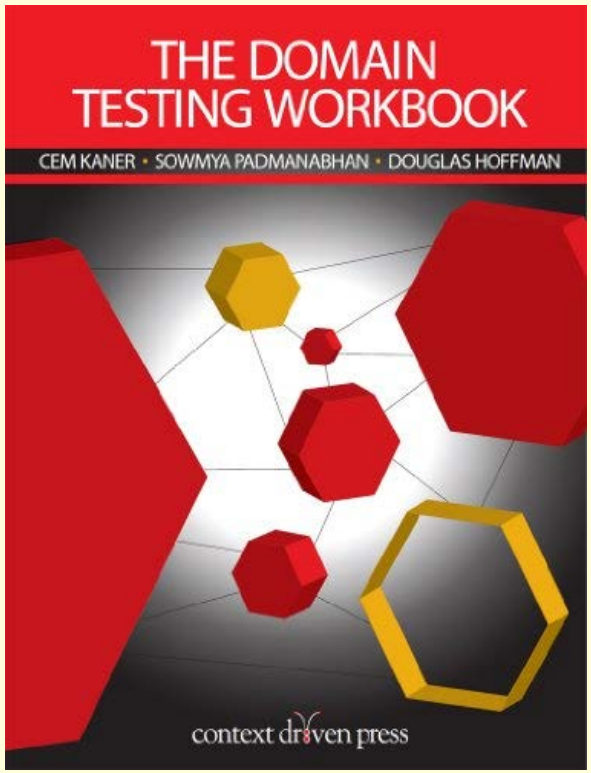
- Values on or inside the circle form one equivalence class (whose boundary is the circle).
- Values outside the circle are another equivalence class.

Test selection requires deep knowledge of relationships among variables.

Treatment of Non-Independent Inputs







Discussed at length in hard-to-read academic papers.

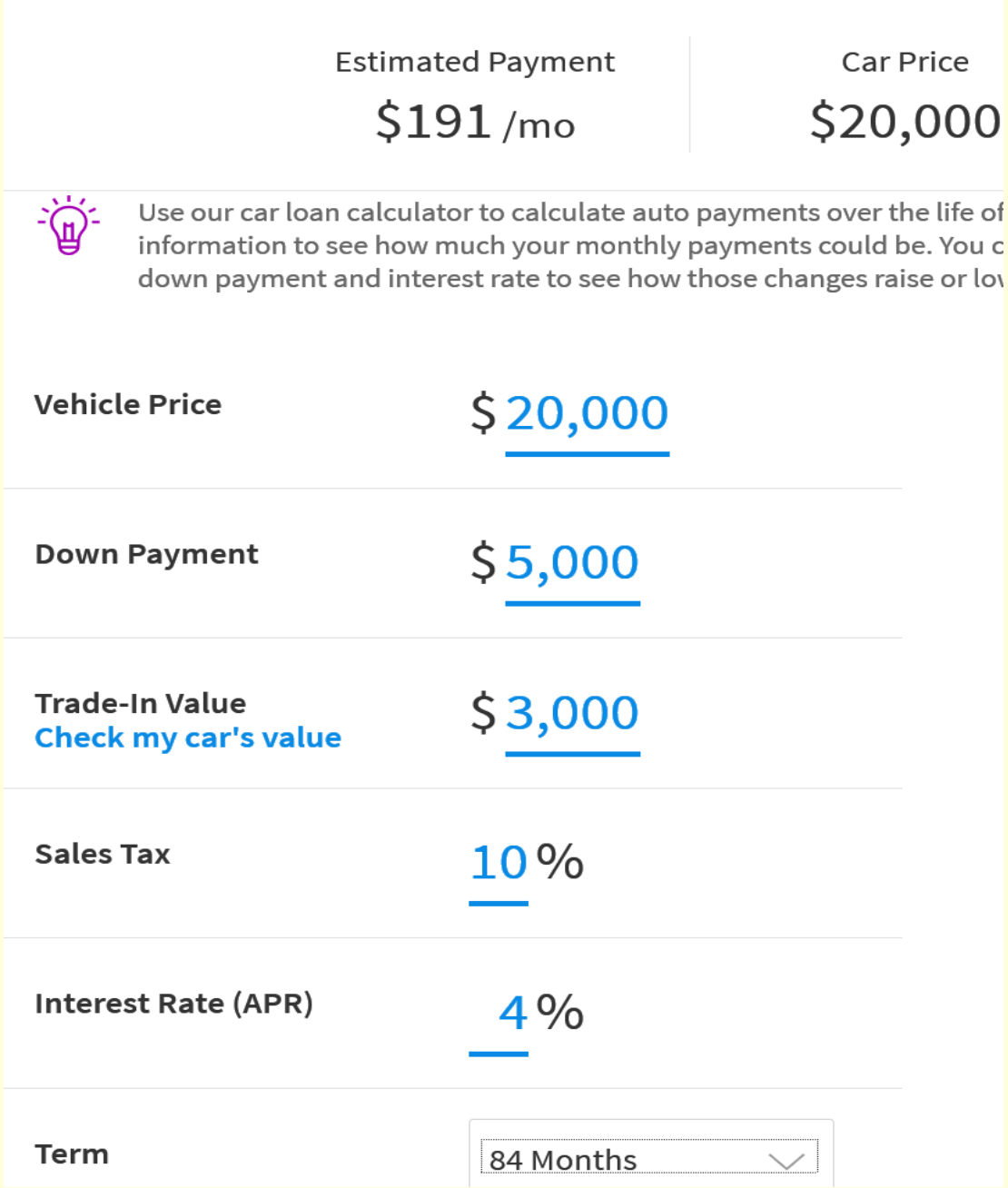
I think these are the best (but not completely successful) efforts to explain to non-mathematician readers the use of domain-test reasoning with non-independent variables

Domain Testing is NOT Only About Inputs

We are usually interested in the accuracy and value of the program's outputs.

It's nice to know that valid input values are accepted without drama

But the person who uses this wants to know what their payment will be.



Vocabulary: Types of Variables

$$z = f(x, y) = x + y$$

- ***x*** and ***y*** are **input variables**
- ***z*** is a **result variable**
- When the program displays the value of ***x*** or ***y*** or ***z***, prints it, saves it, or sends it to another device or program, that variable is an **output variable**.

- The Estimated Payment is both a result variable (it was calculated) and an output variable (it is displayed on the screen).

Testing Beyond The Inputs

When I started testing commercial software in 1983, it was common for testers to push result variables and output variables to their boundary values and beyond.

- ***Is it doing this calculation correctly?***
- ***Is it displaying this value properly?***
- ***Is it saving this value to disk appropriately?***

Questions like these were essential in our work.

Also essential: managing the efficiency of our test suites (running as few tests as possible for each concern that we wanted to check).

Domain Testing was widely used on result variables and input variables in the 1980's even though this application of domain testing was rarely described or taught in the published literature.

It showed up more often in practitioners' conference talks and courses, but I did not see any systematic description or instruction.

Welcome To Risk-Based Testing

- In risk-based testing, we imagine ways that the program might fail, and then design tests to show that the program can actually fail in those ways
- When we look at domain testing through a risk-based lens, we see these questions:
  1. When the program is trying to do a specific task, how could the values of the variables it is using cause it to fail?
  2. How can we make these variables have the risky values, in order to show that the program can actually fail in the ways we imagine?

Input Filter Testing Involves Simple Risks

- An input filter is bad if
  - it allows the program to accept values that are too small, too big, or too strange
    - Test with values that are too small, too big, or too strange (use domain test analysis to limit the size of the set of test values)
- An input filter is bad if
  - it rejects values that are not too small and not too big and not too strange
    - Test with “valid values” (pick boundaries or other best representatives with domain test analysis)

Examples of Result and Output Risks

$z = x + y$, where x, y, z are short integers

- **Overflow risk**: x+y > 32767, overflowing z
- **Underflow risk**: x+y < -32768
- **Overflowed display risk**: the program shows the value of z but allows room for only 5 characters. It can display 32767 and -3276 but not -32767.
- **Overflowed storage risk**: the program writes data to disk, allocating exactly 5 characters for each sum. It converts
  - z = 2 to 00002
  - z = 32767 to 32767
  - z = -32767 to uh oh

How to do the risk analysis (1)

First, consider ***where*** *the program might use the variable*:

- What other functions of the program use this variable?
- Where does the program display or print values of this variable?
- Where does the program send values of this variable to?
- Where does the program store values of this variable?
- Where does the program find values to load into this variable?

The risk analysis (2)

Next, consider ***how*** *the program might use the variable's value*:

- Calculations
- Constraining the values of other variables
- Cases in which the values of other variables are supposed to constrain this one
- Controlling the execution path of the program
- Time-consuming events (maybe this variable controls or impacts tasks that could take too much time)

The risk analysis (2 continued)

How the program might use the variable's value:

- Displays or reports
- Storage of this variable or other variables in a way influenced by this variable
- Cases in which multiple instances of this program can run in parallel and change storage (in RAM or on disk) in a way that changes this variable or what will be loaded into this variable in the future
- Messages or communications with other processes or systems
- Controlling the operation of a device

The risk analysis (3)

For each possible use:

1. When the program is trying to do this task, how could the values of the variables it is using cause it to fail?
2. How can we make these variables have the risky values, in order to show that the program can actually fail in the ways we imagine?

Once we imagine a risk that depends on the value of a variable, we apply **domain-testing analysis**

- What is the set of values that could cause the failure?
- What is the best representative of that set?
- Run the test with that best representative

Reminder: Best Representative

A **best representative** of an equivalence class is a boundary value OR another value in the class that seems more likely than other values to cause a program failure

Why Use Domain Testing? Efficiency & Power

- Domain testing is a sampling strategy.
- The goal is to test a relatively small number of values of a variable rather than testing all of them
- A variable is suitable for domain testing if the set of best representatives is far smaller than the possible values of the variable.
- Use equivalence classes and best representatives to create a sample that is more powerful (more likely to make the program fail) than a random sample.
- The conclusion you want to draw from a set of domain tests is:

  If the program can pass tests with these values, it will pass with any values.

Summary: Scope of domain testing

- Domain testing applies to
  - input variables
  - result variables (results of calculations)
  - output variables (values that will be displayed, printed or sent to other programs)
- Imagine a risk that involves one or more of these variables
  - construct equivalence classes and best representatives to select test values for each relevant variable, in order to:
    - minimize the number of necessary tests for that risk
    - maximize the likelihood of actually causing the program to fail

Application Of Domain Testing

- Most often taught as a standalone technique for checking input filters
  - But most seasoned programmers (and modern languages/libraries/dev tools) manage input filters pretty well already
- Most valuably used in combination with another test technique:
  - You have an idea for a test
  - You use domain testing to optimize the selection of variables' values as you construct that test
  - This is a useful enhancement to almost every other test technique