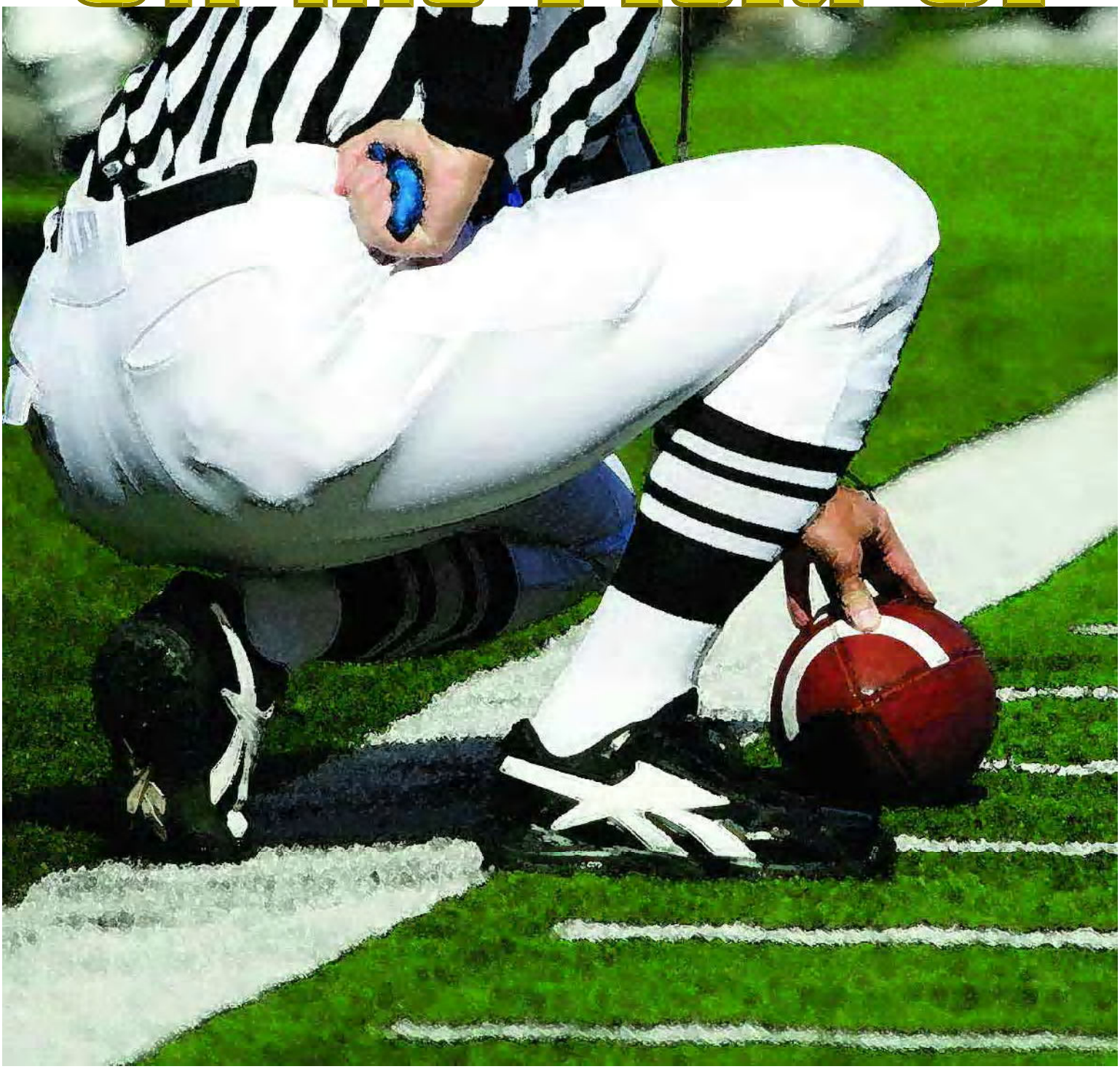
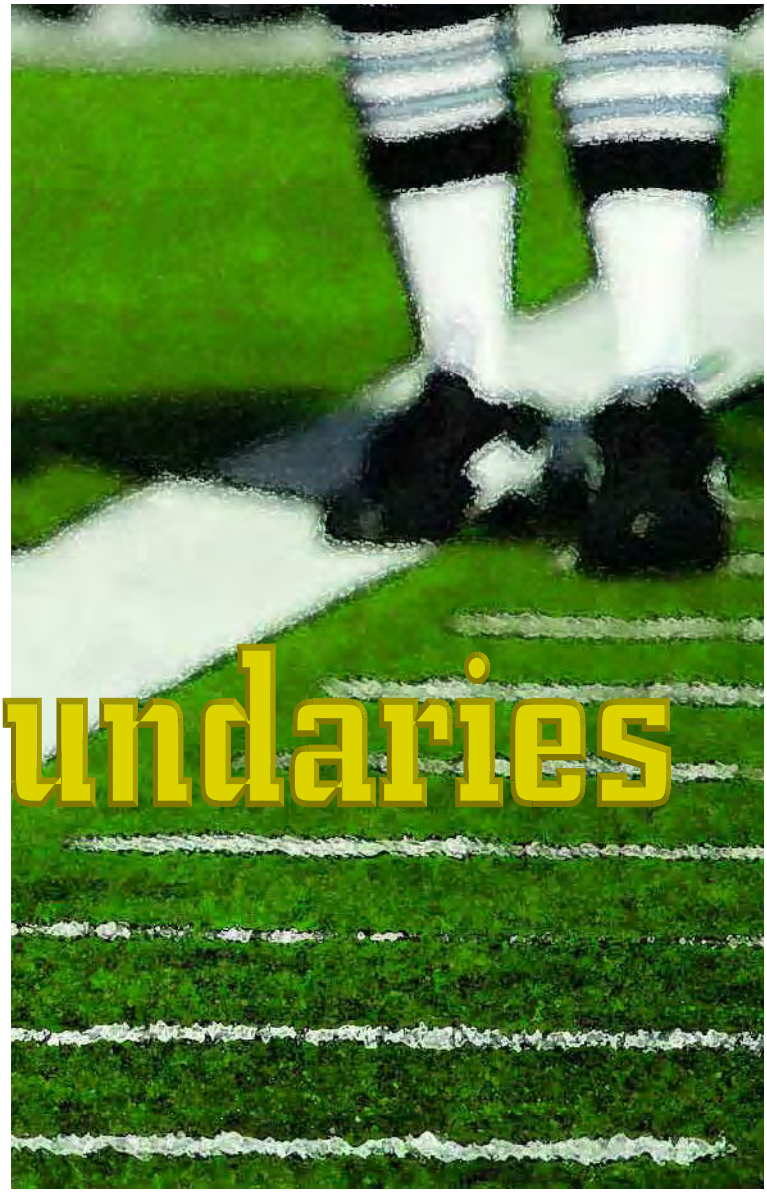
Knowing the Limits of Your Applications Will Give Your Project a Winning Season

On the Field Of



By Rob Sabourin

Finite Boundaries



Photograph by Brandon Laufenberg

Bugs—I love to find bugs! In my testing adventures, I've found a lot of great bugs. Many of the most important ones, I call *boundary bugs*. I've found many boundary bugs that are of great value and potential consequence. When I teach software testing, which I do with thousands of students and conference attendees every year, I often take a simple survey. I first ask, "How many of you use equivalence partitioning to develop tests?" Only about 5 percent raise their hands. My next question is "How many of you use boundary analysis to develop tests?" Invariably, a large percentage—generally about 80 percent—respond with a resounding "Yes, I know boundary analysis and practice it frequently."

I've found that many of my students know what boundary testing is, but are able to discover, explore and expose only a small number of potential boundary risks. So let's start to remedy that shortcoming right now. I'll examine three sources of boundary risks: those found in requirements, those related to data input, and those exposed by the processing, storage and manipulation of data. But first, I'll need to lay some groundwork.

Traditional Gridiron Testing

Equivalence partitioning is a basic test-design technique I first learned from Glenford Myers in his 1979 book "The Art of Software Testing" (Wiley released a second edition in 2004). It involves three simple steps. First, identify variables that influence the behavior of the application under test. Next, define subsets of all possible values each variable can take that the application will handle the same way. These are called *equivalence classes*. Finally, for each equivalence class, choose a representative sample value and use it to test the application.

Equivalence partitioning applies a blend of common sense and set theory to testing problems. The basic assertion of this approach is that the information you get from testing any one member of an equivalence class is the same, so you need not try more than one representative sample to test the entire class.

Traditional boundary analysis derives from Myers' notion of an equivalence class. Many equivalence classes have extreme values, minimum values, maximum values or edge conditions. We can define these extrema as boundary conditions, and use them to develop test cases. I call this *traditional equivalence-class boundary testing*. In my basic approach, for each boundary of the class, I choose three values to test the application:

1. A value exactly on the boundary
2. A value immediately within the boundary
3. A value immediately outside the boundary

So for each equivalence class, I can choose up to six tests to study the behavior of the application at the variable's boundary. Some classes have no boundaries, and some classes have only one. For example, the class of positive numbers has no upper boundary, and the class of negative numbers has no lower one. Classes about membership comprise lists of objects that do have boundaries, but in an insurance application, the class of vehicles with four-wheel drive is an equivalence class without boundaries.

I found many bugs at the boundaries of equivalence classes. These are due to the nature of requirements, design, programming and testing. Requirements that correctly delimit ranges of values are hard to elicit and unambiguously describe. Boundary bugs will continue to exist in software projects, but there are specific techniques you can use to tame them.

Study the Rule Book

Some software engineering techniques can help reduce the injection of boundary bugs into software. Software inspections can help validate our requirements, and decision tables can help clarify business rules. However, any two software development professionals referencing the requirements can interpret them in different ways.

Ambiguous requirements are full of boundary risks. Is a range inclusive or exclusive? Do ranges of values overlap? What does it mean for a value to be in range? How many dig-

its of precision do we consider in defining continuous numeric ranges?

A classic boundary-requirement bug occurred in a communications analysis project I managed a couple of years ago. We had a very small team and pretty clear requirements. Table 1 depicts a decision table that is typical of an application's requirement statements.

This decision table was used as part of the rate-management software requirements. The table indicates the discount rate, invoicing frequency and reporting frequency as a function of the type and volume of traffic. Volume bands are designated up to the amount indicated in the row labeled Monthly Volume; the assumption is volume up to and including the value indicated in the table. So when the table indicates 100,000, it implicitly refers to traffic up to and including 100,000. The volume band labeled 1,000,000 implies traffic greater than 100,000 and up to and including 1,000,000 and so on.

Although this might be an implicit ambiguity, it was common practice on the project. Bugs started showing up when we realized that developers were making different assumptions about the lower bound of the range. Different developers were responsible for the business logic behind rate computations, invoicing and report generation. Furthermore, different testers were responsible for testing the reporting and invoicing software. Testers responsible for invoicing also validated discount-rate computations. Requirements were elicited from different stakeholders depending on traffic type. Different product managers were responsible for the voice and data business segments, creating several possible points of misinterpretation of the requirements.

Each potential misinterpretation is a potential source of a requirement-based boundary risk. Requirement analysis could have misinterpreted the client needs. Requirement analysis of voice could have interpreted needs differently than a data analysis. Business logic, reporting and invoicing developers could have interpreted stated requirements differently, and invoice testers could have interpreted stated requirements differently from reporting testers.

Any misinterpretation can cause a requirement-based boundary risk.

I like to draw a tree of potential boundary misinterpretations, using a simple mind map as in Figure 1 (page 30). The tree has many branches: The requirement may be wrong, the developer may misinterpret the requirement, the tester may misinterpret the requirement, or the test results may be misinterpreted. Only one of the 32 paths through the mind map represents a correctly implemented, validated and verified boundary condition. Some cases are particularly difficult, and all derive from the nature of equivalence-related boundaries.

To avoid misinterpretation, we must develop effective boundary tests for equivalence-class boundaries that validate requirements, confirm the development team's understanding of requirements, confirm the testing team's understanding of requirements and use testing *oracles*—all strategies that help to avoid misinterpretation.

To validate requirements, I generally use software inspections to ensure consistency across the sources used to elicit them as well as consistency in the way the requirement is described.

To confirm development understanding of requirements, I often use peer reviews and unit-testing approaches that challenge assumptions developers may make about boundaries. I ensure that testing includes cases exactly on boundaries, as well as immediately within and outside of boundary conditions.

To confirm testing understanding of requirements, I review test against requirements and test against design, challenge validation of requirements, perform an ambiguity review and use peer reviews of test cases developed.

To help avoid misinterpretation of test results, I try to find testing oracles, the strategies that help to avoid misinterpretation, based on requirements. I use multiple approaches to verify results, and different sources of information to validate correctness.

Requirement-based variables can take on many different forms. Each type of variable has a different type of business logic–related classes, which in turn have different types of boundaries.

TABLE 1: PLAYBOOK FOR COMMUNICATION TRAFFIC

Condition								
Monthly Volume	100,000	1,000,000	10,000,000	>10,000,000	100,000	1,000,000	10,000,000	>10,000,000
Traffic Type	Voice	Voice	Voice	Voice	Data	Data	Data	Data
Action								
Discount Rate	0%	10%	15%	20%	0%	12.5%	15%	17.5%
Invoice	Quarterly	Monthly	Monthly	Weekly	Monthly	Monthly	Weekly	Weekly
Report	Monthly	Monthly	Weekly	Weekly	Monthly	Weekly	Weekly	Weekly

TABLE 2: FIRST STRING

Minimum Length String
Minimum -1 Length String
Minimum +1 Length String
Maximum Length String
Maximum -1 Length String
Maximum +1 Length String

First String Variables

String variables are quite common in applications and generally represent the names of objects or some sort of tests associated with objects. String variables generally have boundaries

represented by their length, with minimum and maximum lengths defining boundary test ranges. Testing boundaries of string variables generally involves the six boundary conditions listed in Table 2.

Date and time variables often have basic components of day, month, year, hour, minute, second and fractional second. Precision is an important element of a date requirement that helps to identify boundary test cases.

For example, if I had to test a data variable defined with a precision of days, I would test a day before and a day after the minimum date, and a day before and a day after the maximum date. If the precision were a millisecond, the equivalent boundary tests would be on the boundary and a millisecond before or a millisecond after the boundary date. Sub-second computations are common in real-time systems related to event synchronization and telecommunication or network switching.

Table 3 summarizes common date boundaries to consider when the date is a variable influencing the behavior or processing of an application.

Testers involved in the infamous Y2K testing initiative undoubtedly will be able to identify dozens of other relevant date and time boundaries based on a single variable.

With international Web-based applications, transactions that originate in one time zone can be processed in another. The same instant in time can be represented by different hours, minutes and days, depending on the originating and processing time zone.

Boundaries can exist in dates when transactions take place in different time zones. Boundary conditions can include cases in which transaction timing occurs at different times. Transaction timing also can "cross the top" of an hour; for example, the Newfoundland time zone is offset by 30 minutes from the Atlantic time zone. Transactions can span morning to afternoon, through days, weeks, months, years, or even centuries.

TABLE 3: FIRST-ROUND DATE BOUNDARIES

Date Component	Low Boundary	Upper Boundary
Day	First day of month	Last day of month
Month	First month of year	Last month of year
Year	First year of century	Last year of century
Year	First year of millennium	Last year of millennium
Year	Leap year	
Year	Leap century	
Hour	First hour of day	Last hour of day
Hour	First hour of morning	Last hour of morning
Hour	First hour of afternoon	Last hour of afternoon
Hour	Leap hour forward	
Hour	Leap hour backward	
Day	Leap day forward	
Day	Leap day backward	
Minute	First minute of hour	Last minute of hour
Second	First second of minute	Last second of minute

Game-Day Boundaries

Date and time ranges can be defined either with a starting point and an ending point, or with a starting point and a duration.

Starting and ending point. Boundaries exist in the absolute definition of the starting and ending points, as well as in the relationship between starting and ending points. For boundary conditions in these cases, I like to look at starting date equal to ending date, starting date just before ending date, and ending date just before starting date.

Starting point and duration. Boundaries exist in the absolute definition of the starting point as well as the absolute definition of the duration. Requirements should indicate minimum and maximum values for the duration.

Generally, date ranges run into boundaries when they cross day, hour, morning, afternoon, year, century or millennium boundaries. If the starting and ending times or dates cross time-related boundaries, there is a risk that business logic and computations that aggregate transactional date may allocate transactions to different time periods.

Compound Fractures

Compound variables are those that influence an application's behaviors in combination. For example, take a look at the dimensions and weight of an envelope used to help compute postage. Let's say the size of the envelope is comprised of length, width, thickness and weight.

I can identify boundaries based on the extreme values allowed for each component of the envelope's size, but I can also define the extremes that are the maximum of all components or the minimum of all components. Look at the object of the testing: the envelope. What's the smallest envelope? What's the largest? These define the boundaries of an envelope.

In some cases, the business logic is defined in such a way as to necessitate testing the boundaries of each component individually and in combination. But when I test compound variables, I

TABLE 4: PRESSURE IN THE POCKET

Test identifier	Length	Width	Thickness	Weight
Env001	Minimum	Minimum	Minimum	Minimum
Env002	Minimum - *E*	Minimum - *E*	Minimum - *E*	Minimum - *E*
Env003	Minimum + *E*	Minimum + *E*	Minimum + *E*	Minimum + *E*
Env004	Maximum	Maximum	Maximum	Maximum
Env005	Maximum - *E*	Maximum - *E*	Maximum - *E*	Maximum - *E*
Env006	Maximum + *E*	Maximum + *E*	Maximum + *E*	Maximum + *E*

focus on the boundaries of the objects being tested. I test boundaries with six test envelopes, as shown in Table 4. *E* represents the smallest unit that depends on the requirement for precision.

Discrete and Continuous Offense

Boundaries exist for some equivalence classes of variables. If a variable is represented by a whole number or an integer value, or can be mapped to a whole number, it's known as a *discrete variable*. An example of a discrete variable is the number of items purchased in a shopping cart application.

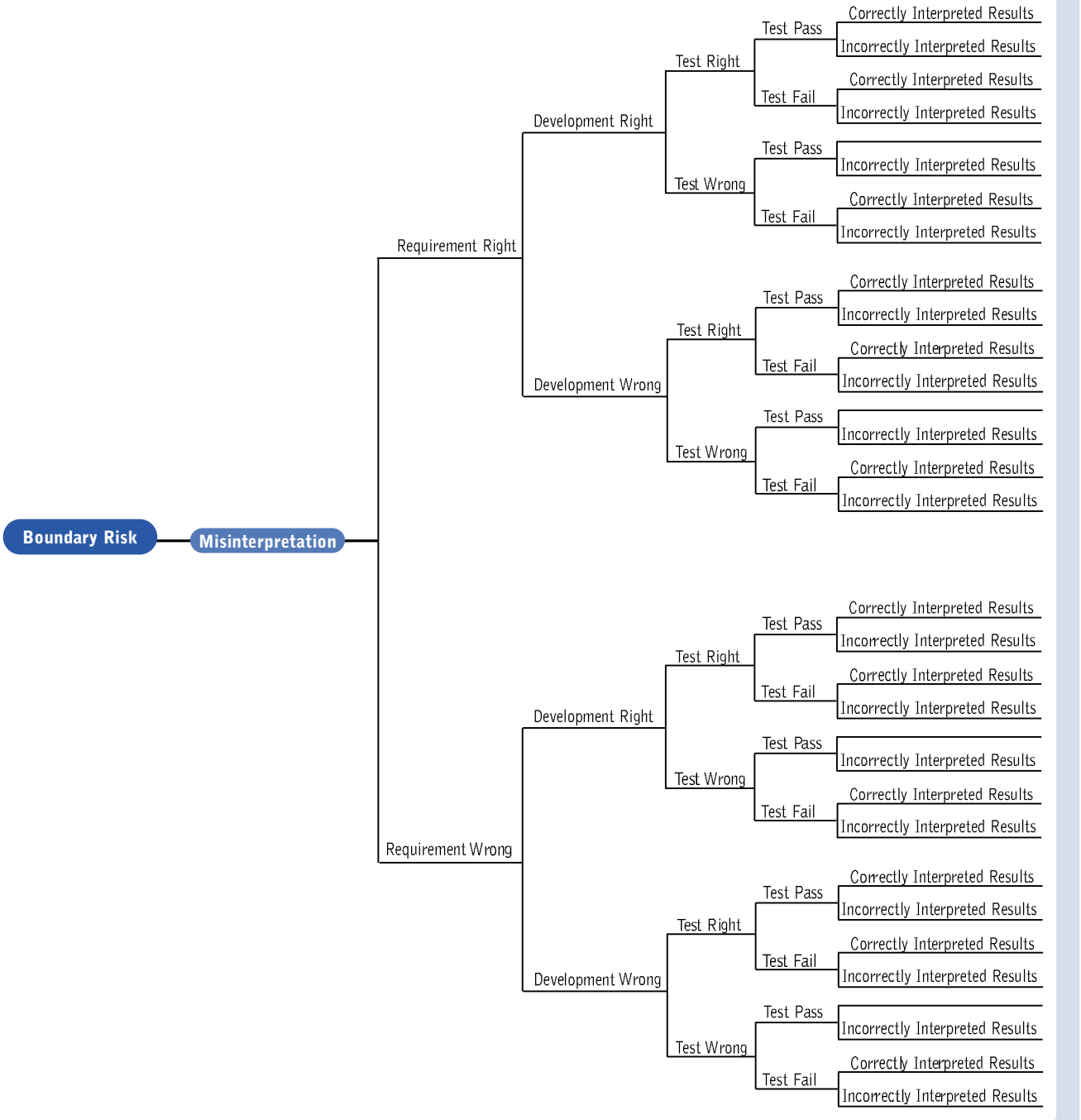
On a recent project, I performed a series of tests to confirm the business logic for order processing as a function of inventory in stock. To test the variable for order entry, I had business rules defined for a minimum and maximum order quantity as well as these rules related to the amount of inventory:

1. Order quantity must be greater than zero
2. Order quantity must be less than 256
3. If order quantity is less than or equal to inventory, the order is processed
4. If order quantity is more than or equal to inventory, a partial order is processed, sufficient inventory is reordered and completion is suspended until inventory arrives

Implementing these four rules put several interesting boundaries into play, as depicted in Table 5.

Note that in determining the inventory quantity for testing, the boundary

FIG. 1: ONE PENALTY-FREE PATH TO THE END-ZONE



You see them all the time: advertisements, newsletters, press releases and email campaigns offering you BIG software discounts, migration utilities or lofty promises to end your software licensing woes. Let's face it; many software vendors spend a lot of time and money just baiting customers to switch to their products. While most savvy consumers can see through the hype, we thought it helpful to dispel some myths and point out some realities.

Myth #1: Cheap software = Lower TCO

Reality check: The total cost of ownership (TCO) of testing comes from many sources, including: software licenses, maintenance, implementation costs, human resources, hardware, the efficiency of the testing process and more. If a vendor promises to lower your TCO through cheap software, think twice. If you don't tread carefully, the results could actually be quite expensive. As a general rule, steer clear of those who offer no real value proposition beyond fire-sale discount programs or permanent "limited" promotions.

Myth #2: Migration utilities = Easy migration

Reality check: You've seen advertisements for the migration utility du jour. They promise simple migration from one vendor's offering to another. What they DON'T mention is the significant switching costs associated with the deployment and customization of the new solution, the migration of existing data and retraining of staff to use new offerings. Don't let the conversion utility hype distract you from seeking the true capabilities and the intrinsic value of the software.

Myth #3: The price of the software is the most important selection criteria

Reality check: Try this quick experiment: Go to any résumé posting website and enter the name of your testing software in the search field. The results can show whether local skilled resources are available for that product. The fact is, labor cost is often the most expensive item in your budget. Selecting widely used technologies increases your chances of getting the right people from the local talent pool.

Myth #4: All QA and testing software companies are more or less the same

Reality check: Just because a vendor makes a claim in advertising doesn't mean it can bring great quality management software to market. Scrutinize the company carefully—you often get what you pay for. Good questions to ask are:

- Has the company been in the quality assurance (QA) software business for at least 10 years?
- Has it been able to consistently innovate and deliver new and useful technologies to market?
- Has it set the tone in QA and testing software or just followed others?
- How many customers does it have in production?
- Does the company have a vision to which it is executing, or is it just producing tactical tools?

Great differences among vendors exist in this space; identifying those differences will benefit you. Choosing the right partner can spell the difference between success and failure.

Myth #5: All QA and testing solutions are more or less the same

Reality check: No two organizations handle quality testing the same way—so it's important that your products can address the complexity of real-world enterprise environments. Use the following simple criteria to test broad vendor claims:

- Does the software have proven ability to scale from individual, project-based testing to centralized, global testing centers of excellence?
- What are the depth and breadth of environment support for applications?
- Does the testing software help you test service-oriented architecture (SOA) as well as custom and packaged applications?
- What is the quantity and quality of the ecosystem for third-party tools available for the solution?
- Is this software likely to grow and expand to support new technologies and new practices as they become available?

The HP Software difference

HP Software brings together 16 years of Mercury experience in Quality Management software with the full range of HP solutions and support. While most vendors make claims, HP Software delivers. We are the market-share leader in key categories including performance testing (77 percent, Yankee Group) and automated software quality (56 percent, IDC). More customers have selected HP Software Quality Management software and services than all other vendors combined. Why? Because our software and services are known to bring real value to customers, and that's not just a bunch of hype.

Get the details

Get the big picture, and the advantages of HP Software become increasingly clear. For more information and materials, visit **www.optimizetheoutcome.com.**

TABLE 5: INVOKING PENALTIES

Test Identifier	Condition	Below Boundary	On Boundary	Above Boundary
Tid001	Rule 1 (INV>1)	-1	0	1
Tid002	Rule 2 (INV>257)	255	256	257
Tid003	Rule 3 and 4 (INV a specific value between 1 and 256)	INV AMOUNT -1	INV AMOUNT	INV AMOUNT +1
Tid004	Rule 3 and 4 (INV=0)	-1	0	1
Tid005	Rule 3 and 4 (INV=1)	0	1	2
Tid006	Rule 3 and 4 (INV=255)	254	255	256
Tid007	Rule 3 and 4 (INV=256)	255	257	257
Tid008	Rule 3 and 4 (INV=257)	256	257	258

related to inventory varies depending on the amount of inventory. For rules 1 and 2, I would test the boundaries by ensuring the associated inventory was greater than the order quantity. However, for rules 3 and 4, I would explicitly vary the inventory and the order quantity.

Continuous variables are those that can have any possible real number in some range. Continuous variables have the interesting characteristic that there always exists a third value between any two possible values. There are an infinite number of possible values. If you look at the Windows Calculator and try to test the square root function, you'll observe that any real number can be an input value. The valid range of values for the square root function is positive real numbers including zero.

When you're working with systems that have a continuous range of values, it's easy to identify the boundary values but it becomes challenging to identify values a little above and a little below boundaries.

This is where the notion of precision is critical. An application's precision may be indicated in the requirement documentation or may be part of the system design. To unearth the relevant information for boundary testing, you must identify the smallest value that the application can process such that if you take a continuous variable, the following is true:

Let EPSILON be the smallest represented value such that

VARIABLE + EPSILON > VARIABLE

VARIABLE – EPSILON < VARIABLE

If the variable can have different magnitudes, the EPSILON value is generally defined to be a function of the magnitude of the value. For example, if the value is a very large number such as 10**100 (1 followed by 100 zeros, a.k.a. a *googol*), the value of EPSILON may well be a value of 10*30. However, if the same variable has a value with a smaller number like 1,000, EPSILON may well be defined as 0.000001. Therefore, boundaries of continuous variables depend on the magnitude of the variable, and different values of EPSILON should be used for different orders of magnitude. I run into the issue of different EPSILON values when involved in scientific calculations.

Calling Plays From The Sideline

So far we've explored boundary risks related to variables derived from software requirements and business rules. These boundaries are critical and can be identified and explored based on discovering and understanding what the application should do.

In parallel, I generally identify a series of boundary risks that relate to input—how variables get populated by data before processing begins.

Variables get data from many different sources. Users can enter it though user interfaces, forms, menus, dialogs and controls. Messages can be exchanged between processes and applications, and also between the operating system and the application. As well, variables may be populated from data sources, persistent storage, files, databases and registries.

The way variables get populated leads to a series of potential boundary risks that are different from those related to business rules. The input boundaries that I find in testing products are often the most critical.

The input boundaries found in testing products are often the most critical.

Whenever a form is populated on a screen or dialog, operators are asked to enter data before processing can occur. The fields on the display may have different restrictions or rules based on user interface requirements. Some of the common constraints I run into in discovering input boundaries include the following: For string length, what is the acceptable length of a field? For white space, how much leading and trailing white spaces are allowed? Are there restrictions in the number of digits? What is the number of significant digits or precision; for example, how many values after the decimal point are computed?

For each of these constraints, I consider identifying boundaries related to what the rules are and what I can do. For example, a field may expect a user name of between six and 12 characters. There may be user interface constraints that restrict the field length to this range.

I also ask myself what I can possibly enter into the field; for example, can I enter a string of length 0 (null string)? Can I enter a long string longer than the acceptable range? Can I enter an extremely large string (perhaps a BLOB [Binary Large Object]) into the field? The boundaries related to these extremes could expose potential failure modes of the application that can lead to discovering security breaches.

Numeric precision in input fields relates to the number of digits that will be actually processed. Interesting boundaries show up based on the number of significant digits.

For example, if an application processes values of up to four decimal places, I would identify input-based test cases related to entry of data with three, four or five decimal places.

I might also see if the application attempts to validate, process or ignore these values. If the values are validated, I'd expect the application to warn the

user that an attempt was made to exceed the maximum precision supported. If the application processes the values, I'd expect it to round them up or down to the nearest value of the appropriate precision.

For example, 10.01256 with four digits of precision rounds up to 10.0126. When the application does the rounding, it becomes interesting to test and ensure that processing is of the rounded value and not the entered value. Processing for 10.01256 and 10.01259 both round up to 10.0126 and thus should generate the same result. If not, you may have identified a precision-related boundary bug.

Memory, Storage and Processing: A Head Coach's Headache

The way data is stored in computer memory or databases can lead to interesting boundary conditions that are independent of the other two types.

Table 6 lists valid ranges of values for common data types in the ANSI-C programming language.

If we can find out how an integer is stored in memory, it may be interesting to explore how the application behaves when the boundaries are determined by the range of the variable type.

We can learn about the types of variables for any string, numeric, currency, date or other compound data type by consulting with the software developers, checking database schemas or actually reviewing the code. If a quantity value is stored as an unsigned short integer, I'd want to explore how it processes values around the lower possible boundary, 0, and the higher possible boundary, 65,635; perhaps computational overflow or underflow bugs would occur.

TABLE 6: ZONE DEFENSE

Type	Min	Max
unsigned char	0	255
short int	-32,767	32,767
unsigned short	0	65,535
int	-32,767	32,767
unsigned int	0	65,535
long int	-2,147,483,647	2,147,483,647
unsigned long	0	4,294,967,295

Boundaries related to data type are also significant when programming languages attempt to convert data from one type to another. In C, an integer variable may be cast into a char type variable, thus dramatically impacting the range. An unsigned short has a maximum value of 65,535, whereas an unsigned char has a maximum value of 255.

Code process review also leads to some interesting boundaries. For example, imagine an algorithm that takes values *A* and *B* and multiplies them, placing the result in a variable *C*:

$$C = A * B$$

The variables *A*, *B* and *C* are of the type unsigned short. Each variable can store values between 0 and 65,535. There are many combinations of data stored in *A* and *B* whose product would be above the maximum value stored in *C*. A serious bug could occur if an attempt is made to overflow the variable *C*. Such problems could result in incorrect computations, corruptions of memory or both.

The boundaries of interest in testing are combinations of A and B that result in values on the line depicted in Figure 2. I generally do the type of "white box" analysis required to discover processing boundaries in collaboration with the software development staff. I then explore application behavior at the memory-related processing boundaries to make sure the application considers the possibility that we overflow an internal boundary condition as an internal intermediate step in processing.

Any data type has different boundaries, and occasional restrictions and ranges vary across tool vendors, technologies and operating environments. To find and test these memory and processing variables, you'll have to get your hands dirty, but you'll be helping to ensure that the code is healthy.

These boundaries are sometimes considered hidden since they can't be discovered by merely looking at the application from the outside, studying the requirements or exploring how data can be input into the application being tested.

Highlight Reel

Here, I've reviewed software boundaries from traditional sources based on data requirements, input and processing. Next month, in part 2, I'll relate further tales of exposing and discovering boundary-related bugs, and share some systematic and exploratory testing techniques to unearth them.

FIG. 2: AN UNSIGNED (SHORT) FREE AGENT