Databorough

Controlling Test Environments with X-Analysis and X-Test

Simon Savage

WHITE PAPER

Table of Contents

Executive Summary....1
Introduction....2
The Challenges of Testing....2
  UI Testing....3
  Batch Testing....3
  Checking Results....4
  Checking Results....4
Building a Batch Test Environment....5
  Choosing appropriate Test Cases....5
  Setting Up a Physical Test Environment....6
The Tools for the Job....7
Setting Up Test Case runs....12
Running Test Cases....13
Reviewing the Results....14
  First View - The important difference is the first one you spot....14
  Refining by Field Exclusion....16
Summary....17

Executive Summary

Malfunctions in software cost money. The cost of software defects in a production environment is considered to be up to 100 times greater than that of the same defect when detected and corrected in earlier stages of development. An undetected error can have potentially disastrous consequences for a business.

Only thorough testing of changes before going live can achieve an acceptable level of stability before release. Tests may be of many kinds. Functional tests, validating and verifying, manual or automated tests, user interface or batch tests... not to mention all forms of non-functional and technical testing done by development teams.

Complete “by the book” tests engender skyrocketing costs that are not compatible with the cost-effectiveness inherent to business applications, so there will always be an element of compromise in the type and amount of testing that is done.
Validating the conformity of a new version to the (changed) user requirements and verifying that it works flawlessly are generally the most obvious types of functional tests to implement, and are well understood. Such tests will by definition be specifically adapted to each change that is being implemented, so will never be the same.

Verifying that nothing has been broken – regression testing – is the second key area of functional testing. Here we will very often be running the same tests over and over again across many generations of changes. Because of this, regression tests are a natural candidate for automation. And as ever in the software world, automation is the key to achieving considerable gains in quality, at a fraction of the cost.

This document is not a review of all aspects of software testing. It illustrates how the X-Analysis X-Test tool framework will help you achieve repeated gains in functional test operations by automating regression tests.

Introduction

What immediately springs to mind when we think about functional testing is a user (a developer in the early stages, an end-user in the later stages) using the new interface as thoroughly as possible and checking that the results conform to expectations.

This is a manual process, and is well adapted to each new change. Although such manual testing is generally efficient for locating defects in a software application, it is a laborious and time consuming process. Furthermore, it may not be effective in locating some kinds of defect – notably, software regression, where the change has broken something apparently unconnected that was previously working. It's generally not a practical proposition to test every single part of an application to make sure nothing has been broken. Regression testing needs a different approach.

We can use computer programs to automate some of the tests that would otherwise need to be done manually. Once tests have been automated, they can be run quickly and repeatedly. This is often the most cost effective method for regression tests on software applications with a long maintenance life, as even minor changes can cause potentially disastrous software regression.

Once such automated tests have been run, the biggest challenge is to detect exactly what has changed but should not have. Changes caused by the new software version will generally be immediately obvious, as the testers and the test process are very focused on these. But finding errors in other unexpected parts of an application, errors that may be buried somewhere beneath huge amounts of data, is a daunting situation, akin to the proverbial "needle in a haystack".

X-Test provides a framework and the appropriate tools to let us configure and automate test cases which will then run as batch jobs; to programmatically detect any unwanted differences in the results; and to review those differences in a user-friendly manner. In this document we will examine:

- The challenges of testing
- Where X-Test fits in
- Choosing appropriate test cases
- Setting up and running test cases
- Reviewing the results

The Challenges of Testing

To run the risk of stating the obvious, the central challenge of the test process is to find all of the defects. We will always worry that we have missed something important. The more tests we run, the better our chances of finding defects, though even the most critical applications can never be guaranteed to be 100% bug-free!
As we have said, to enable greater coverage with fixed resources, there is one

obvious direction to choose: that is to automate, to let the computer do as much of the work as is possible.

Let’s take a look at how we can automate our tests. We won’t be discussing the earlier stages of testing as done by the development team, in which we ensure that each individual component does its job reliably. It’s the later stages that interest us here, when the entire application is already up and running.

UI Testing

Although user interface testing remains the most natural and probably the most efficient way to find defects in a new version of a business application, it is not the object of this document and is only examined briefly in this section. User interface testing falls into one of two categories – manual tests or scripted tests.

Manual tests are the most revealing. A user-tester performs his tasks on the new version in a safe test environment. The user will see and report if any bugs appear or if results do not conform to specifications.

As users will generally be very concentrated on what they are doing, what they expect to see, and what results they actually do see, and also have the uncanny ability to find strange and unexpected ways of doing things, most defects can be found this way.

Automation is nonetheless an option for UI testing. It generally entails recording and playback using scripts (there are many tools available for this, such as IBM’s 5250 emulator, multiple open-source web interface scripting tools, etc).

The case for automation in UI tests is compromised by 2 basic issues. Firstly we lose the focus and the unpredictability of user input, which are the strong points of UI testing. Secondly, even the most minor change in the interface will render the script inoperable – and the whole point is to test things that have changed. As a result, scripts may need to be rewritten for each set of changes, which partially defeats the object of the exercise. Scripted automation in UI testing is however useful for regression tests on unchanged parts of the application.

Batch Testing

Running test on the batch processes is a major part of the tests of any large project. These tests are usually much simpler to run, as any changes are clearly taken into account by the code and have no effect on the “outer shell” script – otherwise the code will not work. For example, a monthly audit program may have undergone a extensive rewrite, but will still be triggered by an unchanged CALL ORDERAUDIT command. No particular user input is required.

As with UI testing, batch tests fall into 2 categories -

- batch tests to verify expected changes and check they are working as per spec.
- batch tests to check that nothing has been broken inadvertently – i.e. regression tests.

Checking Results

How can we verify our test results on batch test runs?

UI errors are likely to be visually apparent when the user examines his screen and printed output, and thus easily reported. Although some printed output may be produced by the batch runs and may easily be visually checked, when we do need to verify those test results – and also the UI test results in some cases – we need to check the data in the underlying database files. This is particularly true for regression tests, and it requires a different approach.

To verify the batch tests and check for regression, we need two sets of the application and database: one with the software version before the changes were made; and the second with the new software version and identical data. Then we can run the batch process on both sets and compare the results from the two runs. The differences will tell us if we have introduced bugs.

For example, on a monthly sales report, figures should all be identical (unless specific changes have been implemented on that report). In a customer account database file, the balance amount should tally, etc.

This is simple enough in theory, but in practice it can be a nightmare. Checking parallel values line by line throughout two large reports or files is time-consuming, error-prone, and likely to cause severe stress and job dissatisfaction amongst those doing the checking. When there are many reports or data files, this process is simply unrealistic.

The only realistic way to compare large numbers of parallel results is by automating the process. We need to set up software to read through all appropriate data, detect any differences in the two sets and report on them. If we don't automate this process, we are not going to run these tests regularly. Some initial thought and configuration is required to set up the automation, but once this has been done the same process can be used repeatedly over the application life-cycle, each time software changes are made.

Checking Results

This is where X-Test steps in. X-Test is part of the X-Analysis tool suite, dedicated to the problem of checking test data. X-Test provides a framework, tooling and a user-interface to facilitate batch test result comparison, thus increasing reliability and productivity of batch tests. Using the tool, we will dispose of the tools and a methodical approach to -

- set up the test environments
- define and populate checkpoints
- run the tests
- compare the resulting images
- review the differences in a user-friendly manner.

Building a Batch Test Environment

Choosing appropriate Test Cases

Not all parts of an application can or need to be systematically included in test runs. We need to think carefully about whether we want or need to test individual parts or the entire application, and what the consequences of each alternative may be. Are some batch processes critical, easily separated from other processes? Can we easily set up a valid stand-alone entry point to trigger this particular part of the application. The choice of test cases is never simple, and requires skilled and experienced users.

We will no doubt write down some test result specifications so we can check results against something specific – but these test result specifications are dependent on the actual changes. We will not be able to code result specifications to specify that “everything other than what is specifically changed should be unaltered”.

Performance Considerations - Disk space and performance are important factors in batch testing. If the production database occupies 500GB of disk space, we may not be able to make repeated copies of that base.

Firstly, available disk space may be insufficient. We will need several copies of the database – different sets for the different versions, copies for the checkpoints, and so on. If this adds up to terabytes of data, then this is unlikely to be readily available.

Secondly, manipulating a database of this size requires extensive system resources. Restoring or replicating such a large database may tie up a system for hours or even days.

Test run times are also a potentially limiting factor. If a batch process requires a complete weekend to run on a complete database, then careful planning must be done to take into account the potential need for an unscheduled reload and restart if errors are found.

Setting Up a Physical Test Environment

Data Extraction - Whatever our test cases and strategy, we will of course need a copy of our data for test purposes. In some situations it will of course be the most practical and option to copy the complete database. But generally, that option will not be practical. In view of the constraints of time and volume, and the need to focus our testing on manageable amounts of data, a complete copy is very often too large. A data subset is the ideal support for our tests.

Nonetheless, in many organizations, we realize that data subsets are not used extensively.

Why don't we see test data subsets more often? Largely because there are no practical means to build and manage them. Creating a coherent subset of data is difficult to accomplish, and consequently is not retained as an option. Even when an organization has taken the trouble to build a subset, there may be a reluctance to refresh and rebuild, as the subset is somewhat unwieldy.

Extracting that coherent subset can indeed entail a vast amount of research and work.

Imagine, for example, that we want to extract a simple subset for customer FRED from our order entry application. We need to extract the customer master record for primary key FRED, and any other customer files for that key. To be able to do anything meaningful with that record, we also need to pull in all related data, both from dependent files and owning files. Dependent data might include all order headers for FRED, then all order details and order history for those order headers. As each order detail record naturally has a reference to an item code, we then need the item master record for the order details items. Customer FRED may also belong to a particular company, so we will need that company header record. And so on.

If this needs to be done manually, it may take days or even weeks to achieve coherent results. To even envisage extracting a data subset, we will at very least need both a precise understanding of the data model to tell us which files are related to which other files, and an automated means of copying the related data into our test files.

Application Program and Data Objects - Locating and using the appropriate application program objects is not generally an issue, as only one copy of program objects is required on the system for multiple data environments. If we are going to run tests to compare results over different versions, we simply need to know how to implement the changes between versions in a controlled manner.

We may well not require the complete set of data files. A selection of the actual data objects that are impacted by our test run will be sufficient. This will make the test set smaller and therefore much easier to manipulate. Our challenge here is to locate all of those impacted and related objects without any room for error.

Hiding Sensitive Data - Running our tests may also highlight the problem of

confidential data. The people running the tests may not be authorized to view the data in the files. We may wish to scramble email addresses to make sure that no e-mails are sent inadvertently to real customers. Whatever the reason, we may need to change any sensitive data.

At the same time, while scrambling data sufficiently to make it unrecognizable and untraceable, we have to avoid generating user-unfriendly gobbledygook values such as a customer name of “lkjhdfpoi rltxcbg”: such values seriously handicap any user interaction during tests, as the users are no longer able to identify the values they see on screen or report.

The Tools for the Job

As in so many areas of software, we can use dedicated tools to increase our productivity to a level where tasks that were previously unimaginable become within easy reach.

We can use features of the X-Analysis tool suite to achieve simple and controlled setup and execution of these potentially complex operations that are an integral part of setting up a test environment. These include building a coherent data (content) subset, identifying and isolating a coherent set of objects, and automatic data encryption.

The X-Analysis cross-reference and data-model features make this possible.

Object Cross-Referencing - X-Analysis originally built its 20-year reputation by providing reliable and user friendly application cross referencing and documentation. The X-Analysis repository automatically builds a data-base of all requisite object cross referencing information. This lets us determine instantly which objects are related to which other objects, and how – read only, update, etc. We can view that information graphically in the client.

Figure 1 shows a data flow diagram centred on a selected program.

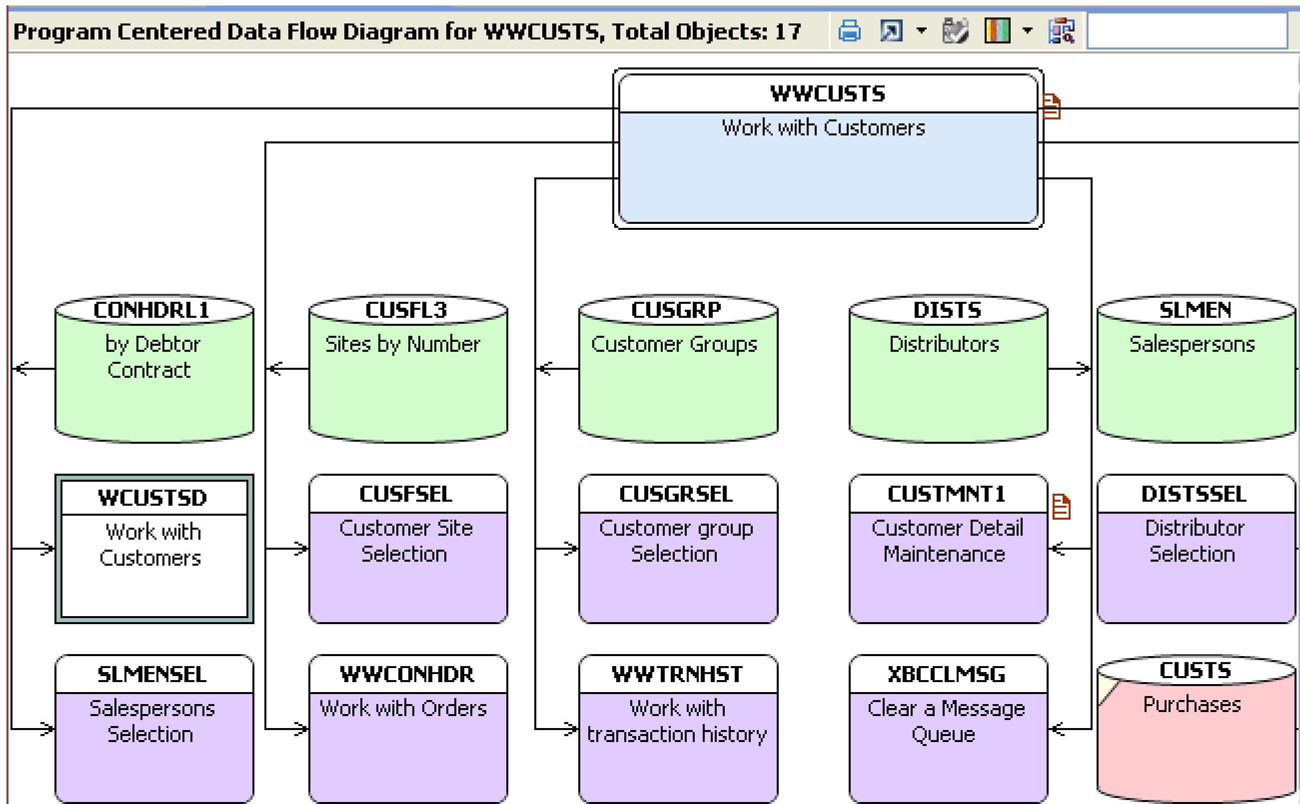

Figure 1: A program-centered Data Flow Diagram

The data flow diagram shows us which objects are in contact with which other objects, and how. A simple color-coding tells us instantly what is going on.

Data Modeling - X-Analysis provides a unique reverse-engineering data modeling feature, which examines the system and automatically builds complete entity-relationship data. It does this by examining in detail the database descriptions, the program code to find file fields that are used as access to other file fields, and by verifying the actual data in the database files. In this way, X-Analysis provides relationship details and foreign key details from an existing and undocumented database.

Figure 2 illustrates an data model extract centred on one particular file, with the appropriate relationship key details in the lower panel.

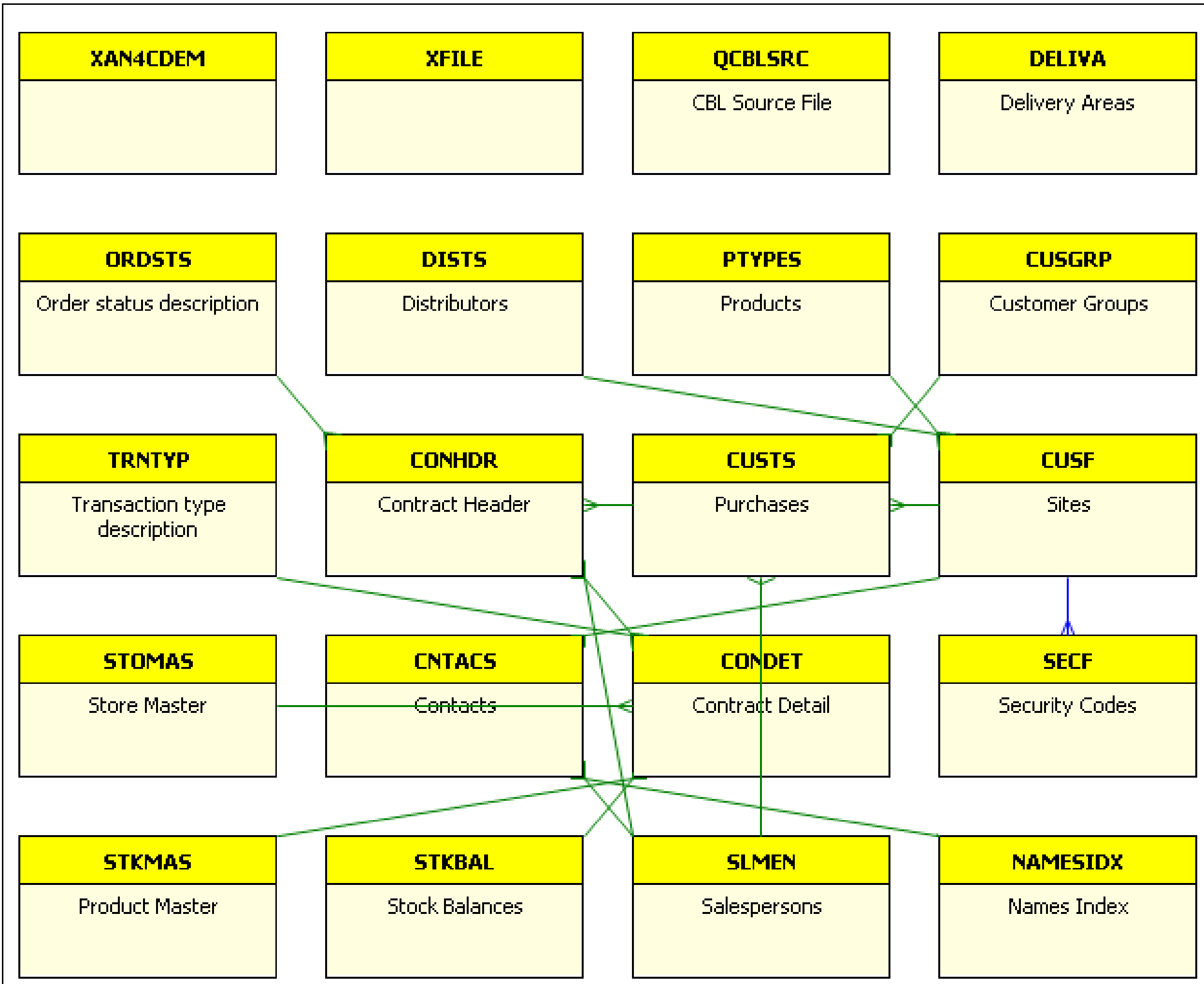

Figure 2: Viewing part of a data model

Application Area Management - X-Analysis provides facilities for subdividing an application area into groups of objects that meet user defined selection criteria. These criteria might be based on function or even generic name. When we associate an object with an application area, X-Analysis then uses the sophisticated cross-reference and data model information to automatically include all the related elements such as programs, displays, or files that we need in our application area. For our test environment purposes, we'd probably start from one or more programs or functions and automatically build up the list of all related objects.

Figure 3 shows how using object cross-referencing and data-modelling information, X-Analysis can propose simple rules to identify all related objects in one simple automated step.

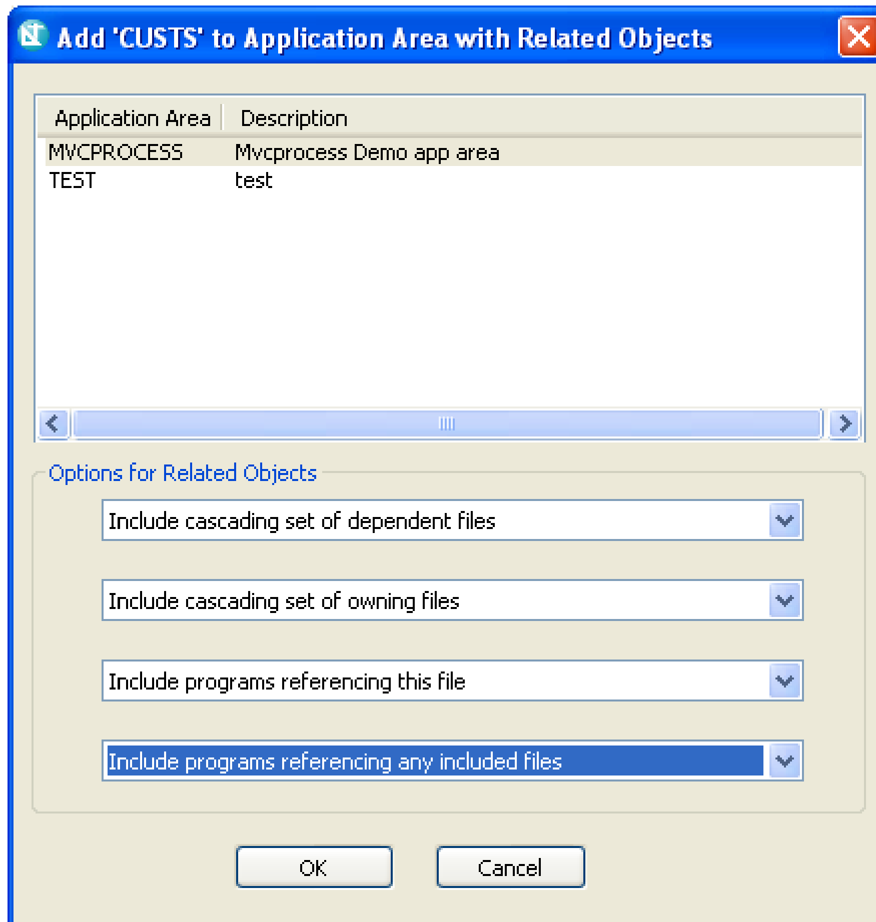

Figure 3: Rules for including an object in an application area

Application Area management also provides an option to build a new library containing all objects from that area, which we can use if we are pinpointing parts of an application.

Data Subsets - The subset feature uses the data model derived by X-Analysis to drill down through the complete data and extract all related data from initial seed record values. We specify the seed values we need to get the process started, then the data model takes over.

Figure 4 shows a user specifying a seed value. As X-Analysis has built the data model and knows where related data is to be found, this is all we need to extract a coherent subset.

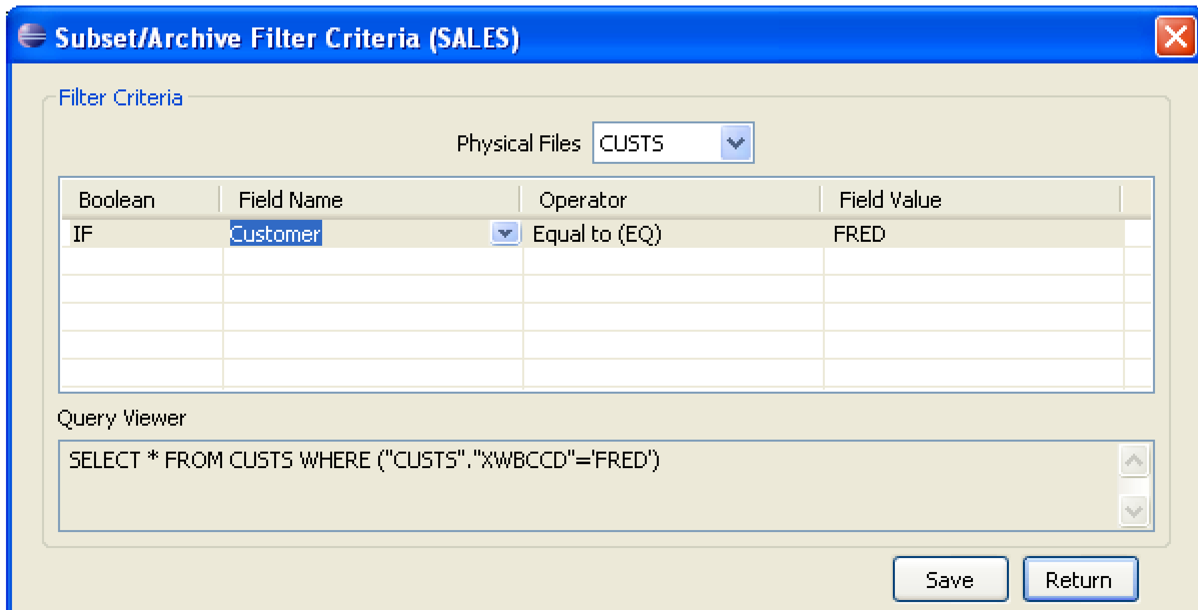

Figure 4: Entering a subset seed selection criterion

Once the subset seed values have been set up, we run the subset process which builds data in an appropriate library. Figure 5 shows the dialog from a simple right-click on the application area to run the actual sub-setting process.

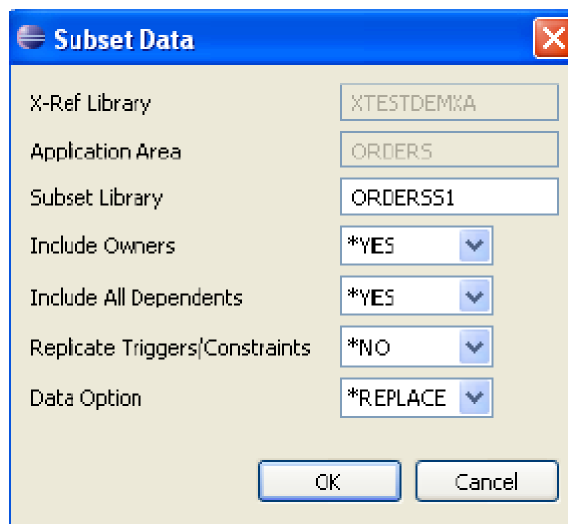

Figure 5: Running the subset extraction process

To sum up : this section has illustrated how X-Analysis provides all the necessary functionality to extract and manage the objects and data we will need for our test environments.

Setting Up Test Case runs

XA provides a complete control panel to help us configure the test environments and run the tests. Figure 6 shows an overview of an application we have configured for testing purposes.

Define the Test Process - We will need to know how to start our test process. The test framework requires a simple command or program to call in a consistent manner, to run a test process. Behind the initial call, there may of course be complex things going on, but the entry point is simple. This may well require some specific code such as a small CL script to enable X-Analysis to trigger the test process correctly. One of the advantages of IBM i is that such batch scripting is usually very simple to implement, so although the script will need to be developed for each test case, this is not a major job.

When our test trigger command or program is ready, we just need to register it in X-Test, so it can be run at will.

Set up Test Run Images - As we want to repeatedly compare data that we are busy changing, we need to define various checkpoints where we can freeze each successive step. We also need simple mechanisms to save the complete test result image to a checkpoint at the appropriate moment, and to reinstate a previous image when we wish to restart a particular run. The images will also be used to compare any two test runs and detect differences.

Set up Field Exclusions - In general, not all fields in a file need to be examined for differences. The most obvious example is a file in which each change updates a times tamp in a dedicated field. Such fields will logically never have identical values over any 2 runs. If this data was included when checking for differences, every single record in the file would appear as different. Because of this, X-Test provides a facility to specify any fields you wish to remove from the image comparison.

We can implement all of the steps outlined above via simple right-click options in the X-Test client user interface.

In **Figure 6** we can see an application area called TESTRUN that we have configured for our test run purposes.

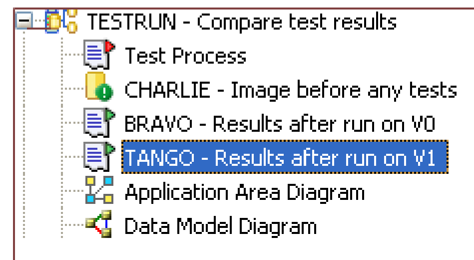

Figure 6: An application configured for test result processing

Running Test Cases

Once we have defined and configured our test environment, we want to run our batch test process on both the old and new versions of our software, then isolate the resulting data so we can compare results from those runs. This section describes the method for running those tests.

Set up an initial checkpoint - We've built the data in the appropriate library. We've got the correct software version (Let's call it V0, i.e. before any changes). We've set up X-Analysis with an application area and test process definition. So we're ready to run. Our first task is to save the image as an initial checkpoint (we'll call this CHARLIE). We're going to need to start all of our tests from the same point, and CHARLIE provides that point.

Run the test process on the original version - We trigger the test process from the X-Test interface. At the end we save the results, for example to a new image called BRAVO. This provides a Base checkpoint, where the run has taken place without any software changes.

Restore the original image and implement changes - We should now restore the image from checkpoint CHARLIE, then implement the requisite software changes – this gives us Version V1. Implementing the changes will require some thought, but should not pose any insurmountable problems! We may use a change management tool to implement and remove a set of changes, we may use a library in the library list which is blank for the base run on V0 and then contains the new object versions for the run on V1. Whatever process we use it should be easy, automatic and reliable!

Run the test process on the new version - Now we need to run our tests again. Once more we trigger the test process from the X-Test interface. Once more save the results of the test run to a pre-defined checkpoint, for example to an image called TANGO. We now have 2 separate images. BRAVO which contains the results after running tests with V0, and TANGO which contains the results after running tests with V1. We can now compare these results. We've done nothing particularly difficult to achieve this, we just made 2 extra copies of our test result data – but we can appreciate having simple and reliable tools that are dedicated to these operations and keep track of where we stand and what we've done so far.

We can repeat this as often as we like. Once the environment is configured, it can be used over the entire application life-cycle.

Register spool files - As X-Test has triggered the batch job that represents the test run, it is aware of any spool files produced by the job. X-Test is thus able to locate and register any spooled output produced by the test run. As copies are made of these spool files, we can compare spooled output the same way we compare database files. If the test process itself submits extra batch jobs, we'll need to register the spool files manually with X-Test.

Identify differences in the results of the two test runs - At this point, the runs are complete, the result sets have been saved in an appropriate image, and X-Test knows about all of this. Now we want to spot the differences in the two result sets. To compare the results of any two test runs, we just run the 'Compare Result' process in X-Test, which is nothing more than a right-click option on the appropriate test result name. We're prompted for name of the base test against which we want to compare our test run results, and X-Test submits an automatic batch job. This process reads through all files in the application area and builds an internal database that records any differences it may find.

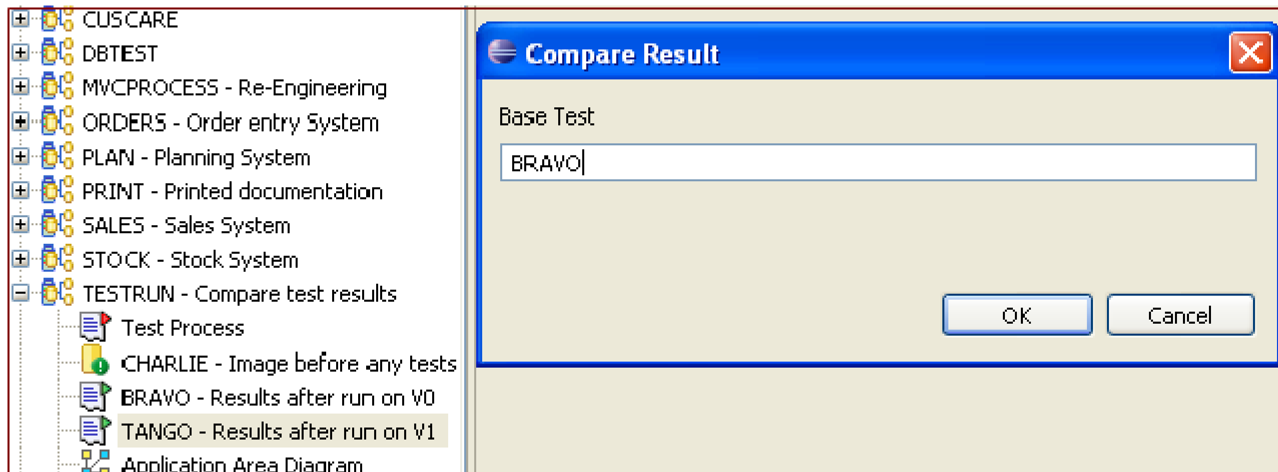

Figure 7: Comparing two sets of test run results

Reviewing the Results

First View - The important difference is the first one you spot

The objective of these test runs is to compare the results. We need to know where there are any differences between the two runs, and if so, where they occur. Once the compare results option has been run, X-Test instantly show us a list of the files where differences occur. We can expand the view to see the individual records that register differences.

X-Test doesn't necessarily show all of the records with differences – there is a limit set by the user on the maximum number of errors we want to be located and displayed. This will usually be set quite low, because what is important for us to know is the fact that something has gone wrong in a given file. We don't need to know absolutely everything that has gone wrong in that file, this would potentially provide far too much unhelpful information that we would waste time sifting through.

Even if there are hundreds of records in error, we only really need to know ONCE that something has gone wrong. We now know from X-Test that the results of our test run TANGO are not identical to results from BRAVO. The data on screen shows us where – in which file and which record – to look. Once we have spotted an error, the issue would most probably be passed to the technical team who are responsible for the application, to explain the difference and correct it if needs be.

Zoom into the details -

Record differences - X-Test does however display record details. We need to see as precisely as possible what the differences are before we can analyze their implications. The record detail display shows the data from the offending record and also the same record from the base version, highlighting any fields that are different in the two result sets. This makes locating any potential problems very fast.

Figure 8 & **Figure 9** show the detailed view of file records and differences. The upper part of the screen shows the files where differences have been located, expanded to show a summary of differing records. The lower part shows the field values for an individual record, with both test runs side by side.

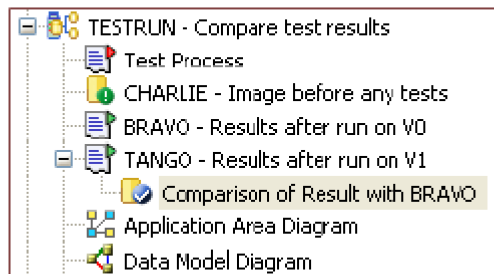

Figure 8: Test Result Area

Comparison of Results with BRAVO - App Area: TESTRUN

Result Comparison

Changed Record : "Store"(XWAACS) was: UK now: SWI
Changed Record : "Store"(XWAACS) was: UK now: SWI
Changed Record : "Store"(XWAACS) was: UK now: SWI
Changed Record : "Store"(XWAACS) was: UK now: SWI
Changed Record : "Store"(XWAACS) was: UK now: SWI
Changed Record : "Store"(XWAACS) was: UK now: SWI
Changed Record : "Store"(XWAACS) was: UK now: SWI
Changed Record : "Store"(XWAACS) was: UK now: SWI

⊟ Stock Balances(STKBAL)

Changed Record : "Onhand Quantity"(XWBHQT) was: 0.0 now: 1.0; "Pur Ord Balance"(XWBKQT) was: 0.0 now: 1.0
Changed Record : "Onhand Quantity"(XWBHQT) was: 0.0 now: 1.0; "Pur Ord Balance"(XWBKQT) was: 0.0 now: 1.0
Changed Record : "Onhand Quantity"(XWBHQT) was: 0.0 now: 1.0; "Pur Ord Balance"(XWBKQT) was: 0.0 now: 1.0
Changed Record : "Onhand Quantity"(XWBHQT) was: 0.0 now: 1.0; "Pur Ord Balance"(XWBKQT) was: 0.0 now: 1.0
Changed Record : "Onhand Quantity"(XWBHQT) was: 0.0 now: 1.0; "Pur Ord Balance"(XWBKQT) was: 0.0 now: 1.0
Changed Record : "Onhand Quantity"(XWBHQT) was: 0.0 now: 1.0; "Pur Ord Balance"(XWBKQT) was: 0.0 now: 1.0
Changed Record : "Onhand Quantity"(XWBHQT) was: 0.0 now: 1.0; "Pur Ord Balance"(XWBKQT) was: 0.0 now: 1.0
Changed Record : "Onhand Quantity"(XWBHQT) was: 0.0 now: 1.0; "Pur Ord Balance"(XWBKQT) was: 0.0 now: 1.0

Detailed Test Report Comparison

Detail Comparison of Test Result with BRAVO for File STKBAL - Rec No 126

Field	Test value	Base Value
Store (XWAACS)	RSA	RSA
Product (XWABCD)	000100	000100
Grp 1 (XWAGCD)	10	10
Grp 2 (XWAHCD)	10	10
Grp 3 (XWAICD)	30	30
U/M (XWA2CD)	EA	EA
Onhand Quantity (XWBHQT)	1.0	0.0
Pur Ord Balance (XWBKQT)	1.0	0.0
Stk Bal Sales Order (XWBM...	0.0	0.0
Stk Bal Production (XWFVQT)	0.0	0.0

Figure 9: Viewing record differences and their details

Journal images - Journaling provides invaluable help when we're trying to track what has happened in a database file. If the data files are journaled during the test run, X-Test will also let you scroll through the list of journal entries for a given record, so you can see which program is responsible for any given change. You can also zoom to view the details of the journal entry, to see exactly which fields were changed on an update operation.

Refining by Field Exclusion

We may decide that the issue that caused the differences to occur for a given field was not really a problem. If this is the case, we can change the field exclusion criteria and run the comparison again. In this way, we refine the results each time, and build up a robust test case which we can use over and over again.

Summary

Running batch tests and validating the results constitutes an essential part of any major software change cycle.

Implementing a regular and rigorous batch test strategy requires very careful setup of multiple environments, and thorough checking. These demanding and time-consuming requirements often lead to a reticence in this area.

Using the dedicated tools for the job that X-Analysis and X-Test provide, these requirements become simple to set up and manage. Checking an entire database becomes a simple task, whether we want to make sure that changes between two versions are as expected, or check that results of two test runs are identical.

Simon Savage
© Databorough