Chapter 3

Test Rules: Build the Backbone for Effective Testing

The core of our approach is to start testing as early as possible. We do this by asking for the acceptance criteria for each requirement. For instance, in Chapter 1 we used an example of someone looking for a means of transport. His acceptance criteria might be:

- It has to be red.
- It can travel 100 miles an hour.
- It has a convertible roof.
- It looks like a Porsche Carrera.

Acceptance criteria are what we call the test rules.

Test cases, which include test rules, are the basis for all testing activities, independent of which testing phase they occur in.

A test case consists of the test rule and the test data. This differentiation is important, because the same test rule can be used with different sets of test data to form different test cases. But from a coverage perspective, these various test cases can prove or test the same functionality.

Test case = test rule + test data

Test rule = a determined operation that leads to a defined result

Test data = data needed with the test rule to execute and evaluate a test

Using another example, validating a purchase order, we check the age of a purchaser, who must be 18 years or older for the purchase to be accepted. To execute the test, we need two test rules to verify the correct implementation:

- A purchaser's birth date is 18 years or more in the past
- A purchaser's birth date is less than 18 years in the past

These two test rules cover the described functionality completely, so we have 100 percent functional coverage. However, to get executable test cases, we need to add test data — in this case, birth dates -- one for each rule.

The two dates are the least amount of test data we need for functionality. We could create more dates to execute the test with, and each date would then lead to a new test case, but it would not lead to better coverage of the functionality to be tested. (Testing the same function represented by the test rule with more data points makes sense, because it builds more confidence for the technical implementation and increases the data coverage of the test, but it does not increase the functional coverage.)

Functional coverage: The degree to which each business rule is executed

Data coverage: The degree to which a function is executed with the possible data combinations

Because testing is always a balance between the security we want for a quality application and the effort we can afford spend, it is up to the test manager to define the quality goals for functional coverage and data coverage for the application under test.

Differentiating between test rule and test data is helpful for achieving various quality goals, and it enables managers to split the test work among the stakeholders allowing an early start for test activities.

However, this chapter is about how to get to test rules efficiently. Before we start, let's go back to an aspect of testing we discussed in Chapter 1. There, we pointed out that various quality goals are achieved through different test phases. The test rules we're developing here prove the functional correctness of the application, so they're used in these test phases:

- Requirement Verification
- Integration testing
- User Acceptance testing

The final test case could describe other test phases where it makes sense to reuse the rules, but the goal of these test rules is to verify functional correctness. (As we'll see later, we won't be able to spend the effort for test-rule development that leads to 100 percent functional coverage for all the business functions of the application. We introduce an approach that would ensure 100 percent functional coverage, but we'll discuss how this can be trimmed and so reduce the effort for the test-rule development.)

Equivalence Partitioning with Root Cause Analysis

The approach we use for test-rule determination is a combination of equivalence partitioning and root cause analysis.

Equivalence partitioning is a software-testing technique designed to:

- Reduce the number of test cases to a necessary minimum
- Select the right test cases to cover all possible scenarios

Requirements are the basis for test-rule definition. In Chapter 2, we recommended clustering the requirements, which results in what we call "business functions." These are the activities that describe the end user's steps that the application needs to support.

In the first step, the test rules are defined separately for each business function. This achieves a high level of parallelization and independence during test preparation, test execution, and test evaluation, which allows us to start testing at the earliest possible time.

To show how the approach works, let's use the example of the business function, "date."

Date is a standalone functionality that checks if the 8-digit number entered is a valid date.

- We assume the proper entry format for the date is DD MM YYYY.
- We assume that only numbers can be entered.

With this specification we proceed to the test rule definition:

STEP 1: Identifying the data entry fields

DD

MM

YYYY

STEP 2: Equivalence partitioning

Data Field Name	Data Entry	Expected Result
DD	00, 32–99	Invalid date
	00–28	Invalid date
	29	Depends
	30	Depends
	31	Depends

Equivalence partitioning is based on the whole set of data that's possible to enter into the field. In the DD fields, it's the numbers from 00 to 99. Next, we split this set into subsets in which each subset has a clearly defined result with respect to the requirement.

- In our example, 00 and 32 will not be valid days, and so all members of this subset will give the result "Invalid date."
- The numbers 01-28 will always be valid dates.
- The numbers 29, 30, and 31 depend on the month and the year entered to determine whether the date is valid.

Data Field Name	Data Entry	Expected Result
MM	00, 13–99	Invalid date
	02	Depends
	01, 03, 05, 07, 08, 10, 12	Depends
	04, 06, 09, 11	Depends

The set of possible data for the MM field is again the numbers 00 to 99.

- The subset 00 and 13 - 99 will result in an "Invalid date" response, and the rest will result in "Depends."

- With some subject matter knowledge, we can split this subset even further:
  - February (02) can be 28 or 29, depending on whether it occurs in a leap year.
  - All other months can be split into two subsets, one with 30-day months and the other with 31-day months.

The subject matter knowledge we used in this case is important for splitting the data into further subsets. So, whenever you split data into subsets with a common expected result, you should get input from the end user -- in most cases you'll gain information that's not documented in the requirement.

Data Field Name	Data Entry	Expected Result
YYYY	Leap year	Depends
	Non-leap year	Depends

The YYYY field is split into two subsets:

- All the leap years
- All the non-leap years

You'll notice there are no invalid years. Often, dates before 1900 would be considered invalid, but our specification has no such rule. However, we could discuss this with the business users of the application.

If we put all this together, we come to the following overview:

Data Field Name	Data Entry	Expected Result
DD	00, 32–99	Invalid date
	00–28	Invalid date
	29	Depends
	30	Depends
	31	Depends
MM	00, 13–99	Invalid date
	02	Depends
	01, 03, 05, 07, 08, 10, 12	Depends
	04, 06, 09, 11	Depends
YYYY	Leap year	Depends
	Non-leap year	Depends

STEP 3: Resolving the dependencies

When we specify the expected result for each subset, we sometimes need more information before we can determine that result. We've indicated the demand for more information by entering "Depends" in the expected result field. We'll collect information for the fields that need to be evaluated, so we can determine the expected result.

In our *date* example, this leads to the following result:

Data Field Name	Data Entry	Expected Result
DD	29	Depends
	30	Depends
	31	Depends
MM	02	Depends
	01, 03, 05, 07, 08, 10, 12	Depends
	04, 06, 09, 11	Depends
YYYY	Leap year	Depends
	Non-leap year	Depends

We'll consider all three data fields to determine the expected result. To limit the complexity of the test case development process, try to limit the number of subsets for data entry for each data field to the absolute minimum possible. The more subsets you select, the more test cases you have to draw from.

STEP 4: Creating combinations

To identify all possible rules that describe the dependency of the three fields (day DD, month MM, and year YYYY), we combine the three subsets of data with each other.

There are 18 combinations possible:

		Combinations																	
Data Field Name	Data Entry	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
DD	29	✓			✓			✓			✓			✓			✓		
	30		✓			✓			✓			✓			✓			✓	
	31			✓			✓			✓			✓			✓			✓
MM	02	✓	✓	✓							✓	✓	✓						
	01, 03, 05, 07, 08. 10, 12				✓	✓	✓	✓	✓	✓				✓	✓	✓	✓	✓	✓
	04, 06, 09, 11							✓	✓	✓							✓	✓	✓
YYYY	Leap year	✓	✓	✓	✓	✓	✓	✓	✓	✓									
	Non-leap year										✓	✓	✓	✓	✓	✓	✓	✓	✓

In this matrix, we have combined each subset of each data field with the subsets of all the other data fields.

STEP 5: Adding expected results to each combination

Now we can determine the expected results for each combination. Two different results are possible:

- R1: valid date
- R2: invalid date

If we add this information to the table, we get the following:

		Combinations																	
Data Field Name	Data Entry	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
DD	29	✓			✓			✓			✓			✓			✓		
	30		✓			✓			✓			✓			✓			✓	
	31			✓			✓			✓			✓			✓			✓
MM	02	✓	✓	✓							✓	✓	✓						
	01, 03, 05, 07, 08. 10, 12				✓	✓	✓							✓	✓	✓	✓	✓	
	04, 06, 09, 11							✓	✓	✓							✓	✓	
YYYY	Leap year	✓	✓	✓	✓	✓	✓	✓	✓	✓									
	Non-leap year										✓	✓	✓	✓	✓	✓	✓	✓	✓
Expected result		R1	R2	R2	R1	R1	R1	R1	R1	R2	R2	R2	R2	R1	R1	R1	R1	R1	R2

Each of the columns in this matrix defines a test rule for the function date.

So, if we tested the function with a representative data combination for each column, we'd have a complete test for this dependency. However we can improve this scenario with this assumption: Rules that lead to the same result, that are only different in one data field, can be combined. The differentiation into two subsets is not necessary.

STEP 6a: Reducing the number of combinations

If we look at the different combinations, we find that some of them lead to the same results. Here's an opportunity to reduce their number by combining them. When we do this, two rules have to be obeyed to avoid a degradation in the test coverage.

- Unite only combinations that have the same expected result.
- Unite only two combinations that are different in only one data field.

To see how we apply these rules in our example:

- Look at combinations 2 and 3 in the table below: in both cases the result is R2.
- Look at the data fields only for DD: two different subsets are selected, 30 and 31.
- Look at MM: in both cases the result is 2.
- Look for the leap year in the YYYY field: if we read it a little differently, we notice that whether month 02 in a leap year has 30 or 31 days, it is always an invalid date.

So the combinations 2 and 3, shown in gray, could be reduced to one. To show this in the table, we introduce a new symbol (✘) to show that this combination is covered already.

The table for the reduction of combination 2 and 3 would then look as follows:

		Combinations																	
Data Field Name	Data Entry	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
DD	29	✓			✓			✓			✓			✓			✓		
	30		✓			✓			✓			✓			✓			✓		✓
	31			✘			✓			✓			✓			✓			✓	
MM	02	✓	✓	✓							✓	✓	✓	✓	✓	✓	✓	✓	
	01, 03, 05, 07, 08. 10, 12				✓	✓	✓							✓	✓	✓			
	04, 06, 09, 11							✓	✓	✓							✓	✓	✓
YYYY	Leap year	✓	✓	✓	✓	✓	✓	✓	✓	✓									
	Non-leap year										✓	✓	✓	✓	✓	✓	✓	✓	✓
Expected result		R1	R2	R2	R1	R1	R1	R1	R1	R2	R2	R2	R2	R1	R1	R1	R1	R1	R2

To find the combinations that can be made part of another combination, look for those with equal expected results. If two combinations are only different in one data field, they can be merged.

STEP 6b: Further reducing the number of combinations

We'll now go through all combinations and look for possible reductions:

		Combinations																
Data Field Name	Data Entry	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
DD	29	✓			✓			✓			✓			✓			✓		
	30		✓			x			x			x			x			x	
	31			x			x			✓			x			x			✓
MM	02	✓	✓	✓							✓	✓	✓						
	01, 03, 05, 07, 08. 10, 12				✓	✓	✓							✓	✓	✓	✓	✓	
	04, 06, 09, 11							✓	✓	✓							✓	✓	✓
YYYY	Leap year	✓	✓	✓	✓	✓	✓	✓	✓	✓									
	Non-leap year										✓	✓	✓	✓	✓	✓	✓	✓	✓
Expected result		R1	R2	R2	R1	R1	R1	R1	R1	R2	R2	R2	R2	R1	R1	R1	R1	R1	R2

- Combinations 4, 5, and 6 can be merged because every month in the second subset of MM has 31 days.
- Combinations 7 and 8 can be merged because every month in the third group of MM has 30 days.
- Combinations 10, 11, and 12 can be merged because in a non-leap year, 02 never has 29, 30, or 31 days.
- Combinations 13, 14, and 15 can be merged because every month in the second subset of MM has 31 days.
- Combinations 16 and 17 can be merged because every month in the third subset of MM has 30 days.

STEP 6c: Reducing the number of combinations still further

The result of step 6b looks as follows:

		Combinations																	
Data Field Name	Data Entry	1	2	4	7	9	10	13	16	18									
DD	29	✓		✓	✓		✓	✓	✓										
	30	✓	✓																
	31					✓		✓		✓									
MM	02	✓	✓				✓												
	01, 03, 05, 07, 08. 10, 12			✓				✓	✓										
	04, 06, 09, 11				✓	✓				✓									
YYYY	Leap year	✓	✓	✓	✓	✓													
	Non-leap year						✓	✓	✓	✓									
Expected result		R1	R2	R1	R1	R2	R2	R1	R1	R2									

The rule to merge combinations can be applied again.

		Combinations								
Data Field Name	Data Entry	1	2	4	7	9	10	13	16	18
DD	29	✓		✓	✓		✓	✓	✓	
	30		✓							
	31					✓				✓
MM	02	✓	✓				✓			
	01, 03, 05, 07, 08. 10, 12			✓				✓	✓	✓
	04, 06, 09, 11				✓	✓				
YYYY	Leap year	✓	✓	✓	✓	✓				
	Non-leap year						✓	x	x	x
Expected result		R1	R2	R1	R1	R2	R2	R1	R1	R2

- Combinations 4 and 13 can be merged because every month in the second subset of MM has 31 days, which is true in leap and non-leap years.
- Combinations 7 and 16 can be merged because the third subset of MM has 30 days, which is true for leap and non-leap years.
- Combinations 9 and 18 can be merged because every month in the third subset of MM has only 30 days, so entering "31" would lead to the result invalid date, which is true in leap and non-leap years.

So finally, we have to look only for six different combinations, which represent all the business rules behind this dependency.

STEP 7: Putting it all together

Now we synthesize the results of the different steps, and that gives us the test rules that have to be executed to test the application date.

These test rules represent the complete functionality of the business function, so we have 100 percent functional coverage.

To define the test rules, we did not need the application, we just needed:

- The data that goes into the application
- The business knowledge about the functionality

		Test Rules								
Data Field Name	Data Entry	1	2	3	4	5	6	7	8	9
DD	00, 32–99	✓								
	00–28		✓	✓						
	29				✓		✓	✓		✓
	30					✓	✓	✓		
	31					✓	✓		✓	✓
MM	00, 13–99			✓						
	02	✓	✓		✓	✓				✓
	01, 03, 05, 07, 08. 10, 12	✓	✓				✓			
	04, 06, 09, 11							✓	✓	
YYYY	Leap year	✓	✓	✓	✓	✓	✓	✓	✓	
	Non-leap year	✓	✓	✓						✓
Expected result		R1	R2	R2	R2	R2	R1	R1	R2	R2

The next step will transform these test rules into test cases — and we'll do that in the next chapter.

Chapter 4

Test Cases: Let's Get Down to the Real Stuff

In the last chapter, we learned how to systematically establish the test rules for a business function. Now we can derive the test case from those test rules, which describe the results we want to see in the test.

For instance, if we were to test the software for an ATM machine before it shipped to the customer, one test rule might be: When a customer puts a credit card into the card slot, the customer is prompted to enter a PIN number.

For test execution we need to define:

- Account number
- Name of the bank that holds the card
- Type of credit card
- Name of the account owner

And we need to ensure that the data on the card matches the data in the system. The additional information will transform the *test rule* into a *test case*.