3 WAYS TO IMPROVE YOUR TESTING SKILLS

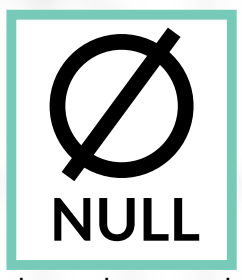
Did I write enough tests? Too many?
Did I cover the most common causes for defects?
If you ever have asked yourself any of those questions,
take a look at test case heuristics.

A heuristic (proper: "heuristic technique") is an approach to problem solving based on experience that will get you close to the ideal solution.

Using heuristics in unit testing helps you to cover the most common causes for defects. But they also help to keep the number of tests low, reducing maintenance costs.

In this document you find the three Test Case Heuristics that cover the most common issues. While there are more heuristics that I teach in my classes, these are the three that I use all the time during development.

Special Values

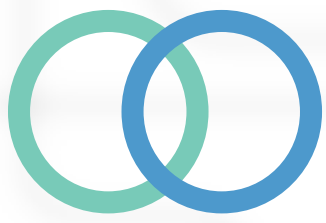


- These are values that the code interacts with or makes decisions based on (like parameters) that are uncommon or unexpected
- They are one of the most common causes for software defects
- Examples are:
  1. Negative numbers (e.g. for the number of rows)
  2. NULL for an ID, a date range or really anywhere else
  3. 0
  4. Very large values or far future (or past) dates

Every time you deal with user-supplied values (parameters, but also data in any table), you should:

1. Spend a minute to think about what uncommon values the code might possibly encounter
2. Decide what the appropriate behavior in such a case should be
3. Write a test for it.

0-1-Some



The "0-1-Some" heuristics helps testing the behavior of code that interacts with sets. In SQL Server that is:

- Code that accesses a table
- Joins
- Code that acts on database objects (e.g. code that rebuilds all indexes on a table)

Any time you are testing code that is dealing with sets, you want to write three tests:

- One test to prove that the code handles an empty set
- One test for a single element
- One for a few elements.

In most cases, the "some" case requires no more than 3 rows. In rare cases up to ten can be required.

Boundary Conditions

In development, a boundary is any input value that causes a change in the behaviour.

Take these requirements for example:
Write a function to calculate the order discount. Any order with a total between $50 and $100 receives 1% discount, orders beyond that receive 3%. Lower priced orders are not discounted. In these requirements, both $50 and $100 define a boundary.

There are two common issues arising from requirements like this:

- The requirement might be ambiguous (as in the example).
- The implementation involves (naturally) a comparison operator. While that seems trivial at first, a very common cause for defects is the accidental use of the wrong operator (like "<" instead or "<=").

To discover requirement ambiguities and prevent operator typos, write three test cases for each boundary:

- One testing the boundary value itself
- One just a tad left of it
- One just to the right of it.

Powered by sqlity.net