Are test design techniques useful or not?

By Hans Schaefer
Software Test Consulting, 5281 Valestrandsfossen, Norway
www.softwaretesting.no

Many people test, but few people use the well-known black-box and white-box test design techniques. The technique most used, however, seems to be testing randomly chosen valid values, followed by error guessing, exploratory testing and the like. Could it be that the more systematic test design techniques are not worth using?

I do not think so. When safety-critical software is produced, it is tested very systematically using the good old well-known techniques. Standards recommend or require doing so. Therefore there must be some value. What kind of value?

The most important one is representativeness: When using a systematic technique, we test representative and special values. We test for two reasons: To find as many problems as possible, and to assess the quality of the product, i.e. we test with destructive values and with representative values.

The other much-needed quality of testing is accountability: We need to be able to tell our clients what we have tested, what not and why. Or what we tested more or less.

When the software works or fails for one value, we assume, very often rightly so, that it will do the same for other values. The basis for this is called **equivalence partitioning**. The benefit: We do not unnecessarily duplicate testing effort.

But how can we do it?

Let me introduce a few methods, to make it easier.

First method: Test one right and one wrong value

In many cases software works relatively well if your inputs are correct or ”mainstream”. If some input is ”special” or wrong, it fails. For consumer software, when confronted with failures, the user will often ask, ”How could anyone have forgotten to test this obvious thing?”

Examples:

- The input should be numeric: Test one typical correct value. For the incorrect one, choose a value which is too high (extremely high for example) or a negative number. Zero is a typical destructive value, often difficult to handle for a program. Trying a non-numeric input is also a valuable choice.
- The input is text: Try a usual text with usual length as correct. Try special characters, a missing text (no input) or a text which is too long as wrong possibilities.

- Logical inputs: Just try yes and no.
- A group of allowed values: Try the most common one as correct, and something else that is wrong. For the wrong one you could choose a value that is ”near correct” or that was correct in an earlier version or is correct in other software.
- In any case: Try to leave out inputs, and try repetitions.

The value of this technique: You will at least get some impression of how error handling works with your software. Error handling is often badly designed and implemented, and also badly tested by programmers. Thus the wrong values have good value when it comes to discovering trouble.

As you see, this technique leaves many choices, e.g. which of the discrete allowed values you should choose. Or if you should choose a numerical value that is too high or too low. Thus we have another method:

Second method: Equivalence partitioning

Somehow, this is ”the mother of all testing”. The idea is to partition any set of possible values into sets that you think are equivalent. The idea of equivalence means that you think the program will handle equivalent values in, principally, the same way. YOU think! This means someone else may find a different partition. It depends a bit on viewpoint and experience. Then you select one value per equivalence class for testing. IN PRINCIPLE, handling the input in the same way means that you can assume the program is executing the same lines of code. For example, for a bank transfer, it should not matter if the amount is 100, 140, 145, 150, or something like that. Or for checking validity of a date input, at least every day between 2 and 27 should work the same way.

A different use of this technique is backwards: When I review test plans, I look at the values planned to be used in the test. Then I try to figure out what equivalence classes might be behind them, which other equivalence classes there may be, and then I search for the other test values covering these other equivalence classes (and often find ”holes in the test”).

Here are the main rules:

- If an input is numeric, choose a value that is too small (wrong); one that is too high (wrong) and one that is correct (right).
- If an input is logical, then test both true and false.
- If an input is a time value, do as with numeric values, but include a fourth class: impossible values (like 35th of December or 36 o’clock).
- For discrete inputs, test every allowed value plus one wrong.
- Test every input in two ways: given and not given.
- Try correct and wrong data type for inputs. (Example: texts in a number field, Chinese characters for an ASCII text field etc.)
- If an input can be repeated, try zero, one, more than one repetitions.
- **The pessimist rule**: If you think something might not be equivalent, then partition any such equivalence class into subclasses. (This creates a more

thorough test, but also more testing work). Real pessimists would test every value, but then they would never finish testing.

If you look at other systematic testing techniques, many of them are based on equivalence classes.

This technique is easy to learn and use, and it saves a lot of duplicated test effort. You test one value per equivalence class only. Combinations of equivalence classes may be tested as a more advanced technique. For most inputs, if the data type and the conditions are given, equivalence partitioning is easy: It is the same over and over again. In principle, just use the checklist above!

There are tools, which support the technique. CASEMAKER is one of them.

The trouble with this technique:

- Logical combinations are not very well covered.
- The technique is blind for trouble when the implementation of the program is done in a completely different way than you think it may be.
- The technique may be too complicated when experienced people use the pessimist rule a lot.
- Some values may find more problems or may be more representative than other values.

Therefore we may need more. This is described in the remainder of this paper. There are several techniques.

Third method: Logical combination testing

Logical combinations are dangerous. In many cases there is unclear thinking: Some combinations are forgotten. Distributed logic, implemented in different places, may have holes or side effects. The conclusion is: Logical combinations must be tested. The problem: Combinatorial explosion. The number of tests would grow by the power of two per new combination. This may be OK for small problems, but if the number of parameters exceeds about five, this tends to generate too much work.

As an example, here is the table of all combinations of three logical inputs:

Test case number	Input 1	Input 2	Input 3	
1	N	N	N	
2	N	N	Y	
3	N	Y	N	
4	N	Y	Y	
5	Y	N	N	
6	Y	N	Y	
7	Y	Y	N	
8	Y	Y	Y	

Every new variable will duplicate this table!

One way to cut down on this is testing only lower-level combinations: Observation has shown that pairwise combinations detect most errors. Less errors by combinations of three values, and so on. There are many tools supporting pairwise combinations, where all pairs of values from different parameters are generated automatically. The technique is popular, as it saves a tremendous amount of work.

As an example, here is the table of all pairwise combinations of three logical inputs:

Test case number	Input 1	Input 2	Input 3	
1	Y	Y	Y	
2	Y	N	N	
3	N	N	Y	
4	N	Y	N	

However, pairwise combination testing may be dangerous, as it may overlook errors. This is especially true if there are implicit dependencies. An example of such dependencies is the validity checking of a date input: The number of days in a month is dependent on the month and even the year. However, if there are no known dependencies, then pairwise testing is a good first choice. It may even be applied if some values are not logical, but discrete.

Example for general combination testing with discrete values:
One input is the month, another one is a color.
The month falls into three categories: 31 days, 30 days, and February.
The colors may be red, blue, black, and white.
All combinations of these two inputs are 12.

A more difficult technique is cause-effect graphing. This technique is only useful if supported by a tool. Otherwise it is too difficult to apply. The analysis may introduce more problems than are found.

There are more techniques available for logical combination testing, but these would be too complicated for a short article.

Fourth method: State-transition testing

Many programs are defined as state-transition diagrams, state charts, or state machines. The technique is popular and supported by UML. A corresponding testing technique exists. It was defined back in the 1960s and 1970s.

Examples for state machines are:

- A mobile phone. It may be on, off, may be a phone, a camera, a database, and SMS machine etc.
- A traffic light: The state is often visible as the displayed light.
- Most home appliances. Buttons introduce state transitions.

- The status accounting in a defect reporting and tracking system.

These are the different coverage criteria:

1. The test reaches every (reachable) state
2. The test passes through every (reachable) transition
3. The test should cover every possible event in every state
4. The test should cover combinations of transitions after each other.

All these test design methods are easy to apply if the state machine is defined in a formal, machine-readable way, and if a test design tool is applied. Actually, most of the model-based testing is about testing state machines.

State machine testing tests only the state machine logic in itself. If data can vary, they must be tested using other techniques, like equivalence partitioning (boundary value analysis (see later) or logical combination testing).

The first method is not very valuable. The main value is in finding states that are not reachable, even if they should be, or requirement problems (misunderstandings, unclear requirements etc.). From the start state or from initialization, the state machine is fed with inputs in order to get it to assume every state. If some state seems unreachable, the problem is researched.

The second technique has a value: It checks if the state machine "works", i.e. if every defined transition gives a reasonable result. An example is the answer to the question what should happen if someone presses this or that button. The test design starts again from the start state, and now in every state, every defined transition is tested. Typically, these are the "valid" transitions. An example would be to switch on your mobile phone, and then test everything there is in the user manual that has to do with state (e.g. the menu selection).

The third technique will find that not every possible event has a defined reaction for every state. The technique described above, testing one right and one wrong, is an example of this: The "wrong" events are often forgotten. However, if there is a "water-tight" specification, testing in every state has probably been done already. To continue the mobile phone example, this technique would require trying every button in every state of the phone (considering only buttons as inputs here).

The last technique is difficult and possibly leads to indefinite amounts of work: One may test two, three, n events after each other, starting from each and every state. This kind of testing finds "long-term corruption", i.e. the state machine somehow depends on several events after each other, and on the order of these events. Even hidden extra states implemented may be found. But the amount of work for this is probably too much. This technique, if applied at all, requires tools.

However, a "trick" may be applied: When testing state machines, one test should be chained after the other, with no restart in between. This will at least cover some of the combinations and thus be more destructive.

In the mobile phone example, we may try to press buttons one after each other, in all possible different orders. (People who tried this sometimes had to take back their

phones for service). With these mobile phone examples, I ignore that the phone has some more states, depending on incoming calls etc. Thus, in reality, the test is even more complicated than outlined here.

Fifth method: Boundary value and domain analysis

Some values are better than others in finding trouble. Boundary values have this property. This is because people have problems expressing boundaries correctly, and because programmers tend to use the wrong comparison operator (less or equal instead of less, for example). Another problem is counting: Boundary value analysis is good against off-by-one errors.

Boundary value analysis can very easily be combined with equivalence partitioning. When choosing values in an equivalence class, the boundaries are prioritized.

The technique asks us to test two or three values near the boundary: The boundary itself and one or two values as near as possible either side of the boundary. I used to test all three of them: Below and above the boundary as well as the boundary value itself. This requires less thinking and is safer. I call this ”the Friday afternoon method”. If you have less time and experienced people, you may just test two values: The boundary and another one nearby which shall be in the other equivalence class. This requires more thinking. I call this the ”Monday through Thursday method”.

When it comes to boundaries in several dimensions, like the border of a window, you test each linear boundary with three values: Two (on n, for n-dimensional problems) **on** the boundary, one inside or outside, in the other equivalence class. Domain testing is interesting when boundaries are not geometrically vertical or horizontal. Then you save a lot of combinatorial testing. However, for three or four dimensions, the technique soon becomes difficult.

Boundary value analysis should not only be applied to user inputs. There are many other (more implicit) inputs, and they should be tested. James Bach, a testing guru, defines a few more rules. Boundary value analysis should also be used to

- question the validity and interpretation of requirements,
- discover (implicit) boundaries the programmers overlooked,
- learn about boundaries that emerge from the interactions among sub-systems,
- discover the absence of boundaries where such absence creates the opportunity for performance and reliability problems.

Here is a list of (implicit) boundaries to check up:
Buffer sizes
Table sizes
First and>> last elements in buffers, tables, files etc.
Memory boundaries
Allowed number of repetitions
Message sizes
Lengths of output fields

File lengths
List lengths
Transition in time (over the hour, day, year)
Timeouts

This kind of testing requires programming experience. You need to know what is going on behind the scene.

An example boundary problem in PowerPoint for Mac, WORD ART:

- Fill in characters until the maximum (just hold down some key!).
- When the maximum (about 128) has been reached, no more characters can be input, but no error message is displayed. Keyboard input is just ignored.

What is missing?

There are many more techniques. Some of the most useful or often discussed ones are these:

- Random and statistical testing
- Exhaustive testing
- Error guessing
- White-box testing
- The test oracle

Random and statistical testing

Software is digital. It may fail for any value. Equivalence class partitioning is a black-box technique. It tends to overlook things implemented in the program unknown from the outside. For example “Easter eggs”, special conditions triggering very special behavior. This means, in principle, that any value could be tested and has some chance of discovering trouble. On the other hand, a randomly selected test normally has a low defect-finding ability. It only works if there are very many tests. Statistical testing tries to improve random testing by concentrating on values, which will occur more often in practice. However, in order to apply it, the tester must design a usage profile for the application under test. This may be a huge endeavor or impossible. Anyway: If anything can easily be tested by generating random inputs, and checking the output can also be automated, then this method is promising.

Exhaustive testing

This is the extreme case: Every possible combination of input values is tested. In principle, this should find all problems. In practice apllying this method is not possible. There are plainly too many possibilities.

Error guessing or fault attack

This method concentrates on typical problems with the program. There is an assumption that the tester has some idea about what typical problems are. Examples include:

- Boundary values (covered by boundary value analysis)
- National and special characters for text values
- Missing and duplicated input
- Too much input
- Low memory
- Unavailable resources
- And more.

A tester should always keep a log about typical problems. Checking the existing defect log is a good idea.

Error guessing is the oldest test method. Over time, much of it has been systematized and has flown into the other methods described here. But a tester should always put in "the little extra" for concentrating on typical problems.

White-box testing

This method is typically used for unit or component testing where there is access to the program code. The tester tries to cover all code. There are many different coverage criteria, but an absolute minimum should be to cover most of the branches in the code. This will assure that most of the statements or lines of code are also executed. The question is: How much do you know about a line of code that is never executed?

The test oracle

Testing requires you to interpret the system's reaction to the inputs. You compare the output with the expected output. But who tells you the expected output?

The test oracle!

It is defined as a mechanism that decides if a result is right or wrong. In principle, you look into the specification. But often, results are complicated and interpretation is difficult. It would be nice to have an automatic oracle. This would be another program, but it could be a spreadsheet implementation for the mathematics, or an older version of the program. If this is not available, you may test with "easy values", where manual calculation is easy. Or you might deviate from full checking and just do a plausibility check. In the worst case, the only thing you do is check that the program does not crash.

Summary

Test design techniques are valuable. You have to design your test cases anyway, so why not do it systematically. The benefit is: If somebody asks how you did it, you are able to describe it, plus your reasoning behind it. Your test will be accountable. And you may be able to improve over time.

To randomly fire some shots at the program is not testing!