6

June, 2009

testing experience

The Magazine for Professional Testers

ISSN 1866-5705 www.testingexperience.com free digital version print version 8,00 € printed in Germany

Security Testing

© iStockphoto/alexandru_lamba

© iStockphoto.com/kutaytanir

Advanced Software Test Design Techniques, Decision Tables and Cause-Effect Graphs

by Rex Black

Introduction

The following is an excerpt from my recently-published book, *Advanced Software Testing: Volume 1*. This is a book for test analysts and test engineers. It is especially useful for ISTQB Advanced Test Analyst certificate candidates, but contains detailed discussions of test design techniques that any tester can—and should—use. In this first article in a series of excerpts, I start by discussing the related concepts of decision tables and cause-effect graphs.

Decision Tables

Equivalence partitioning and boundary value analysis are very useful techniques. They are especially useful when testing input field validation at the user interface. However, lots of testing that we do as test analysts involves testing the business logic that sits underneath the user interface. We can use boundary values and equivalence partitioning on business logic, too, but three additional techniques, decision tables, use cases, and state-based testing, will often prove handier and more powerful techniques.

In this article, we start with decision tables. Conceptually, decision tables express the rules that govern handling of transactional situations. By their simple, concise structure, decision tables make it easy for us to design tests for those rules, usually at least one test per rule.

When I said "transactional situations," what I meant was those situations where the conditions—inputs, preconditions, etc.—that exist at a given moment in time for a single transaction are sufficient by themselves to determine the actions the system should take. If the conditions are not sufficient, but we must also refer to what conditions have existed in the past, then we'll want to use state-based testing, which we'll cover in a moment.

The underlying model is either a table (most typically) or a Boolean graph (less typically). Either way, the model connects combinations of conditions with the action or actions that should occur when each particular combination of conditions arises.

If the graph is used, this technique is also referred to as a cause-effect graph, because that is the formal name of the graph. However, it's important to keep in mind that any given decision table can be converted into a cause-effect graph, and any given cause-effect graph can be converted into a decision table. So, which one you choose to use is up to you. I prefer decision tables, and they are more commonly used, so I'll focus on that here. However, I'll show you how the conversion can be done.

To create test cases from a decision table or a cause-effect graph, we design test inputs that fulfill the conditions given. The test outputs correspond to the action or actions given for that combination of conditions. During test execution, we check that the actual actions taken correspond to the expected actions.

We create enough test cases that every combination of conditions is covered by at least one test case. Frequently, that coverage criterion is relaxed to say, we cover those combinations of conditions that can determine the action or actions. If that's a little confusing, the distinction I'm drawing will become clear to you when we talk about collapsed decision tables.

With a decision table, the coverage criterion boils down to an easy-to-remember rule of at least one test per column in the table. For cause-effect graphs, you have to generate a so-called "truth table" that contains all possible combinations of conditions and ensure you have one test per row in the truth table.

So, what kind of bugs are we looking for with decision tables? There are two. First, under some combination of conditions, the wrong action might occur. In other words, there is some action that the system is not to take under this combination of conditions, yet it does. Second, under some combination of conditions, the system might not take the right action. In other words, there is some action that the system is to take under this combination of conditions, yet it does not.

Consider an e-commerce application like the one found on our RBCS Web site, www.rbcs-us.com. At the user interface layer, we need to validate payment information, specifically credit card type, card number, card security code, expiration month, expiration year, and cardholder name. You can use boundary value analysis and equivalence partitioning to test the ability of the application to verify the payment information, as much as possible, before sending it to the server.

So, once that information goes to the credit card processing company for validation, how can we test that? Again, we could handle that with equivalence partitioning, but there are actually a whole set of conditions that determine this processing:

Does the named person hold the credit card entered, and is the other information correct?

Is it still active or has it been cancelled?

Is the person within or over their limit?

Is the transaction coming from a normal or a suspicious location?

The decision table in Table 1 shows how these four conditions interact to determine which of the following three actions will occur:

Should we approve the transaction?

Should we call the cardholder (e.g., to warn them about a purchase from a strange place)?

Should we call the vendor (e.g., to ask them to seize the cancelled card)?

Take a minute to study the table to see how this works. The conditions are listed at the top left of the table, and the actions at the bottom left. Each column to the right of this left-most column contains a business rule. Each rule says, in essence, "Under this particular combination of conditions (shown at the top of the rule), carry out this particular combination of actions (shown at the bottom of the rule)."

Notice that the number of columns—i.e., the number of business rules—is equal to 2 (two) raised to the power of the number of conditions. In other words, 2 times 2 times 2 times 2, which is 16. When the conditions are strictly Boolean—true or false—and we're dealing with a full decision table (not a collapsed one), that will always be the case.

Conditions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Real account?	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
Active account?	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N
Within limit?	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
Location okay?	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Actions																
Approve?	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Call cardholder?	N	Y	Y	Y	N	Y	Y	Y	N	N	N	N	N	N	N	N
Call vendor?	N	N	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Table 1: Decision Table Example (Full)

Did you notice how I populated the conditions? The topmost condition changes most slowly. Half of the columns are Yes, then half No. The condition under the topmost changes more quickly but more slowly than all the others. The pattern is quarter Yes, then quarter No, then quarter Yes, then quarter No. Finally, for the bottommost condition, the alternation is Yes, No, Yes, No, Yes, etc. This pattern makes it easy to ensure you don't miss anything. If you start with the topmost condition, set the left half of the rule columns to Yes and the right half of the rule columns to No, then following the pattern I showed, if you get to the bottom and the Yes, No, Yes, No, Yes, etc., pattern doesn't hold, you did something wrong.

Deriving test cases from this example is easy: Each column of the table produces a test case. When the time comes to run the tests, we'll create the conditions which are each test's inputs. We'll replace the "yes/no" conditions with actual input values for credit card number, security code, expiration date, and cardholder name, either during test design or perhaps even at test execution time. We'll verify the actions which are the test's expected results.

In some cases, we might generate more than one test case per column. I'll cover this possibility in more detail later, as we enlist our previous test techniques, equivalence partitioning and boundary value analysis, to extend decision table testing.

Collapsing Columns in the Table

Notice that, in this case, some of the test cases don't make much sense. For example, how can the account not be real but yet active? How can the account not be real but within limit? This kind of situation is a hint that maybe we don't need all the columns in our decision table.

We can sometimes collapse the decision table, combining columns, to achieve a more concise—and in some cases sensible—decision table. In any situation where the value of one or more particular conditions can't affect the actions for two or more combinations of conditions, we can collapse the decision table.

This involves combining two or more columns where, as I said, one or more of the conditions don't affect the actions. As a hint, combinable columns are often **but not always** next to each other. You can at least start by looking at columns next to each other.

To combine two or more columns, look for two or more columns that result in the same combination of actions. Note that the actions must be the same for all of the actions in the table, not just some of them. In these columns, some of the conditions will be the same, and some will be different. The ones that are different obviously don't affect the outcome. So, we can replace the conditions that are different in those columns with the dash character ("-"). The dash usually means either I don't care, it doesn't matter, or it can't happen, given the other conditions.

Now, repeat this process until the only further columns that share the same combination of actions for all the actions in the table are ones where you'd be combining a dash with Yes or No value and thus wiping out an important distinction for cause of action. What I mean by this will be clear in the example I present in a moment, if it's not clear already.

Another word of caution at this point: Be careful when dealing with a table where more than one rule can apply at one single point in time. These tables have non-exclusive rules. We'll discuss that further later in this section.

Table 2 shows the same decision table as before, but collapsed to eliminate extraneous columns. Most notably, you can see that columns 9 through 16 in the original decision table have been collapsed into a single column.

Conditions	1	2	3	5	6	7	9
Real account?	Y	Y	Y	Y	Y	Y	N
Active account?	Y	Y	Y	N	N	N	-
Within limit?	Y	Y	N	Y	Y	N	-
Location okay?	Y	N	-	Y	N	-	-
Actions							
Approve?	Y	N	N	N	N	N	N
Call cardholder?	N	Y	Y	N	Y	Y	N
Call vendor?	N	N	N	Y	Y	Y	Y

Table 2: Decision Table Example (Collapsed)

I've kept the original column numbers for ease of comparison. Again, take a minute to study the table to see how I did this. Look carefully at columns 1, 2, and 3. Notice that we can't collapse 2 and 3 because that would result in "dash" for both "within limit" and "location okay." If you study this table or the full one, you can see that one of these conditions must **not** be true for the cardholder to receive a call. The collapse of rule 4 into rule 3 says that, if the card is over limit, the cardholder will be called, regardless of location. The same logic applies to the collapse of rule 8 into rule 7.

Notice that the format is unchanged. The conditions are listed at the top left of the table, and the actions at the bottom left. Each column to the right of this left-most column contains a business rule. Each rules says, "Under this particular combination of conditions (shown at the top of the rule, some of which might not be applicable), carry out this particular combination of actions (shown at the bottom of the rule, all of which are fully specified)."

Notice that the number of columns is no longer equal to 2 raised to the power of the number of conditions. This makes sense, since otherwise no collapsing would have occurred. If you are concerned that you might miss something important, you can always start with the full decision table. In a full table, because of the way you generate it, it is guaranteed to have all the combinations of conditions. You can mathematically check if it does. Then, carefully collapse the table to reduce the number of test cases you create.

Also, notice that, when you collapse the table, that pleasant pattern of Yes and No columns present in the full table goes away. This is yet another reason to be very careful when collapsing the columns, because you can't count on the pattern or the mathematical formula to check your work.

Cause-Effect Graphs

When collapsing a decision table, a cause-effect graph can help you make sure you don't accidentally collapse columns you shouldn't. Some people like to use them for test design directly, but, as I mentioned earlier, I'm not too fond of trying to do that.

The process for creating a cause-effect graph from a decision table or decision table from a cause-effect graph is straightforward. To create a cause-effect graph from a decision table, first list all the conditions on the left of a blank page. Next, list all the actions on right of a blank page. Obviously, if there are a lot of conditions and actions, this will be a big page of paper, but then your decision table would be big, too.

Now, for each action, read the table to identify how combinations of conditions cause an action. Connect one or more conditions with each action using Boolean operators, which I show in Figure 1. Repeat this process for all actions in the decision table.

If you happen to be given a cause-effect graph and want to create a decision table, first list all the conditions on the top left of a "blank" decision table. Next, list all the actions on the bottom left of the decision table, under the conditions. Following the pattern shown earlier, generate all possible combinations of conditions. Now, referring to the cause-effect graph, determine the actions taken and not taken for each combination of conditions. Once the actions section is fully populated, you can collapse the table if you'd like.

In Figure 1, you see the cause-effect graph that corresponds to the example decision tables we've looked at so far. You might ask, "Which one, the full or collapsed?" Both. The full and the collapsed are logically equivalent, unless there's something wrong with the collapsed version.

At the bottom left of this figure, you see the legend that tells you how to read the operations. Let's go clockwise from the top left of the legend.

We have simple causality: If A is true, B will occur, or, in other words, A causes B.

We have negation: When A is not true, B will occur, or, not A causes B.

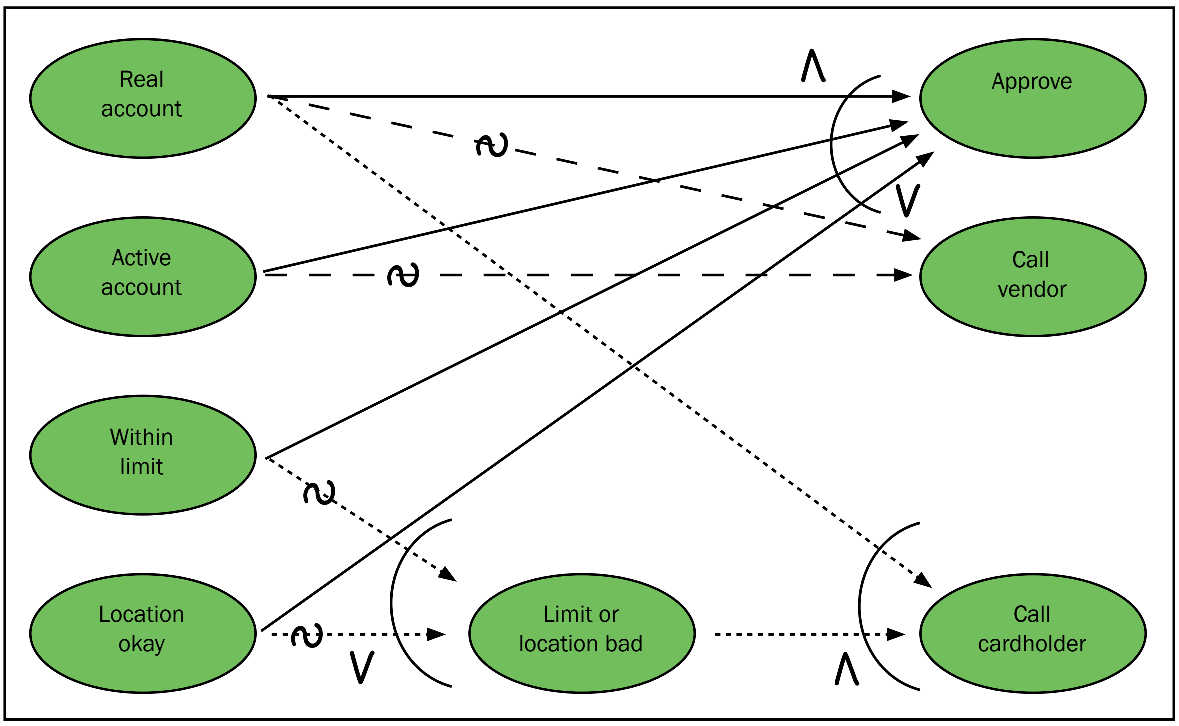
We have AND operation: When A1 and A2 are both true, B will occur, or A1 and A2 causes B.

We have OR operation: When A1 or A2 is true, B will occur, or A1 or A2 causes B.

Let's look at the connection between conditions and actions. The solid causality lines, together with an AND operator, show that all four conditions must be met for the transaction to be approved.

The dashed causality lines, together with negation operators and an OR operator, show that, if the account is not real or the account is not active, we will call the vendor.

The dotted causality lines are a bit more complicated. First, we combine the "within limit" and "location okay" conditions, with negation operators and an OR operator, to create an intermediate condition of "Limit or location



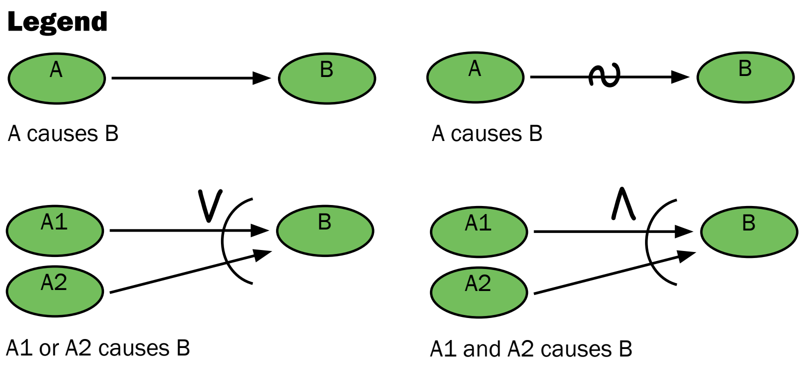


Figure 1: Cause-Effect Graph Example

bad". Now, we combine that with the "real account" condition to say that, if we have an over-limit or bad location situation, and the account is real, we will call the cardholder.

Combining Decision Table Testing with Other Techniques

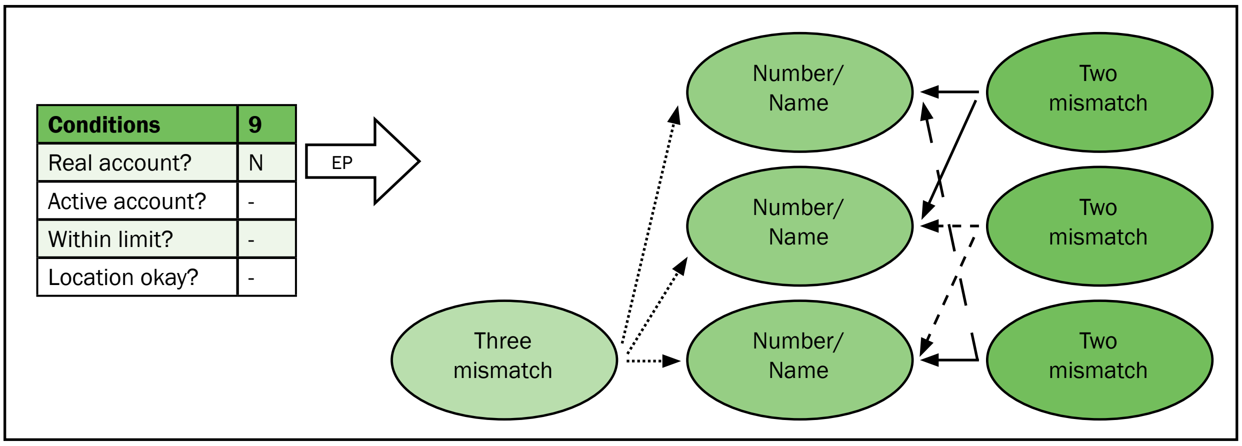
Let's address an issue I brought up earlier, the possibility of multiple test cases per column in the decision table via the combination of equivalence partitioning with the decision table technique. Let's refer back to our example decision table shown in Table 1, specifically column 9.

We can apply equivalence partitioning to the question of, "How many interesting—from a test point of view—ways are there to have an account not be real?" As you can see from Figure 2, this could happen six potentially interesting ways:

- Card number and cardholder mismatch
- Card number and expiry mismatch
- Card number and CSC mismatch
- Two of the above mismatches (three possibilities)
- All three mismatches.

So, there could be seven tests for that column.

How about boundary value analysis? Yes,



Conditions	9
Real account?	N
Active account?	-
Within limit?	-
Location okay?	-

Figure 2: Equivalence Partitions and Decision Tables

you can apply that to decision tables to find new and interesting tests. For example, "How many interesting test values relate to the credit limit?"

As you can see from Figure 3, equivalence partitioning and boundary value analysis show us six interesting possibilities:

- The account starts at zero balance
- The account would be at a normal balance after transaction
- The account would be exactly at the limit after the transaction
- The account would be exactly over the limit after the transaction
- The account was at exactly the limit before the transaction (which would ensure going over if the transaction concluded)
- The account would be at the maximum overdraft value after the transaction (which might not be possible)

Combining this with the decision table, we can see that would again end up with more "over limit" tests than we have columns—one more, to be exact—so we'd increase the number of tests just slightly. In other words, there would be four within-limit tests and three over-limit tests. That's true unless you wanted to make sure that each within-limit equivalence class was represented in an approved transaction, in which case column 1 would go from one test to three.

Non-Exclusive Rules in Decision Tables

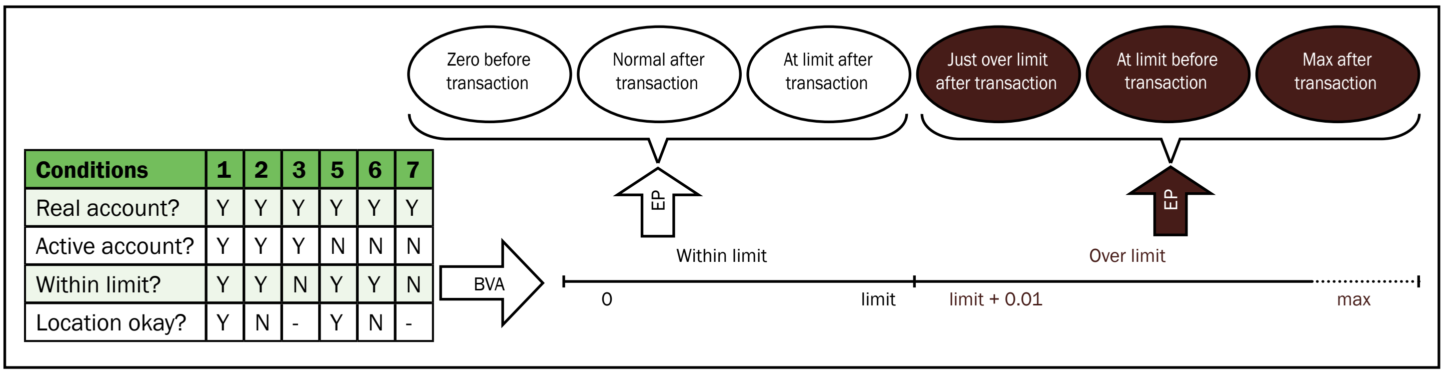
Let's finish our discussion about decision tables by looking at the issue of non-exclusive rules I mentioned earlier. Sometimes more than one rule can apply to a transaction. In Table 3, you see a table that shows the calculation of credit card fees. There are three

Conditions	1	2	3
Foreign exchange?	Y	-	-
Balance forward?	-	Y	-
Late payment?	-	-	Y
Actions			
Exchange fee?	Y	-	-
Charge interest?	-	Y	-
Charge late fee?	-	-	Y

Table 3: Non-exclusive Rules Example

conditions, and notice that zero, one, two, or all three of those conditions could be met in a given month. How does this situation affect testing? It complicates the testing a bit, but we can use a methodical approach and risk-based testing to avoid the major pitfalls.

To start with, test the decision table like a normal one, one rule at a time, making sure that no conditions not related to the rule you are testing are met. This allows you to test rules in isolation—just like you are forced to do in situations where the rules are exclusive. Next, consider testing combinations of rules. Notice I said, "consider," not "test all possible combinations of rules." You'll want to avoid combinatorial explosions, which is what happens when testers start to test combinations of factors without consideration of the value of those tests. Now, in this case, there are only eight possible combinations—three factors,



Conditions	1	2	3	5	6	7
Real account?	Y	Y	Y	Y	Y	Y
Active account?	Y	Y	Y	N	N	N
Within limit?	Y	Y	N	Y	Y	N
Location okay?	Y	N	-	Y	N	-

Figure 3: Boundary Values and Decision Tables

two options for each factor, 2 times 2 times 2 is 8. However, if you have six factors with five options each, you now have 15,625 combinations.

One way to avoid combinatorial explosions is to identify the possible combinations and then use risk to weight those combinations. Try to get to the important combinations and don't worry about the rest. Another way to avoid combinatorial explosions is to use techniques like classification trees and pairwise testing (see my books *Advanced Software Testing: Volume 1* or *Pragmatic Software Testing* for a further discussion on those techniques).

Conclusion

In this article, I've shown how to apply decision tables and cause-effect graphs to the testing of sophisticated and complex internal business logic in applications. Decision tables are a great way to test detailed business rules in isolation, especially for transactional types of situations. However, we will need to look at two additional techniques, use cases and state-based test techniques, to deal with the full range of internal business logic testing we need to do. I'll address those techniques in the next two articles in this series.

Biography

With a quarter-century of software and systems engineering experience, Rex Black is President of RBCS (www.rbcs-us.com), a leader in software, hardware, and systems testing. For over a dozen years, RBCS has delivered services in consulting, outsourcing and training for software and hardware testing. Employing the industry's most experienced and recognized consultants, RBCS conducts product testing, builds and improves testing groups and hires testing staff for hundreds of clients worldwide. Ranging from Fortune 20 companies to start-ups, RBCS clients save time and money through improved product development, decreased tech support calls, improved corporate reputation and more. As the leader of RBCS, Rex is the most prolific author practicing in the field of software testing today. His popular first book, Managing the Testing Process, has sold over 35,000 copies around the world, including Japanese, Chinese, and Indian releases. His five other books on testing, Advanced Software Testing: Volume I, Advanced Software Testing: Volume II, Critical Testing Processes, Foundations of Software Testing, and Pragmatic Software Testing, have also sold tens of thousands of copies, including Hebrew, Indian, Chinese, Japanese and Russian editions. He has written over thirty articles, presented hundreds of papers, workshops, and seminars, and given about thirty keynote speeches at conferences and events around the world. Rex has been President of the International Software Testing Qualifications Board and is a Director of the American Software Testing Qualifications Board.

Testing Services and Outsourcing

RSI Test Factory

Tradition:
15+ years of experience
500+ highly qualified professionals
3 million+ hours in QA projects

Test and Project Methodology:
International standards compliance
Web Portal for Project Portfolio Management
Customizable approaches:
business, requirements, risks...

Best Practices:
Highly reusable testware
Large scale production capacity
Charging models: FPA, UUCP, test case...
Competitive costs

www.rsinet.com.br