Create proper Test Data for Application Development

If the test data is not good/great, it will lead into **many problems** in application/systems development:

- You might think something is wrong (false positive), but in fact the problem is that the test data is not logical, or even has mistakes.
- You might also think some feature works (false negative), as test data is missing variation that would show that feature does not work in all situations.
- Or it might be hard to track a problem in our code or application as test data doesn't allow for revealing tests that would test from many angles.

0. While programming – First technical testing

Use **at least** 5 objects/items/rows of each.

Reason1: After that you have first item (1.), item after first item (2.), last item (5.), item before last item (4.) and one neutral item which is not at those places (3.).

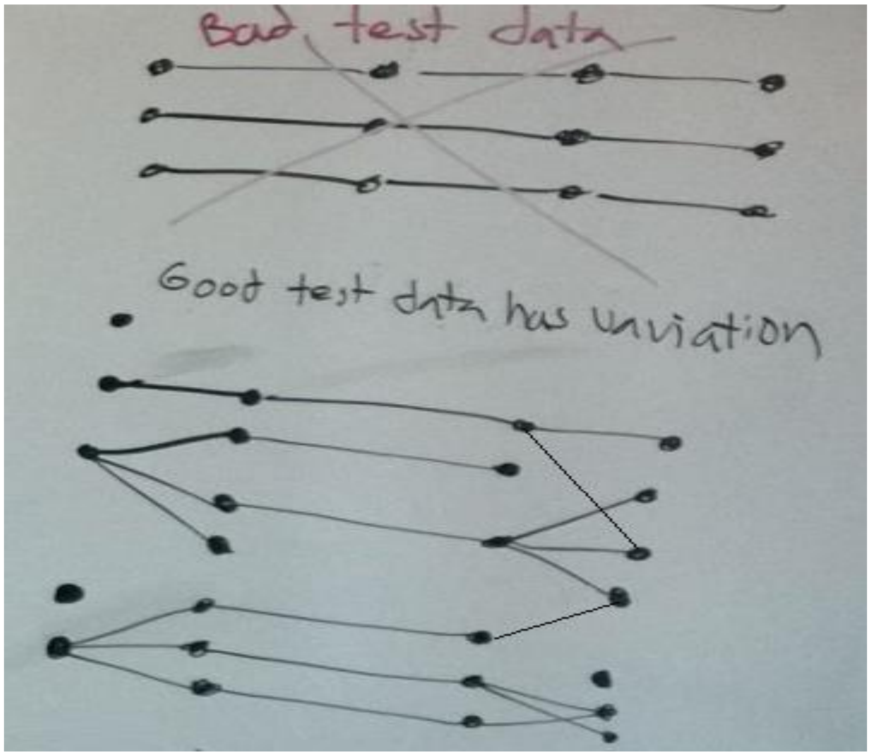
Reason2: If you code does not work with 5 items, it won't work with million. (But not the other way around)

1. Real smoke testing of features of a larger application proto

To ensure that some feature really works (technically speaking) in all special cases, you'll need at least 5 rows in some far end tables and more in the tables in between. E.g. **5 – 10 – 30 – 20 – 10**

Reasons: You'll need **variation to the test data** = variation to possibilities for defining test cases = Test cases. In short remember these numbers: You'll need items that have the following relationships to other tables/types:

0 children
1 child
5 children (many children)


~~Bad test data~~
Good test data has variation

1. b "Empty" test data (an additional set of insert statements) for testing how the app works, looks and behaves with empty lists = empty database (or empty data tables with codes in code tables, e.g. colors in Color table)

- (In system testing) To ensure that empty lists of <u>any kind</u> don't cause anomalies in technical ways NOR in UX (user experiences) or graphical ways. This is something many developers often forget.
- (While in technical tests like module / unit tests) In addition to empty test data, programmers often forget to test their modules, methods and API interfaces with nulls or empty objects or empty strings or other special values:

  e.g. the following: last accepted and first unallowed boundary values, 0, -1, negative values, null references, database NULL, database searches that return an empty result set, empty String "", emptyish Strings with white space or punctuation " ", ".", "\t" =tabulator,"\n",)

2. Usability testing

200-500 rows per those tables that naturally have more than just a few lines. Natural amount to those who do have such, e.g. list of countries. Some lists naturally short, like "Product Owner, Developer, Scrum Master".

Reasons: You'll realize the possible need for e.g. search / filtering / paginating / visualizing scroll / memory aids /…

3. Performance testing

100 000 – millions of rows depending on the business case, technology, platform, networking, scalability of the solution, nature and length of the data on each row…

Reasons: to test network bottlenecks, memory, processing power, concurrency locks between database items …

Remark, while using auto-incremented id:s !!! (AUTO_INCREMENT, Identity, SEQUENCE, …other terms…)

- If you are using SQL DML INSERT statements to reset your database to the initial test data 'snapshot': You'll have to count how many rows the previous table insert did enter => use those in next statements as foreign keys. (And yes, we could also use database snapshot and recovery tools to simply write the test database tables to a certain situation, but we still need editable insert statements to create the snapshot in the first place.)
- In the case of external system relationships (e.g. to another REST API) make sure there is a fixed list of allowed values for the external id:s and use only them. Otherwise your test cases won't work consistently! = regression testing impossible and any testing needs manual fixing.)
- Deleting all data rows of a table **doesn't** usually reset the automatically incremented id:s. You'll need to fully DROP TABLEs to reset the counter(s). (((Or run some other commands to reset the counter (e.g. ALTER TABLE ….)))
- If you remove individual rows, a hole is often left in the id chain. Thus running through id:s is never correct way to handle sets of rows that have auto-increment keys=id:s. Well, never in other cases either. Better just always give nothing/null for insertion.
- **Make each table's id:s look different. 1->, 101->,201->,1001->,2001->, 10001->, Helps a lot in debugging.**
- Professional frameworks have ways to get back the inserted item/s' id/s (that the database creates and front end did not yet have).