RBCS
TIME TESTED.
TESTING IMPROVED.
www.RBCS-US.com

Equivalence Partitioning

Rex Black, President, RBCS, Inc.

To celebrate completion of the update to Advanced Software Testing: Volume I, here is an excerpt of Chapter 3. This is the central chapter of the book, addressing test design techniques.

We start with the most basic of specification-based test design techniques, equivalence partitioning.

Conceptually, equivalence partitioning is about testing various groups that are expected to be handled the same way by the system and exhibit similar behavior. Those groups can be inputs, outputs, internal values, calculations, or time values, and should include valid and invalid groups. We select a single value from each equivalence partition, and this allows us to reduce the number of tests. We can calculate coverage by dividing the number of equivalence partitions tested by the number identified, though generally the goal is to achieve 100% coverage by selecting at least one value from each partition.

This technique is universally applicable at any test level, in any situation where we can identify the equivalence partitions. Ideally, those partitions are independent, though some amount of interaction between input values does not preclude the use of the technique. This technique is also very useful in constructing smoke tests, though testing of some of the less-risky partitions frequently is omitted in smoke tests. This technique will find primarily functional defects where data is processed improperly in one or more partitions. The key to this technique is to take care that the values in each equivalence partition are indeed handled the same way; otherwise, you will miss potentially important test values.

ISTQB Glossary

Equivalence partitioning: A black box test design technique in which test cases are designed to execute representatives from equivalence partitions. In principle test cases are designed to cover each partition at least once.

In equivalence partitioning, the underlying model is a graphical or mathematical one that identifies equivalent classes--which are also called equivalent partitions-

-of inputs, outputs, internal values, time relationships, calculations, or just about anything else of interest. These classes or partitions are called equivalent because they should be handled the same way by the system. Some of the classes can be called valid equivalence classes because they describe valid situations that the system should handle normally. Other classes can be called invalid equivalence classes because they describe invalid situations that the system should reject or at least escalate to the user for correction or exception handling.

Once we've identified the equivalent classes, we can derive tests from them. Usually, we are working with more than one set of equivalence classes at one time; for example, each input field on a screen has its own set of valid and invalid equivalence classes. So, we can create one set of valid tests by selecting one valid member from each equivalence partition. We continue this process until each valid class for each equivalence partition is represented in at least one valid test.

Next, we can create invalid tests. For each equivalence partition, we select one invalid member for one equivalence partition, and a valid member for every other equivalence partition. This rule—don't combine multiple invalid members in a single test—prevents us from running into a situation where the presence of one invalid value might mask the incorrect handling of another invalid value. We continue this process until each invalid class for each equivalence partition is represented in at least one invalid test.

Notice the coverage criteria implicit in the discussion above. Every class, both valid and invalid, is represented in at least one test.

What is our bug hypothesis with this technique? For the most part, we are looking for a situation where some equivalence class is handled improperly. That could mean the value is accepted when it should have been rejected or vice versa, or that a value is properly accepted or rejected, but handled in a way appropriate to another equivalence class, not the class to which it actually belongs.

Visualizing Equivalence Partitioning

How can we visualize equivalence partitioning? Let's look.

As you can see in the top half of Figure 3-2, we start with some set of interest. This set of interest can be an input field, an output field, a test pre-condition or post-condition, a configuration, or just about anything we're interested in testing. The key is that we can apply the operation of equivalence partitioning to split the set into two or more disjoint subsets, where all the members of each subset share some trait in common that is not shared with the members of the other subset.

For example, if you have a simple drawing program that can fill figures in with red, green, or blue, you can split the set of fill colors into three disjoint sets: red, green, and blue.

In the bottom half of Figure 3-2, we see the selection of test values from the subsets. The dots in the subsets represent the value chosen from each subset to be represented in the test. This involves selecting at least one member from each subset. In pure equivalence partitioning, the logic behind this selection is outside the scope of the technique. In other words, you can select any member of the subset you please. If you're thinking, "Some members are better than others," that's fine, hold that thought for a few minutes and we'll come back to it.

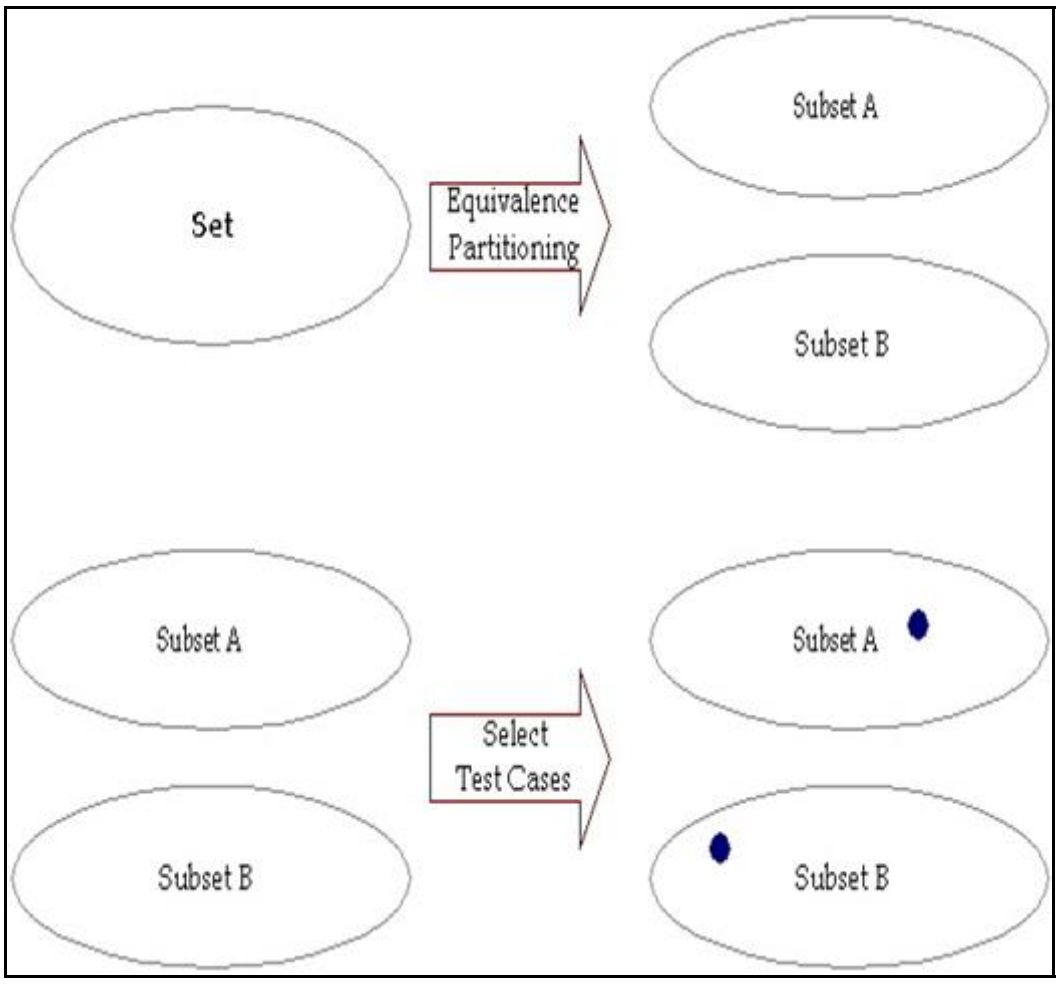

Figure 3-1 Visualizing equivalence partitioning

Now, at this point we'll generate the rest of the test. If the set that we partitioned was an input field, we might refer to the requirements specification to understand how each subset is supposed to be handled. If the set that we

partitioned was an output field, we might refer to the requirements to derive inputs that should cause that output to occur. We might use other test techniques to design the rest of the tests.

Figure 3-3 shows that equivalence partitioning can be applied iteratively. In this figure, we apply a second round of equivalence partitioning to one of the subsets to generate three smaller subsets. Only at that point do we select four members—one from subset B and one each from subsets A1, A2, and A3—for tests. Note that we don't have to select a member from subset A, since each of the members from subsets A1, A2, and A3 are also members of subset A.

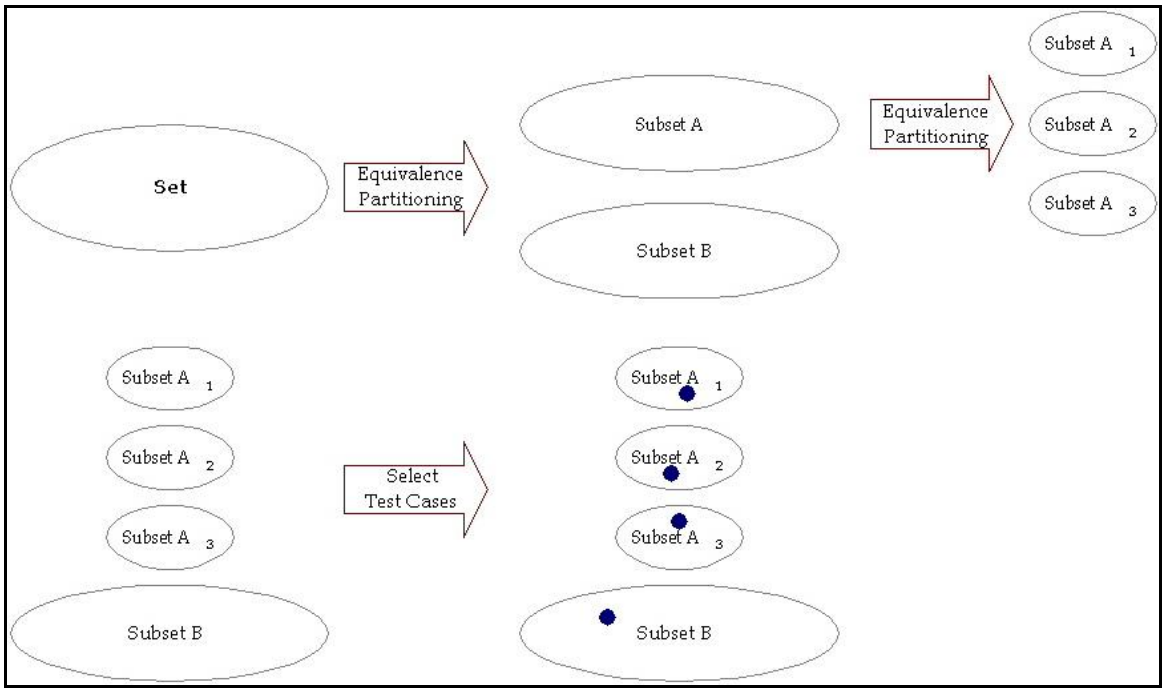

Figure 3-2 Equivalence partitioning applied iteratively

Avoiding Equivalence Partitioning Errors

Let me emphasize three key points

One, as shown in the top half of Figure 3-4, the subsets must be **disjoint**. That is, no two of the subsets can have one or more members in common. The whole point of equivalence partitioning is to test whether a system handles different situations differently (and properly, of course). If it's ambiguous as to which handling is proper, then how do we define a test around this? "Try it out and see what happens?" Not much of a test!

Two, as shown in the bottom half of Figure 3-4, none of the subsets may be **empty**. That is, if the equivalence partitioning operation produces a subset with

no members, that's hardly very useful for testing. We can't select a member of that subset, because it has no members.

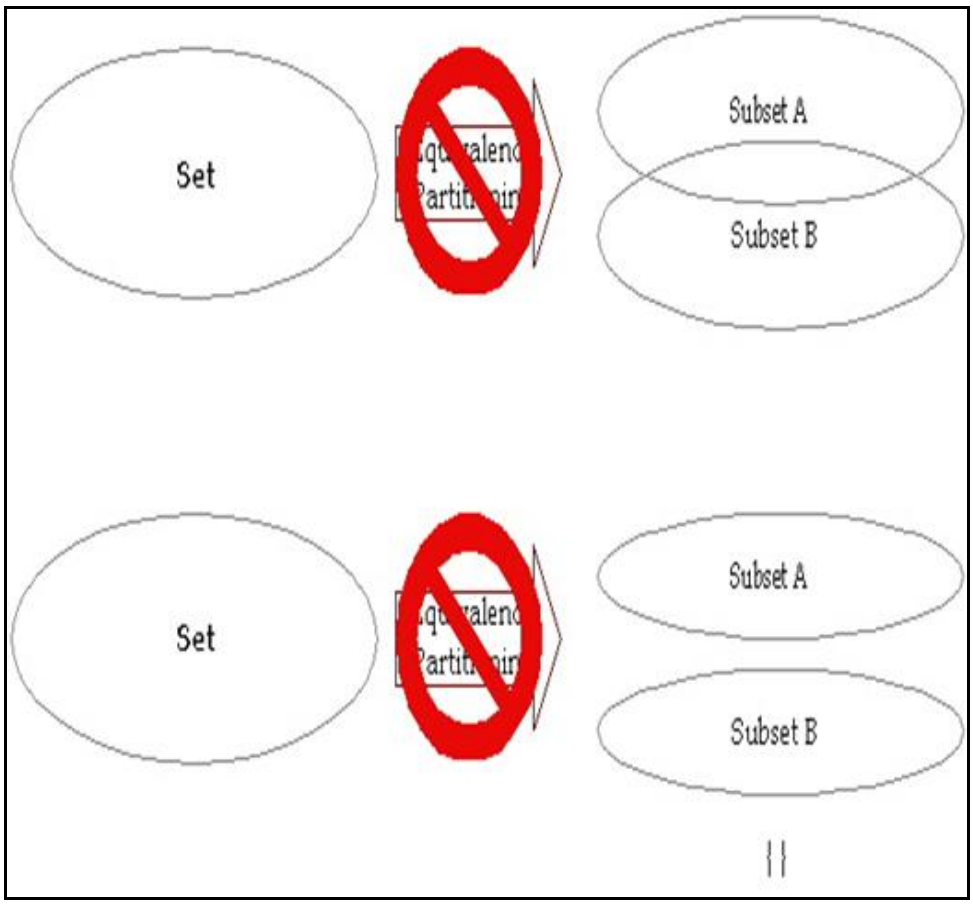

Figure 3-3 Avoiding equivalence partitioning errors

Third, while not shown graphically—in part because I couldn't figure out a clear way to draw the picture—note that the equivalence partitioning process does not subtract, it divides. What I mean by this is that, in terms of mathematical set theory, the union of the subsets produced by equivalence partitioning must be the same as the original set that was equivalence partitioned. In other words, equivalence partitioning does not generate "spare" subsets that are somehow disposed of in the process—at least, not if we do it properly. Notice that this is important, because, if this is not true, then we stand the chance of failing to test some important subset of inputs, outputs, configurations, or some other factor of interest that somehow was dropped in the test design process.

Composing Tests

When we compose test cases in situations where we've performed equivalence partitioning on more than one set, we select from each subset as shown in Figure 3-5.

Here, we start with set X and set Y. We partition set X into two subsets, X1 and X2. We partition set Y into three subsets, Y1, Y2, and Y3. We select test values from each of the five subsets, X1, X2, Y1, Y2, and Y3. We then compose three tests, since we can combine the values from the X subsets with values from the Y subsets (assuming the values are independent and all valid).

For example, imagine you are testing a browser-based application. You are interested in two browsers, Chrome and Firefox. You are interested in three connection types, WiFi, DSL, and cable modem. Since the browser and the connection types are independent, we can create three tests. In each of the tests, one of the connection types and one of the browser types will be represented. One of the browser types will be represented twice.

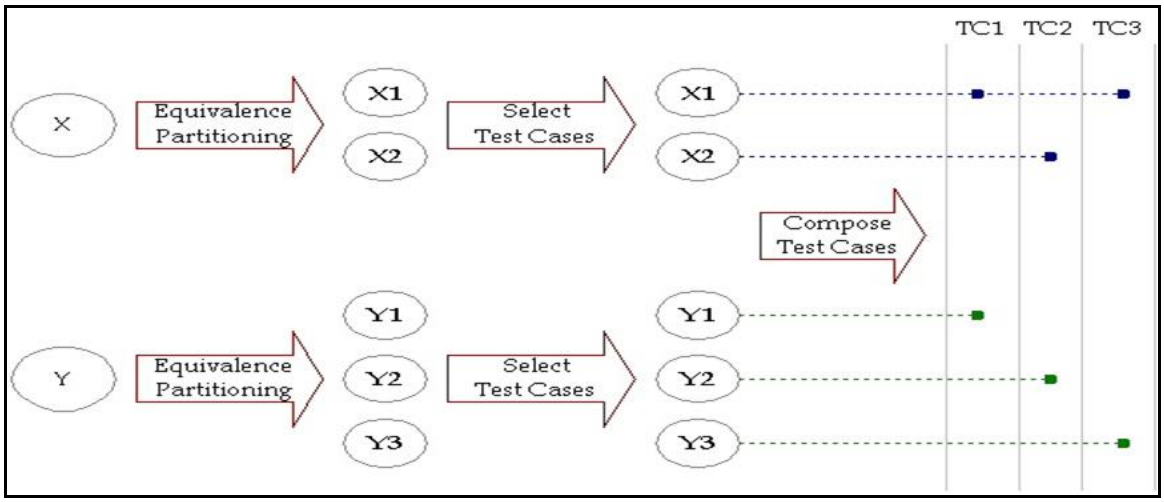

Figure 3-4 Composing tests

Earlier, I made a brief comment that we can combine values across the equivalence partitions when the values are independent and all valid. Of course, that's not always the case as shown in Figure 3-6.

In some cases, values are not independent, in that the selection of one value from one subset constrains the choice in another subset. For example, imagine if you're trying to test combinations of applications and operating systems. You can't test an application running on an operating system if there is not version of that application available for that operating system.

In some cases, values are not valid. For example, in Figure 3-6, imagine that we are testing a project management application, something like Microsoft Project.

Suppose that set X is the type of event we're dealing with, which can be either a task (X1) or a milestone (X2). Suppose that set Y is the start date of the event, which can be in the past (Y1), today (Y2), or in the future (Y3). Suppose that set Z is the end date of the event, which can be either on or after the start date (Z1) or before the start date (Z2). Of course, Z2 is invalid, since no event can have a negative duration.

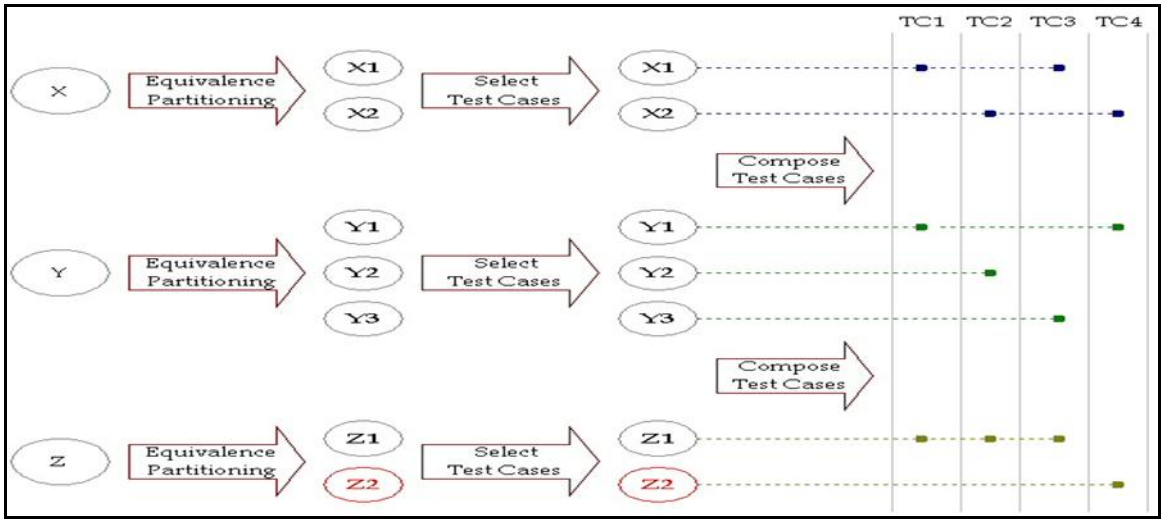

Figure 3-5 Composing tests with invalid values

So, we test combinations of tasks and milestones with past, present, and future start dates and valid end dates in test cases TC1, TC2, and TC3. In T4, we check that illegal end dates are correctly rejected. We try to enter a task with a start date in the past and an end date prior to the start date. If we wanted to subpartition the invalid situation, we could also test with a start date in the present and one in the future, together with an end date before the start date, which would add two more tests.

In this particular example, we had a single subset for just one of the sets that was invalid. The more general case is that many—perhaps all—of the sets will have invalid subsets. Imagine testing an e-commerce application. On the checkout screens of the typical e-commerce application, there are multiple required fields, and usually there is at least one way for such a field to be invalid.

When there are multiple invalid values, we have to select one invalid value per test—at least to start with. That way, we can check that any given invalid value is correctly rejected or in some way handled by the system.

Now, I said one invalid value per test to start with. If, after doing that, you want to test combinations of invalids, by all means, do so—if the risk is sufficient to justify it. Any time you start down the trail of combinatorial testing, you are

taking a chance that you'll spend a lot of time testing things that aren't terribly important.

ISTQB Glossary

Combinatorial testing: A means to identify a suitable subset of test combinations to achieve a predetermined level of coverage when testing an object with multiple parameters and where those parameters themselves each have several values, which gives rise to more combinations than are feasible to test in the time allowed.

Example: Equivalence Partitioning

Here's a simple example of equivalence partitioning on a single value, in this case system configuration. On the Internet appliance project I've mentioned before, there were four possible configurations for the appliances. They could be configured for kids, teens, adults, or seniors. This configuration value was stored in a database on the Internet Service Provider server, so that, when an Internet appliance connected to the Internet, this configuration value became a property of its connection.

Based on this configuration value, there were two key areas of difference in the expected behavior. For one thing, for kids and teens systems, there was a filtering function enabled. This determined the allowed and disallowed Web sites the system could visit. The setting was most strict for kids, and somewhat less strict for teens. Adults and seniors, of course, were to have no filtering at all, and should be able to surf anywhere.

For another thing, each of the four configurations had a default set of e-commerce sites they could visit called the mall. These sites were selected by the marketing team, and were meant to be age appropriate.

Of course, these were the expected differences. We were also aware of the fact that there could be weird unexpected differences that could arise, because that's the nature of some types of bugs. For example, performance was supposed to be the same, but it's possible that performance problems with the filtering software could introduce perceptible response-time issues with the kids and teens systems. We had to watch for those kinds of misbehaviors.

So, to test for the unexpected differences, we simply had at least one of each configuration in the test lab at all times, and spread the non-configuration-specific tests more or less randomly across the different configurations. (More on testing combinations of configurations in a subsequent part of this section.) To test expected differences related to filtering and e-commerce, we made sure these configuration-specific tests were run on the correct configuration. The challenge

here was that, while the expected results were concrete for the mall—the marketing people gave us four sets of specific sites, one for each configuration—the expected results for the filtering were not concrete but rather logical. This lead to an enormous number of very disturbing defect reports during test execution, as we found creative ways to sneak around the filters and access age-inappropriate materials on kids and teens configurations.