An approach to unit testing

Rod Furey, Statistics Netherlands

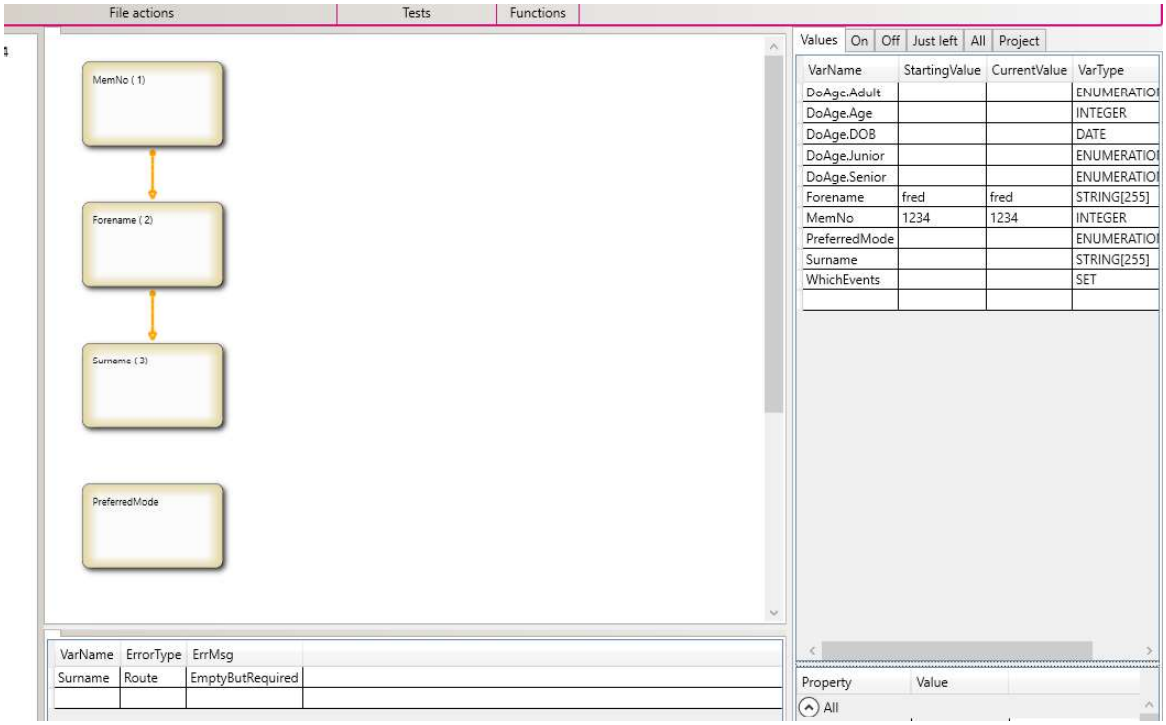
It is often mooted that it would be nice to be able to unit test various bits and pieces of a datamodel. One obvious section that can be tested stand-alone is a block. The concept presented here suggests selecting a block, extracting it (in various ways), defining a controller for the block and checking various things inside the block.

Test IDE

The basic Test IDE is based on various items that have been assembled over time and which have subsequently been collected in one place. The main concepts are as follows:

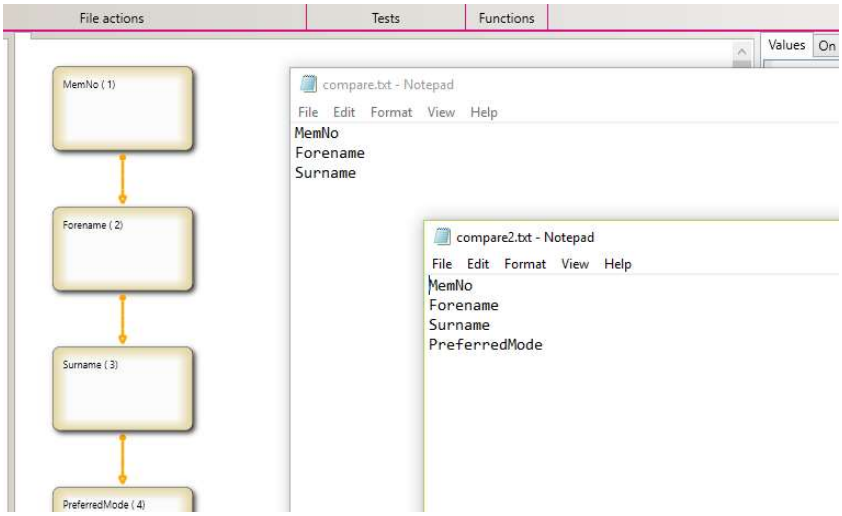
- display the asked fields in the instrument
- check the route by running the rules on a datarecord
- check for any errors that may occur
- extract the block and create the block controller
- perform the items 1, 2 and 3 on the extracted block
- if necessary, correct the block and re-prepare the instrument
- compare the results of the various runs
- repeat as necessary

The first 3 items are well-known and have been presented before.



Extraction of the block can be accomplished in a number of ways, depending on how the test IDE suite is constructed (e.g. directly from the sources or reconstructed out of the .bmix). The basic controller for the block doesn't have to be complicated and can be constructed by analyzing the source or retrieving FIELD information out of the .bmix. The problem with constructing the controller is working out what dependencies the block may have (e.g. ALIENs, references to things outside the block, PARAMETERS etc.) Dependency determination can be accomplished in a number of ways depending on what information is available to the programmer.

Once the controller has been created and compiled, the results of a test run can be checked in any number of ways, most usually by checking the results of executing the rules on a datarecord. Comparison of results can be achieved reasonably easily by marching the route and writing the output to a text-file which can then be fed into a text-file comparison program. In this way the output can be compared against a previous run or against an expected result. Again, if things don't match up, the analysis and repair steps can be repeated.



Data generation

Instead of creating the test datasets by hand, it is possible to generate them programmatically. There are various approaches to this, from randomly generating the data values for each field through supplying data for each field on the various routepaths to analyzing the flow of the source code and supplying values that exercise each route through the program code. Some of these possibilities are easier to program than others.

Once the datasets have been generated, they can be stored away and re-used at a later date. If a dataset is not valid any more, it will either produce an error when value assignment takes place (e.g. value out of range) or an error will have to be trapped when trying to assign to a field that no longer exists.

Conclusion

Creating the advanced items that facilitate unit testing (e.g. exercising every path through the code) costs time. Creating the basic IDE and supporting routines is relatively cheap and can deliver useful results relatively quickly whilst the more expensive items are analyzed and programmed.