Jiwaji University
SOS in Computer Science and Applications
B.C.A. (VI Semester)
Paper-604 Software Testing
Unit 2

Topic:
Random Testing
Boundary Value Analysis
Equivalence Class Partitioning

By Arun Singh

Random Testing

Each software module or system has an input domain from which test input data is selected. If a tester randomly selects inputs from the domain, this is called random testing. For example, if the valid input domain for a module is all positive integers between 1 and 100, the tester using this approach would randomly, or unsystematically, select values from within that domain; for example, the values 55, 24, 3 might be chosen. Given this approach, some of the issues that remain open are the following:

1. Are the three values adequate to show that the module meets its specification when the tests are run? Should additional or fewer values be used to make the most effective use of resources?
2. Are there any input values, other than those selected, more likely to reveal defects? For example, should positive integers at the beginning or end of the domain be specifically selected as inputs?
3. Should any values outside the valid domain be used as test inputs? For example, should test data include floating point values, negative values, or integer values greater than 100?

Use of random test inputs may save some of the time and effort that more thoughtful test input selection methods require. However, the reader should keep in mind that according to many testing experts, selecting test inputs randomly has very little chance of producing an effective set of test data.

There has been much discussion in the testing world about whether such a statement is accurate. The relative effectiveness of random versus a more structured approach to generating test inputs has been the subject of many research papers.

Boundary Value Analysis

Equivalence class partitioning gives the tester a useful tool with which to develop black box based-test cases for the software-under-test. The method requires that a tester has access to a specification of input/output behavior for the target software. The test cases developed based on equivalence class partitioning can be strengthened by use of an technique called boundary value analysis. With experience, testers soon realize that many defects occur directly on, and above and below, the edges of equivalence classes.
The rules-of-thumb described below are useful for getting started with boundary value analysis.

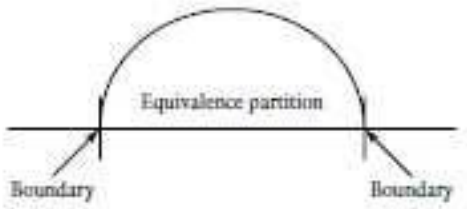

FIG. 4.3
Boundaries of an equivalence partition.

1. If an input condition for the software-under-test is specified as a *range* of values, develop valid test cases for the ends of the range, and invalid test cases for possibilities just above and below the ends of the range. For example if a specification states that an input value for a module must lie in the range between _1.0 and _1.0, valid tests that include values for ends of the range, as well as invalid test cases for values just above and below the ends, should be included. This would result in input values of _1.0, _1.1, and 1.0, 1.1.

2. If an input condition for the software-under-test is specified as a *number* of values, develop valid test cases for the minimum and maximum numbers as well as invalid test cases that include one lesser and one greater than the maximum and minimum. For example, for the real-estate module mentioned previously that specified a house can have one to four owners, tests that include

3. If the input or output of the software-under-test is an ordered set, such as a table or a linear list, develop tests that focus on the first and last elements of the set. It is important for the tester to keep in mind that equivalence class partitioning and boundary value analysis apply to testing both inputs and outputs of the software-under-test, and, most importantly, conditions are *not* combined for equivalence class partitioning or boundary value analysis. Each condition is considered separately, and test cases are developed to insure coverage of all the individual conditions.

Equivalence Class Partitioning

If a tester is viewing the software-under-test as a black box with well- defined inputs and outputs, a good approach to selecting test inputs is to use a method called equivalence class partitioning. Equivalence class partitioning results in a partitioning of the input domain of the software under test. The technique can also be used to partition the output domain, but this is not a common usage. The finite number of partitions or equivalence classes that result allow the tester to select a given member of an equivalence class as a representative of that class. It is assumed that all members of an equivalence class are processed in an equivalent way by the target software.

Based on this discussion of equivalence class partitioning we can say that the partitioning of the input domain for the software-under-test using this technique has the following advantages:

1. It eliminates the need for exhaustive testing, which is not feasible.

2. It guides a tester in selecting a subset of test inputs with a high probability of detecting a defect.

3. It allows a tester to cover a larger domain of inputs/outputs with a smaller subset selected from an equivalence class.

There are several important points related to equivalence class partitioning that should be made to complete this discussion.
1. The tester must consider both valid and invalid equivalence classes. Invalid classes represent erroneous or unexpected inputs.

2. Equivalence classes may also be selected for output conditions.

3. The derivation of input or outputs equivalence classes is a heuristic process. The conditions that are described in the following paragraphs only give the tester guidelines for identifying the partitions. There are no hard and fast rules. Given the same set of conditions, individual testers may make different choices of equivalence classes. As a tester gains experience he is more able to select equivalence classes with confidence.

4. In some cases it is difficult for the tester to identify equivalence classes. The conditions/boundaries that help to define classes may be absent, or obscure, or there may seem to be a very large or very small number of equivalence classes for the problem domain. These difficulties may arise from an ambiguous, contradictory, incorrect, or incomplete specification and/or requirements description. It is the duty of the tester to seek out the analysts and meet with them to clarify these documents. Additional contact with the user/client group may be required. A tester should also realize that for some software problem domains defining equivalence classes is inherently difficult, for example, software that needs to utilize the tax code.

Test cases, when developed, may cover multiple conditions and multiple variables.

List o f Conditions

1. If an input condition for the software-under-test is specified as a range of values, select one valid equivalence class that covers the allowed range and two invalid equivalence classes, one
outside each end of the range.“

2. If an input condition for the software-under-test is specified as a number of values, then select one valid equivalence class that includes the allowed number of values and two invalid equivalence classes that are outside each end of the allowed number.“

3. If an input condition for the software-under-test is specified as a set of valid input values, then select one valid equivalence class that contains all the members of the set and one invalid equivalence class for any value outside the set.“

4. If an input condition for the software-under-test is specified as a ―must be‖ condition, select one valid equivalence class to represent the ―must be‖ condition and one invalid class that does not include the ―must be‖ condition.“

5. If the input specification or any other information leads to the belief that an element in an equivalence class is not handled in an identical way by the software-under-test, then the class should be further partitioned into smaller equivalence classes.“

After the equivalence classes have been identified in this way, the next step in test case design is the development of the actual test cases. A good approach includes the following steps.

1. Each equivalence class should be assigned a unique identifier. A simple integer is sufficient.

2. Develop test cases for all valid equivalence classes until all have been covered by (included in) a test case. A given test case may cover more than one equivalence class.

3. Develop test cases for all invalid equivalence classes until all have been covered individually. This is to insure that one invalid case does not mask the effect of another or prevent the execution of another.