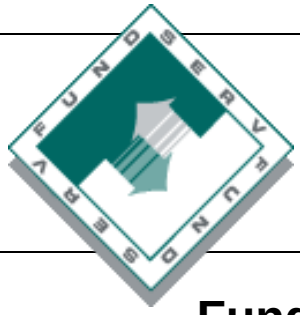

FundServ

FundSERV Automation Testing Tool Manual

Software User & Development Guide

Version: 1.0

Author: Russ Profant

Revision History

Date	Version	Description	Author
Oct 24, 2013	1.0		R. Profant

Table of Contents

1. Introduction 5
1.1. Purpose 5
1.2. Background 5

2 FATT Design & Structure 5
2.1. Conceptual Design of FATT 5
2.2. General Design Philosophy & Functionality 6

3. Data Design 9
3.1. Database Tables 9
3.1.1. TEST_CASE 10
3.1.2. TEST_CASE_DATA 11
3.1.3. TEST_PROCESSING_DATA 14
3.1.4. TEST_ACTION 15
3.1.5. TEST_OUTPUT 18
3.1.6. TEST 20
3.1.7. TEST_VARS 20
3.1.8. TEST_REPOSITORY 22
3.1.9. FS_ERROR_CODE 22
3.1.10. BUSINESS_PROCESS 22
3.1.11. TEST_DATA 23

4. FATT Usage 24
4.1 General Considerations & Approach 24
4.2. Two Types of Use 25
4.2.1. Testing Tool 25
4.2.2. Test Development Tool 25
4.3. Physical Location & Security 25
4.4. Directory Structure 26
4.5. Configuration File(s) 26
4.5.1. Configuration File Section 27
4.6. Shell Wrapper Script ‘fatt.bsh’ 28
4.7. FATT Customization 28
4.7.1. Data Customization 28
4.8. FATT Options & Parameters 29
4.8.1. Rules For Command Line Parameter Entry 29
4.8.2. Full List of Available Command Line Options 30
4.9. FATT Logging & Output 36
4.9.1. Default Logging 37
4.9.2. Debug Logging 37
4.9.3. Failure Only Logging 37
4.10. Practical Guide & Tips For Using FATT 38
5. Field / Element Testing 39

5.1. Single Value (per element) Testing 39
5.1.1. Test Case Consolidation & Refactoring 47
5.2. Multiple Value Element / Field Testing 48
5.2.1. POSITIVE Multiple Value And Element Testing 49
5.2.2. NEGATIVE Multiple Value Element / Field Testing 56
5.2.3. COMBINED Multi-field And Multi-value Testing 60
5.3. Field /Element Testing Recap 70

6. Business Process Testing 72
6.1. Basic Business Process 72
6.2. Business Process Mapping to FATT 73
6.3. More Complex Business Process Mapping to FATT 78

7. TEST DEVELOPMENT 82
7.1. FATT As a Database Interface Tool 82
7.2. FATT As a Test Case Development Tool 87
7.2.1. Order Submission Development 88
7.2.2. Contract Submission Development 95
7.3. Other Preconfigured Development Processes 99
7.3.1. CUTOFF_TIME 100
7.3.2. GET_CONTRACT 101
7.3.3. GET_ORDER 102
7.3.4. ORDER_TRANSFER 103
7.3.5. How To Create New Processes 104
7.4. Test Case Development By Cloning Existing Process or Test Case 104
7.5. Reclassifying Test Cases 110
7.6. Test Setup Tips & Tricks 113

8. Appendix 115
8.1. Regular Expression Use In FATT 115
8.2. User Exercises 115
8.3. Currently Defined Variables In SYS_VARS Table 117
8.4. Possible Enhancements 118
8.5. FAQ 118

9. Roadmap 121

User Manual

Larry Wahl the founder of Perl once famously quipped that software developers should never test their code because testing by definition requires that the tester should be at least 10% smarter (whatever that means) than the developer. This is based on the logical premise that the testing "thing" should be of "higher order" (in this case smarter) than the "thing" being tested.

The same logic applies to the testing systems. If anybody after reading this manual thinks that FATT is too complex then the answer is that it must be as it is supposed to be able to test any FundSERV system in part or in whole. And you can't test a number of complex systems with a less complex one.

1. Introduction

1.1. Purpose

This document describes how to use FundSERV Automated Testing Tool from a user perspective as well as from the perspective of future development or extension of the tool.

1.2. Background

FATT is an all-around testing tool developed to help FS testing teams primarily with the execution of predefined test cases and secondarily as a test case development tool.

The tool is a work in-progress and new functionality is being added daily. This document may not be fully up-to-date with the new functionality at all times.

2 FATT Design & Structure

2.1. Conceptual Design of FATT

The conceptual or logical design of FATT revolves around a few basic concepts:

- **Test Case** – is the main or primary concept within the system; everything else relates or derives in some way from and to the test case concept.

  A test case is a template for an actual test. A test case includes all related and necessary data for an execution of the test as well as for deciding the actual outcome of the test.

- **Test Case Data** is used by the test to perform its function.

There are 2 primary types of data. The static data is data that is not actually being tested but is still required in order for the tested functionality to work properly. The other type is the dynamic data; this is the data that the target system is being tested for.

The static data is usually stored in a template, a generic test file shared by related test cases. The dynamic data is usually stored in a database either as a literal value or a variable calculated at run time.

- **Test Case Actions** are steps that a test case must perform in order to fulfill its function as intended in the design phase.

  It's not necessary for a test case to go through all its predefined steps to come to a valid output. But it is necessary that an early termination of the test is caused by the actual testing system's fault rather than an incorrect or incomplete test case setup.

- **Test Output** is the state of the system at the end of a test.

  Test output is almost always represented/expressed by output data in some specific format (file, database records or other).

- **Test Result** is the status of the system in binary (2-sided) format based on the system output and compared to the predefined output in the test case.

- **Test** is an actual representation of a test case, that is, an instance of a test case. 'Test' and 'Test Case' are sometimes used interchangeably within this document.

These concepts can be summarized as below:

TEST USES TEST CASE DATA, GOES THROUGH TEST CASE ACTIONS TO ARRIVE AT THE TEST CASE OUTPUT TO DECIDE THE TEST CASE RESULT

To simplify the structure of a test and to elevate the testing definition from primarily a technical perspective to a more generic business perspective a concept of a 'Business Process' has been introduced.

IN FATT 'BUSINESS PROCESS' IS A NAME FOR A SET OF CLOSELY RELATED TEST CASES CONSISTING OF THE SAME SET OF TEST STEPS BUT USING DIFFERENT DATA INPUTS.

In the future the concept of a 'business processes may be expanded and further refined.

2.2. General Design Philosophy & Functionality

There were some general considerations that influenced the final product. Some of these were:

- This should be a CLI - **command line-type system**
- The command processing should be easy and **fault-tolerant**
- The program should be **customizable**
- This should be a **dynamic system** that can configure itself on-the-fly for each specific test case or group of test cases
- It must be **data-driven** where data is meant in a broad sense which includes test data as well as test setup/configuration data
- The design is based on **a test per test case and test case per single testing feature** paradigm. This is the general rule but there are exceptions within this functionality.
- It should be possible to **setup new test cases without writing any new code** just by configuring the existing functionality.

There were only a handful of tests developed at the time the project started so it was difficult to tell what specific functionality would be required to execute the tests being developed.

This led to the second major design decision for FATT, namely to develop a **generic system** that is capable of **performing common back end data processing tasks**. Therefore the functional ability at the business level needed was to:

Work with text-based files (but not necessarily text files, see below)
Read, Update and Write XML files
Read and Write into/from any number of databases (Oracle only)
Transfer Files between any two servers

In addition it became clear that this needed to be **a task-oriented, linear system**, that is, the system needed to be able to perform tasks in a certain sequence defined beforehand. This is because a functional test in the FundSERV environment consists of 1 or more actions/tasks/steps (these expressions are used interchangeably) that are chained together to create a full test case. Hence the following requirement:

Process any number of tasks (test actions) in a predefined sequence

One of the main aims of the system is to make the test action definable in such a way that no new code is required to setup a new task at least most of the time. In other words no matter what the specific task is it shouldn't take more than a configuration job to create it.

In the future it may be advisable to convert this linear process-based system to **a non-linear event-based system**.

In fact this should be the solution to strive for in the long term; a testing system that is akin to an integration server such as Jenkins. The system is always running but is in a hibernation mode until it wakes up when it observes some specific events such as code

changes and then performs a series of predefined tests.

The design of the system should be such that it wouldn't pose significant obstacles in converting the system to a daemon-type system even though it's not clear in how or when this should be done.

3. Data Design

The data model used in FATT is not a standard relational model. It uses relational database in Oracle but doesn't conform to many of the standard relational rules as data is mostly related by regular expressions which creates loose relationships rather than primary->foreign key constraints which create exact and specific relations between various records. This type of relationship can be called **associative** rather than relational when used in a relational database context.

The reason for that is the desire to keep the amount of data needed to be created to a minimum so that the users don't have to create tens of thousands of records just to fulfill the relational rules. This way the user is in complete control about which records in which tables are tied to which records in which other tables.

Of course with control comes responsibility. Therefore it's up to the user to ensure that all relevant and needed records are correctly configured and related so that a test can be performed properly.

There are actually three different forms of relationships in FATT data model.

1. There is the "normal" primary key, foreign key relationship. For example this is between a test case and the test case data.
2. Then there is a regex-based (regular expression) relationship. The use of regexes allows relating any number of records from any number of tables to themselves as sets at the expense of very loose and in some cases unclear relationships between related tables and records.
3. The third type is a value-based relationship. This is similar to regex-based relationship except in this case the regex is a literal expression (the value) that delineates the boundaries of the record set.

In addition there are combinations of these three that can be used throughout. However the combinations should be kept to minimum so as not to increase the complexity exponentially. This would easily break much of the functionality of the application. This is because it's virtually impossible to program for anything more than a few main combinations that are assumed to be used most of the time.

3.1. Database Tables

As mentioned above the database uses mostly regexes to associate records in different tables. The 'TEST_CASE.MATCH_ID' column is the key column that holds the expression/key against which other related records are compared. The column in the related tables holding the regex is called MATCH_ID_RX with 'RX' indicating that this column's values will be treated as regexes.

There are four main tables that hold the main data for the test definition and execution. These

are: TEST_CASE, TEST_CASE_DATA, TEST_ACTION, TEST_OUTPUT. The remaining tables are minor and support the main tables in one way or another.

3.1.1. TEST_CASE

This is the main table in the database. Structurally it copies the format of the excel spreadsheet template used for designing and performing tests currently by the testing team with a few minor changes.

Most of the columns are not used within the system, they are free form and are here only because they are found in the excel template. In the future they may be deleted if they are not used.

The only important fields in this table are: test_case_id, match_id, data_template, source_id, error_code and expected_result.

Column	Data Type	Description
TEST_CASE_ID	NUMBER(15,0)	Primary key
MATCH_ID	VARCHAR2(30)	Key used for regex matching
STATUS	VARCHAR2(1)	Status flag
DATA_TEMPLATE	VARCHAR2(50)	Holds the name of the data file to be used as the data template for file creation.
SOURCE_ID	VARCHAR2(50)	This field holds the source id used for the previously-run test case if we need to save that value.
BRD_REF	VARCHAR2(100)	Business request reference/description
SECTION	VARCHAR2(100)	Business/test section to which this test applies
SUBSECTION	VARCHAR2(100)	Subsection of the section above if any
DESCRIPTION	VARCHAR2(500)	Test description
REFERENCED_DATA	VARCHAR2(200)	Data that this test references
EXPECTED_RESULT	VARCHAR2(20)	The result expected; success or failure
USER_ID	VARCHAR2(20)	User id of the user who created ticket based on test results
TICKET_ID	VARCHAR2(20)	Ticket id if test failed and ticket was raised with the development team
COMMENTS	VARCHAR2(200)	Comments
UPDATED_AT	DATE	Timestamp
ERROR_CODE	VARCHAR2(20)	Error code expected for the test if test is expected to fail
TEST_CONDITION	VARCHAR2(500)	Specific test conditions, usually the test data in descriptive form
BUSINESS_PROCESS_ID	VARCHAR2(50)	Business process id if the test is associated to a business process

Here is a more detail description of how the more important columns should be used.

MATCH_ID

This should ideally be a unique value and should be descriptive in some way. There is no prescribed format but the user can come up with his/her own format.

For instance the format for 'buy fund contract field edits' is defined as follows: "24BFUVn(FAIL|PASS)" where '24' stands for version 24, 'BFU' stands for 'buy fund' and 'V' stands for field validation which when it fails produces an error file with the name that starts with 'F**V**T' plus the remainder of the original file instead of the original 'FST'. The 'n' is a unique unformatted integer value (supplied by the user).

Similarly we have '24BCOV' which is the prefix for buy contract field edits. The last part 'FAIL' indicates the expected result on the part of the system being tested which is either 'FAIL' or 'PASS'.

Again, the purpose of this field is to be able to tie any number of records in tables like TEST_ACTION, TEST_OUTPUT, TEST_PROCESSING_DATA to this specific record by the use of regular expressions.

DATA_TEMPLATE

This field specifies the data template to use for file generation (order, contract etc). This field will be overridden by the TEST_ACTION.DATA_TEMPLATE field should it have a value in it. Therefore in order to use this field the associated action to the test case must have null/no value in its DATA_TEMPLATE column.

SOURCE_ID

This field is used in combination with DATA_TEMPLATE field. If data template field specifies a file then if the file has more than one record in it SOURCE_ID can be used to specify which record should be selected as the data source/template for the test case.

This is for situations where users created a single source file with many records then this field will identify the correct record out of them all which is the record with SOURCE_ID value stored in its 'SrcID' element.

3.1.2. TEST_CASE_DATA

This table holds the data the tested system is being tested against. In other words this is the actual test data that decides the test's success or failure.

The test case data should mostly be literal but can include variables defined in the SYS_VARS table.

The precedence in the test case data is as follows:

1) **Lowest precedence have global vars, they have TEST_CASE_ID = -1**
2) **Second lowest are variables with BUSINESS_PROCESS_ID set to a value**
3) **The highest precedence has data with a valid TEST_CASE_ID**

This allows the user to override data values. For instance a global variable SettlData can be overridden by a business process variable which can be overridden by a test case variable.

Column	Data Type	Description
TEST_DATA_ID	NUMBER(15,0)	Primary key
STATUS	VARCHAR2(1)	Status flag
TEST_CASE_ID	NUMBER(15,0)	Foreign key from TEST_CASE table; global data fields are set to -1 meaning they are applicable to all data within the run of the program
TEST_FIELD_ACTION	VARCHAR2(30)	Allowed values are: empty, multiply *n*, exists, not exists, =, create
TEST_FIELD_NAME	VARCHAR2(100)	Name of the element; it doesn't need to be the full name/path but it should be a unique identifier otherwise the first element with the name is used
TEST_FIELD_VALUE	VARCHAR2(50)	The actual value if action is '=', for everything else no value/null
UPDATED_AT	DATE	Timestamp when the record was created/updated
STEP_SEQ	NUMBER(15)	If specified the data will be applied only in that action; if null the data is applied to each step (if field is found in that step)
OPTIONAL_FLAG	VARCHAR2(1)	Default is 'N' for no. That means the field must be found in the template otherwise the processing aborts
BUSINESS_PROCESS_ID	VARCHAR2(30)	Business process id for the test case
TEST_FIELD_SEQ	NUMBER	Field sequence position within the element, only valid when action = 'create'

TEST_FIELD_NAME

This holds the xml element name that the program will perform the "action" on. The data in this table is applicable only to xml files nothing else. Therefore only tasks which create xml files are relevant to this table.

This field specifies how the data element/field is to be treated. The element name must be specified in such a way that it can be uniquely identified. Most of the time the element name

itself is sufficient but sometimes the parent element may need to be specified as well. The format for specifying the element path is simply the element names divided/connected by a dot.

If there are several elements with the same name the indexing needs to be used in the form of 'element[n]' where 'n' is the index sequence starting with 1.

TEST_FIELD_ACTION

Currently these are the supported values:

- **exists** – doesn't do anything other than check that the element exists
- **multiply *n*** – clones the element 'n' times
- **create** – creates an element; if TEST_FIELD_SEQ field is not set the element is added as the last child of the parent otherwise the element is added to the position specified by the field
- **=** - the most widely used action, sets the element to the specified value
- **not exists –** deletes the elements and all its child elements
- **empty –** deletes the value of the element but not the element itself

In general the 'not exists' action should be preferred over the 'create' action. If for instance we have a conditional element that we need to insert for a certain test or group of tests, it's better to include the element in the template so that it doesn't have to be inserted and delete it when it's not required or when we are testing the lack of the element in the xml. The reason for this is that it's easier to delete an element than it is to insert it; when deleting the element only the element name must be specified and it doesn't need to be specified as the full path including all its parent elements. Most of the time element name is unique and it's sufficient for deleting the element. When inserting an element a full path has to be specified every time including the sequence position of the element within the parent element. So inserting element is usually more work or setup than deleting it.

STEP_SEQ

By default the data is applied to every test action if applicable. The program looks for the data element and if it finds it, it performs the desired action. This allows the user to specify only certain test step where to apply the field.

This is used mainly when we are testing whether illegal data changes to the business process are caught. In one step we send a data element set to A in another step we set it to B to see if the testing system recognizes the change.

OPTIONAL_FLAG

This field decides how a failure to find a specific data element in the data source is treated. If this field is set to 'Y' it will force the program to exit with error if it doesn't find the field in the data source (xml file). This only applies to data templates which are xml files.

TEST_FIELD_SEQ

This field is applicable to action ‘create’. If the field is set the program will set the new element at the specified sequence within the parent element. If not set the new element will be set as the last child of the parent.

The sequence can be a number starting from position 1. If the number of elements is dynamic and can change or if it’s not clear how many elements will be in the file the user can use 2 key words to specify where to insert the element:

- after
- before

Both must be followed by the name of a element such as ‘after:SrcID’ or ‘before:OrdType’

Additionally this field can also be used to order records with the same TEST_FIELD_NAME. In such a case we usually need to the records ordered in a specific way, one must be first and the other one second. We can use this field for that purpose.

TEST_CASE_ID

In most cases each tests case has its own test data and this field is used as the foreign key to tie the data to the test case.

If this field is set to ‘-1’ then it means that data is a global variable and will be applied to each record.

BUSINESS_PROCESS_ID

If this field is set the TEST_CASE_ID field will be disregarded, it doesn’t matter what its value is if any.

This allows specific test case data records to be assigned to a specific business process id.

3.1.3. TEST_PROCESSING_DATA

This table’s purpose is to hold the data that needs to be collected during the processing and then reused within FATT.

This data can be reused within FATT processing in at least 2 different ways; the data can be used as test data to be plugged into a specific data file during processing or it can be used as part of the processing instructions used by the program.

The program keeps all the data used during a test case run but only the data defined in this table can be used later in the processing.

One specific use is where an element exists in the template but is not defined as a data field in the TEST_CASE_DATA table. If it turns out that this field is also necessary in later stages of processing instead of defining this piece of data for every test case and entering it in the TEST_CASE_DATA which can be a lot of work, one can simply define the piece of data here for all the test cases.

For instance a SrcType is an important field that is submitted in the order file and needs to be included in the settlement file. However in the settlement file it's called 'OrdSrc'. This table enables a simple mapping of fields by allowing 'SrcType' to be entered in the 'DATA_FILED_SOURCE_NAME' and 'OrdSrc' in the 'DATA_FIELD_DESTINATION_NAME' of the same record. The program then collects a field called 'SrcType' when it processes an order and enters the field as 'OrdSrc' when it processes the corresponding settlement file.

If the mapping is not necessary then entering only the 'DATA_FILED_SOURCE_NAME' is sufficient to include the field in later stages of the processing.

Column	Data Type	Description
TEST_PROCESSING_ID	VARCHAR2(50)	Primary key
MATCH_ID_RX	VARCHAR2(50)	Match regex (currently not used)
TEST_SEQ	NUMBER(150)	Test step sequence (currently not used)
DATA_FIELD_SOURCE_NAME	VARCHAR2(50)	The exact name of the field as found in the data source (file, query etc.)
DATA_FIELD_DEST_NAME	VARCHAR2(50)	The name of the field as used in some other test step. This allows mapping of a field name in a previous to the same field but named differently in a later step.
UPDATED_AT	DATE	Timestamp
STATUS	VARCHAR2(1)	Status flag

3.1.4. TEST_ACTION

This is another key table. It holds all the steps necessary for performing every single test case step-by-step.

The table is called 'TEST_ACTION' which reflects the idea that a test case is built from a number of specific 'test actions'. However within the table the name used is 'step' not 'action'. This may be somewhat confusing but the reason for it was that if 'action' was used for column names there would be a column called 'action_action' which didn't look right. So to be consistent with column names 'step' word is used instead of 'action'.

Column	Data Type	Description
TEST_STEP_ID	NUMBER(15,0)	Primary key

STATUS	VARCHAR2(1)	Status flag
MATCH_ID_RX	VARCHAR2(20)	Match key for relating records from other tables
DESCRIPTION	VARCHAR2(100)	Description of the action
STEP_SEQ	NUMBER	The sequence position the step takes within the full test case process
STEP_ELEMENT	VARCHAR2(10)	What is the "thing" on which the test acts; currently only the following are supported: file, data, ftp, null
STEP_ACTION	VARCHAR2(10)	What action is to be performed on the element; currently supported are put, exists, create, update, get, run, show, submit.
STEP_DATA	VARCHAR2(100)	Currently not used and undefined
COMMAND_LIST	VARCHAR2(500)	List of commands to run on the command line, the commands need to be specified exactly as they would be if run by a user; they should have full paths and user must have needed permissions. The commands can be separated by a comma or a semicolon
SQL_LIST	VARCHAR2(500)	The list of SQL commands to be run, separated by a comma or a semicolon.
REPO_NAME	VARCHAR2(10)	Repository name as defined in the TEST_REPOSITORY table if needed for the action; this can be a database or an ftp server.
IN_DIRECTORY	VARCHAR2(30)	The directory where the source file is if this action involves a source file
OUT_DIRECTORY	VARCHAR2(30)	The directory where the output file should be created if the action creates an output file.
FILENAME	VARCHAR2(50)	The name of the file which is involved in the test action.
MAX_WAIT_IN_SECS	NUMBER(15)	How long should the program wait in seconds to give time to the testing system to respond to this action
DATA_TEMPLATE	VARCHAR2(50)	Data template from which a file is to be created
STEP_SCOPE	VARCHAR2(20)	Currently only 'batch' value supported
ONE_FILE_FLAG	VARCHAR2(1)	Controls whether the output produced is included in an already existing file or whether a new file is created

The most relevant fields are described in detail below,

STEP_ELEMENT

This is the component on which the test action is performed.

- **null** – no value, this is used only for command-type actions, actions that have 'COMMAND_LIST' field specified
- **ftp** – supports ftp processing (upload and download), this is more of an 'action' than 'element' but it's used as an element so that the action field can be used to specify the type of ftp: put or get
- **file** – the action is on file
- **data** – data means the action is on the database

STEP_ACTION

This is the actual test action performed on the element defined above.

- **null –** no value, just like for null element above, the field is not needed for command actions
- **put –** ftp upload; applies when element = ftp
- **get –** ftp download, applies when element = ftp
- **exists –** applies to 'data' element, it's for SQL select statements checking on the existence of specific records
- **create –** applies to files
- **update –** valid for 'data' elements' signifies SQL update/insert/delete
- **run –** applies to data elements means 'to run' stored procedures/SQL files against the database
- **show –** display contents of the file, applies to files
- **submit –** submit or copy (is equivalent to submit) a file to the processing directory (usually from the staging directory)

MAX_WAIT_IN_SECS

How many seconds to wait before determining the action result. The programs 'sleeps' for the specified number of seconds. This is used to give the system being tested time to respond.

If the element action = 'data exists' the behavior is somewhat different as the program will run the SQL statement every 5 seconds for up to this value while the record is not found in the table.

STEP_SCOPE

Currently only a single value is supported 'batch' (other than null). It instructs the program to treat the test action as an action for a batch of test cases possibly all of them.

When a test reaches a 'batch' action it stops and waits for all the other tests in the current run to finish/stop for batch action. Once all the tests have either finished or stopped the test manager runs the batch actions.

ONE_FILE_FLAG

This flag instructs FATT to include the output (an order or a settlement contract) in a compatible file if such already exists. The purpose for this is to speed up the processing. Because the TFS order processing is slow on a per file basis it makes sense to submit all relevant tests in a run in a single file. The settlement loader on the other hand is fast and therefore there is no need to send all related settlement records in a single file.

3.1.5. TEST_OUTPUT

Test output is the last major table in FATT. Not every test case needs to have a record in this table.

If a test case doesn't have defined test output, it's assumed the successful finish of the last step is the indication of the test's success. So by default a finished test case is a success unless output record or records are defined for it.

IT'S IMPORTANT TO UNDERSTAND THAT THERE ARE 2 DIFFERENT BUT RELATED OUTPUTS USED WITHIN FATT:

1. ***THERE IS THE OUTPUT FROM THE SYSTEM BEING TESTED; THIS IS USUALLY A PIECE OR PIECES OF DATA IN ONE FORMAT OR ANOTHER***
2. ***THERE IS ANOTHER OUTPUT FROM THE FATT ITSELF, THIS IS ALWAYS ONE OF THE 'SUCCESS/FAILURE' BINARY OPTIONS.***

OUTPUT #1 IS TOTALLY INDEPENDENT OF OUTPUT #2. BUT OUTPUT #2 FULLY DEPENDS ON THE OUTPUT #1.

IF THE OUTPUT #1 IS THE OUTPUT EXPECTED (WHATEVER IT IS) THE OUTPUT #2 IS A 'SUCCESS'. IF OUTPUT #1 IS NOT WHAT IS EXPECTED THE OUTPUT #2 IS A 'FAILURE'.

Column	Data Type	Description
TEST_OUTPUT_ID	NUMBER(15,0)	Primary key
STATUS	VARCHAR2(1)	Status flag
MATCH_ID_RX	VARCHAR2(15)	Match key
OUTPUT_TYPE	VARCHAR2(20)	Currently supported values are: 'file', 'file data'
OUTPUT_ACTION	VARCHAR2(20)	Currently supported values are: 'exists', 'not exists', '='
OUTPUT_FIELD	VARCHAR2(30)	The field holding the relevant data, in case of file, it's the file name
OUTPUT_VALUE_RX	VARCHAR2(100)	The regular expression to apply to the

		OUTPUT_FIELD to get the actual value
DATA_SOURCE_NAME	VARCHAR2(10)	Data repository name, currently not used
TEST_CASE_ID_REF_FIELD	VARCHAR2(30)	The field which holds the test identifier, currently not used
UPDATED_AT	DATE	Timestamp
OUT_DIRECTORY	VARCHAR2(50)	Output directory for files
STEP_SEQ	NUMBER(150)	Sequence in the output check if output needs to be checked in a specific order
WAIT_FOR	VARCHAR2(20)	How long to wait before checking for results
BUSINESS_PROCESS_ID	VARCHAR2(50)	Business process id to which record applies to

OUTPUT_TYPE

Defines the type of output expected for the test. Currently the valid data is:

- **file** – file output, the program checks for the existence of a file
- **file data** – this depends on the previous type; if a file exists a field within the file needs to be checked

OUTPUT_ACTION

Output action defines relevant action for the defined output in the above field. The currently supported values are:

- **exists –** output file should exists
- **not exists –** output file shouldn't exists
- **= -** the field in the file should contain a specific value defined in OUTPUT_VALUE_RX

OUTPUT_FIELD

This holds the name of the field that needs to be checked for output. This only applies to OUTPUT_TYPE = 'file data'.

For a file output this should hold the file name expected in the output. This needs to be only the file name and not the full file path because there is the 'OUT_DIRECTORY' field for specifying the path to the file.

The file name can be a variable defined in the TEST_VARS table.

OUTPUT_VALUE_RX

As the name of the field suggests the field holds either the literal output value or a regex value. If

This is a regex value the value is applied to the OUTPUT_FIELD expression.

3.1.6. TEST

This table holds the actual tests performed by the system where tests are instances of defined test cases.

During the testing this table holds the test action sequence number where the current processing is.

Column	Data Type	Description
TEST_ID	NUMBER(150)	Primary key
TEST_DATE	DATE	Timestamp
TEST_CASE_ID	VARCHAR2(15)	Test case which the test represents
TEST_RESULT	VARCHAR2(20)	The result; SUCCESS or FAILURE
TEST_OUTPUT	VARCHAR2(500)	Actual output
TEST_ERROR_CODE	VARCHAR2(100)	Actual error code if any
TEST_FILENAME	VARCHAR2(100)	The last file used in the test
TEST_TAG	VARCHAR2(20)	Test run identifier shared by all tests performed within the same run
TEST_SEQ	NUMBER(150)	Test action sequence within the test
TEST_STATUS	VARCHAR2(1)	Test status during the run
SOURCE_ID	VARCHAR2(20)	Source id value used in the test
ORDER_ID	NUMBER(15)	Order id value for the test

TEST_STATUS

The field holds the test status during the testing. The actual test result is indicated in the TEST_RESULT field.

The currently supported values are:

- **A** – active
  B – blocked for batch processing
  D – done

3.1.7. TEST_VARS

The use of variables within FATT is a very important feature that allows a lot of the system functionality. This table holds the global variables within the system. Most of the variables are static in the sense that they are initialized only once but there are some that are initialized on a per test case basis.

In the future there might be a need for a variable to be initialized on a per test action basis.

However this is not currently either required or implemented.

The system supports three types of variables:

- Literal values - any kind of value can be entered
- Function-produced values – the function must exists in the system in one of the modules, the function can take literal parameters but not variables as input and must produce a single value as output
- Composite values - any combination of literal values and variables is allowed but the variables must be specified in this format: ${variable} in order to be parsed properly.

When the variables are defined in this table the 'VAR_NAME' field holds the variable name. The name must be without dollar symbol '$'.

When the variables are used in other tables they must include the dollar symbol to indicate that the value is a variable and not a literal value.

Column	Data Type	Description
VAR_NAME	VARCHAR2(20)	Variable name is used as the primary key
STATUS	VARCHAR2(1)	Status flag
MATCH_ID_RX	VARCHAR2(15)	Match key currently not supported/used
VAR_TYPE	VARCHAR2(20)	Currently only 'function' value (an null) is supported
VAR_VALUE	VARCHAR2(200)	The value or function name with parameters if any
UPDATED_AT	DATE	Timestamp
CURRENT_VALUE	VARCHAR2(50)	If the value needs to be incremented the current value is stored in this field
REFRESH_INTERVAL	VARCHAR2(1)	How often should the variable be refreshed, a new value derived; this is relevant only for function-type variables

REFRESH_INTERVAL

This field controls how often the system variables are refreshed. Most variables need to be initialized/refreshed only once at the beginning of the run. However some variables need to be refreshed for each test case because each test case needs a unique value. In addition there is a case when a single test case supports numerous variations of the test such as when the test case data is in list format. This case is handled within the program and not in the database currently. In the future this may be transferred to the database functionality.

The currently supported values are:

- R – refresh/initialize the variable per run (of the program), this is the default

T – refresh per test case, initialize sequence variables per each test case within the run

3.1.8. TEST_REPOSITORY

This table holds the repository definition. Repository in FATT means any kind of remote network point that needs to be reached during testing.

Currently two types of repositories are supported: databases and remote ftp servers. The fields in the table enable a complete definition of these two types of repositories.

Column	Data Type	Description
REPO_NAME	VARCHAR2(10)	The name is also the primary key
STATUS	VARCHAR2(1)	Status flag
REPO_SERVER	VARCHAR2(15)	The IP address of the server
DB_SCHEMA	VARCHAR2(15)	DB schema if this is a DB
REPO_USER	VARCHAR2(20)	User name
REPO_PASSWORD	VARCHAR2(15)	Password
UPDATED_AT	DATE	Timestamp
REPO_TYPE	VARCHAR2(20)	Supported values are ‘ftp’ or ‘db’, for information only

3.1.9. FS_ERROR_CODE

This table stores the currently defined error codes. The codes before version 24 contain only a single value, an integer. The codes since version 24 will support error code and code detail; hence these are 2-value error codes. For simplicity the 2-value codes are stored as a single value and it’s up to the program using this table to handle the error codes properly.

Column	Data Type	Description
ERROR_CODE	VARCHAR2(30)	Error code, primary key
DESCRIPTION	VARCHAR2(500)	Error description
FIELD_NAME	VARCHAR2(50)	Field/element description to which the error applies
XML_TAG	VARCHAR2(30)	The xml field name to which the code applies to
STATUS	VARCHAR2(1)	Status flag

3.1.10. BUSINESS_PROCESS

Business process table doesn’t hold much data as a business process in FATT environment currently is nothing more than a label for a set of related test steps and test cases. Nevertheless it’s very useful in that it elevates the whole system into the realm of business functionality and

away from a strictly technical perspective.

In the future the concept of business process definition should be expanded further and should be the primary conceptual user interface to the system.

There are only 2 fields other than status field and both are free form and can be anything the user wants. Both should be as precise as possible and the business process id should also be terse in addition to being precise because it will be used as an (undefined) logical foreign key in other tables.

Column	Data Type	Description
BUSINESS_PROCESS_ID	VARCHAR2(50)	Name of the process
PROCESS_DESCRIPTION	VARCHAR2(500)	Process description
PROCESS_STATUS	VARCHAR2(1)	Status flag

3.1.11. TEST_DATA

This table is used to hold live test processing data. FATT doesn't log data by default because it's a big performance hit. To log the runtime data '-log Data' option must be specified.

The main use for this table is for split or interrupted testing where a portion of the test case is executed and later on, usually the next day, the remaining portion is performed. For these test restarts 'log Data' is necessary so that the test can work correctly as designed.

Column	Data Type	Description
TEST_DATA_ID	NUMBER	Primary key
TEST_ID	NUMBER	Test id
TEST_SEQ	NUMBER	Test action sequence within the run
TEST_FIELD_NAME	VARCHAR2(50)	Field name
TEST_FIELD_VALUE	VARCHAR2(4000)	Field value
TEST_TAG	VARCHAR2(30)	Test run tag
UPDATED_AT	DATE	Timestamp

4. FATT Usage

4.1 General Considerations & Approach

Before we get into the details of how FATT can or should be used let's first talk in general terms. A user should always keep in mind these two points:

FATT is a tool, it doesn't do anything that hasn't been programmed into it; it doesn't know anything and in general has no ideas of its own.

It's up to the user to come up with ideas and then find ways how to implement them using FATT.

FATT at its core is business-agnostic, there is very little business-related (specific to FundSERV) functionality built into it. This was done on purpose as there were two good reasons for it:

1) Because the business requirements and business processes were not very clear at the time of FATT design and development

2) The program needed to be designed in a very generic way so that it could accommodate wide range of business functionality without additional development work.

Therefore the onus is on the user when using FATT. The user needs to understand the problem at hand in this case, what business functionality is being tested, what the actual functionality is, how it is performed in the production environment, what are the steps that need to be taken, what the data is, where it's coming from, what the result should be etc. Only when all this is known and well understood should the user take that understanding and try to model a testing process on it.

The good news for the user is that he/she is not starting from scratch there are plenty of test cases defined already and these can be reused in 2 different ways.

1) The test case itself may be reused, specifically the test case actions and output, the user just creates a new test case record and test data.
2) The test case can be copied/cloned to a new test case with new test case actions and the user makes any adjustments to the actions as needed.

The user should have a good idea what tests are already defined in FATT and reuse any existing functionality already there as much as possible.

If any new major testing initiative is to be undertaken, the user(s) should first determine whether and/or how this testing initiative relates to any other tests already defined within the FATT. Beyond the practical considerations mentioned above (reusing existing FATT functionality) there is a more general logical consideration. The testing needs to be

categorized and recorded properly to fit within the existing testing procedures and processes. Ad hoc testing is not very useful or effective in a complex IT environment. The breadth and scope of such an environment is such that it requires thousands of tests to be executed and to execute them ad hoc is just not feasible.

Hence a serious conceptual categorization of any testing activity is a must. Usually the categorization is just an extension of the existing testing taxonomy. However it's possible and conceivable that at some point a complete overhaul of the testing strategy may be attempted or a major portion of it. FATT enables a complete realignment of test cases and the related data without recreating all the records from scratch.

4.2. Two Types of Use

FATT has two related but rather distinct uses:

4.2.1. Testing Tool

FATT is primarily a testing tool; it takes preconfigured test cases from the databases and runs them. At the end it collects the results.

In this scenario the test configuration is assumed and it outside of the scope of the tool. This mission will be explored in detail first as it is very important that the users first understand what FATT does and how it does it. Once that is understood users can start exploring how FATT can help them develop their test cases.

4.2.2. Test Development Tool

In the previous type of use the test case configuration is assumed. In practical terms this is not sufficient as the test cases have to be created somehow so that they can be run.

This is where the second use comes in for FATT. It's also developed as testing development tool. It's not just that a new label has been affixed to it; it actually does have additional significant functionality that enables us to classify it as 'development tool'.

This functionality will be explored in detail in subsequent sections.

4.3. Physical Location & Security

The program is on the QA server with the current IP address of '142.48.44.63'. It must be run by 'nsrapp' user. No login or authentication is required or supported.

4.4. Directory Structure

The code is contained in a number of files. These can be classified into 2 categories (excluding the configuration file):

1. **Wrapper** program is in **$FATT_INSTALLATION/bin/fattRunTest.pl** file

2. All **business objects** are defined in the files in **$FATT_INSTALLATION/lib** directory, an object per file as is common in Perl.

The home directory is '/app/fsrv/nsr/fatt'. There are several subdirectories in the home directory:

bin: holds all the executable files including the main script fattRunTest.pl
lib: holds all the Perl source files for business objects
config: holds all the configuration files, the main configuration file is 'fatt.config'
docs: holds all the documentation files

The wrapper program is as its name suggests used only to wrap the business functionality. I should not be used for any functionality now or in the future.

The purpose of the wrapper is to collect run parameters and control how the program is run.

Currently the program is assumed to be used as an ad hoc command line tool but can run as daemon as well. This may not be very useful at this time but might become useful in the future if or when the program is transformed into a non-linear event-driven system.

There is also the thread functionality included in the program. Since multi-threading is not supported by the current Perl installation, this functionality cannot be used on the back end in the UNIX environment. In can be used in Windows however so far it hasn't shown any performance improvements over single-threaded operations.

Another functionality that is already in the wrapper but not used is the ability run the program as a Windows service. This however requires that the program be compiled into binary executable using ActiveState compiler and installed (as a Windows service) on the machine where it needs to run.

4.5. Configuration File(s)

FATT uses a configuration file to get the basic attributes for the system. The configuration file must be specified either in the environment variable FATT_CONFIG or on the command line as a parameter 'configFile' such as:

perl /app/fsrv/nsr/fatt/bin/fattRunTest.pl –configFile /app/fsrv/nsr/fatt/config/fatt.config

Configuration file can be shared or every user can have his or her own configuration file specified. Another option for users is to share the config file but have their own section within the file.

There is also a '.fatt_profile' file in the same config directory. It looks like this:

alias ll="ls -ltr"

export FATT_LIB=/app/fsrv/nsr/fatt/lib
export FATT_CONFIG=/app/fsrv/nsr/fatt/config/fatt.config

export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/app/fsrv/nsr/fatt/instantclient_12_1

The 'alias' is only for convenience and is not necessary. The other three variables are required.

'FATT_LIB' is used to set the library directory for Perl business objects, 'FATT_CONFIG' was explained above and 'LD_LIBRARY_PATH' is used to set the library path for Oracle Perl driver.

4.5.1. Configuration File Section

The config file can be divided into sections where each section is independent of each other and only a single section can be used per run.

The section is specified on the command line as 'configSection'. In the file the section starts with a header in square brackets such as:

[fatt_user]

A header is then followed by a list of options and parameters one option/parameter per line such as 'dbUser=user_name'. Any option can be specified in the configuration file.

The options must be spelled out fully and with the correct case because 'dbUser' which is a valid option is not the same as 'dbuser' which is not. This is different from the the command line where they don't need to be fully spelled out and neither do they need to be in the correct case.

Hash '#' at the beginning of a line is considered a comment and the line is disregarded.

In general the most common parameters should be entered in the configuration file such as the login for the system database. The options used on a per run basis should be specified on the command line.

4.6. Shell Wrapper Script 'fatt.bsh'

The purpose of the shell wrapper is to further simplify FATT usage so that the user instead of typing (while in 'fatt/bin' directory):

perl ./fattRunTest.pl –configSec fatt_user

only needs to type (while in 'fatt/bin'):

fatt.bsh

The 'fatt.bsh' itself looks like this:

```
#!/usr/bin/bash

. /app/fsrv/nsr/fatt/config/.fatt_profile

cd /app/fsrv/nsr/fatt/bin

perl ./fattRunTest.pl -configSection fatt_user "$@"

exit 0
```

The first line creates the command shell, the next line sources the profile for the program, the next line sets the fatt run directory, the next line actually runs the FATT program and the last line sets the exit value to success.

4.7. FATT Customization

FATT can be customized by users. Customization can be done by creating one's own section in a config file or even by creating a whole new config file and including specific processing options/parametrs in the configuration.

Most importantly the user can enter his/her own data which the system will use instead of the database-defined data.

4.7.1. Data Customization

There are tow important data parameters in FATT:

1) dataList
2) defaultDataList

These 2 parameters are related but their purpose is different.

defaultDataList

This parameter should be entered in the configuration file and it should hold the key basic fields the user uses thropughout his/her work such as MgmtCode, DlrCode, FundID etc.

Default data will always override the database data with the same field or element name as well as the template data.

defaultDataList is a kind of permanent data override.

It can be entered in the configFile this way:

defaultDataList=”MgmtCode:YYJ;DlrCode:2402”

dataList

dataList is a kind of short-term override. It’s meant to be used for test case development as a quick and easy way to check out different data options for a specific test case.

dataList will override defaultDataList values for elements with the same name. This option should be always entered on the command line and never put into the config file.

4.8. FATT Options & Parameters

4.8.1. Rules For Command Line Parameter Entry

There are a few rules that need to be observed when entering paramters on the command line.

- YOU MUST USE DASH ‘-‘ OR DOUBLE DASH ‘—‘ before each parameter otherwise it will be skipped (and you can spend hours wondering what the heck is going on).
- Parameter names are not case sensitive in any environment so ‘-matchIdList’ and ‘-matchidlist’ are equal
- Parameter name short cuts are allowed, that is the shortest possible part of the name that uniquely identifies the parameter is allowed such as ‘-deb’ for ‘debug’ or ‘-match’ for ‘matchIdList’

- Single or double dash before the parameter name is accepted such as '-help' or '—help' (unlike the general Unix practice with long names preceded by a double dash and short names by a single dash)
- Parameter values must be separated from the parameter names by a space such as '-id 9876' or an equal sign '-id=9876'
- Parameter values can be surrounded by quotes or double-quotes but don't have to be such as '-template "/tmp/myfile.dat"' or '-template /tmp/myfile.dat'. Sometimes due to a character in a name the parameter value may not be parsed properly; in such a case the value must be included in quotes (single or double)
- List values are allowed for parameters which end with the word 'list' such as '-matchIdList'
- The list values must be separated by a comma, or a semi-colon and should be in double quotes such as '-testIdList "7821,7822,7823"' to avoid problems with shell extrapolation
- 'dataList' option needs to be specified by listing the field or element name followed by a colon and the then the actual value such as '-data "STATUS:I,ONE_FILE_FLAG:Y"'
- The options ending in 'flag' need to be specified using either of the pair 'Y' ('Yes') or 'N' ('No') of characters not by using the numbers '1' or '0'; 'y' and 'n' are accepted as well
- Parameter values cannot be shortened and usually they are case-sensitive. For instance '-show bus' is not valid and neither is '-show BUSINESS' because 'business' is a value and the program expects it to be specified as '-show business' to show all defined records from the BUSINESS_PROCESS table.

4.8.2. Full List of Available Command Line Options

FATT offers a long list of options. To find out the full list with a short description type in:

fatt.bsh –help

What you get is a full list first in one line and then the full list again but this time with short description, one option per line for easy readability.

The square brackets mean that the parameter is optional, 'str' meant string, 'int' means an integer, options ending in 'list' and/or followed by '<str>…'indicate the parameter is a list and can take 1 or more values separated by a comma or a semicolon. Option name not followed by anything is a Boolean-type option.

Boolean options are specified with only the option name on the command line but they must have a value assigned if they are in the configuration file. For example '-deleteFiles' (or just '-deletef') is sufficient on the command line but in the config file it must be defined as 'deleteFiles=Y'.

Usage:

./fattRunTest.pl -configSection <str> [-configFile <str>] [-dbUser <str>] [-dbPassword <str>] [-dbServer <str>] [-dbName <str>] [-dbSchema <str>] [-dbType <str>] [-pwdCmd <str>] [-debug] [-sqlPrint] [-parallel] [-emailList <str>...] [-emailOnFailureOnly] [-daemon] [-daemonTimeout <int>] [-fullDate] [-hostName <str>] [-programName <str>] [-runType <str>] [-help] [-matchIdList <str>...] [-testIdList <str>...] [-template <str>] [-tag <str>] [-stagingDir <str>] [-showFailuresOnly] [-noSubmit] [-noResultCheck] [-deleteFiles] [-waitInSecs <int>] [-submitAllFirst] [-action <str>] [-businessProcessList <str>...] [-logData] [-dataList <str>...] [-show <str>] [-outFile <str>] [-adHoc] [-continueOnError] [-errorCodeList <str>...] [-skipTestCaseData] [-create <str>] [-delete <str>] [-update <str>] [-id <str>] [-defaultDataList <str>...] [-restart] [-noDataUpdate] [-comboCount] [-recordId <str>] [-checkSubmitFile <str>]

Mandatory Arguments:

-configSection <str>

Configuration file entry name

Optional Arguments:

[-configFile <str>]

The full path to the FATT configuration file; overrides $FATT_CONFIG environment variable

[-dbUser <str>]

Valid FATT database login with the required permissions

[-dbPassword <str>]

Database Password

[-dbServer <str>]

The database server name as defined within the network in the appropritae file or files

[-dbName <str>]

Not relevant for Oracle which doesn't support the concept of a database

[-dbSchema <str>]

The schema name for the FATT database; normally it should be FATT. This is only relevant to Oracle

[-dbType <str>]

Database Type, defaults to Oracle

[-pwdCmd <str>]

Full command to run to extract the FATT database password for the 'dbUser'; this option is only relevant if password ('dbPassword') is not specified.

[-debug]

Include all the logging information available.

[-sqlPrint]

Log all the sql statements used by the program to the screen and/or file

[-parallel]

Perform parallel processing, currently not used

[-emailList <str>...]

List of user emails to receive the recap

[-emailOnFailureOnly]

Email user(s) on job failure only

[-daemon]

Run as daemon, currently not used

[-daemonTimeout <int>]

Daemon timeout in seconds, currently not used

[-fullDate]

When the program is run in single-run mode it displays only time in the output but not the date. In daemon mode the time alone is insuficient because the daemon can run for many days therefore the full date is required

[-hostName <str>]

Name of the server where the program is running; internal use by the system

[-programName <str>]

Name of the running program, internal use by the system

[-runType <str>]

Type of the current program run, internal use by the system

[-help]

Display usage information, print this very output

[-matchIdList <str>...]

Test match ids to process; this can be a a list composed of actual values or expressions SQL wild card expressions such as "24BFUV%,24BCOV%"

[-testIdList <str>...]

Test IDs to process

[-template <str>]

The full path to the data file

[-tag <str>]

The tag for the test run

[-stagingDir <str>]

Temp or Staging directory

[-showFailuresOnly]

Show only failed tests

[-noSubmit]

Don't submit the tests

[-noResultCheck]

Don't check the result

[-deleteFiles]

Delete data files when done; this option forces the program to delete the files from the staging directory once they are submitted for processing to TFS/NSR

[-waitInSecs <int>]

Wait in seconds before result check; defines how many seconds the program waits after file or files have been submitted for processing before checking for results. This assumes at least 1 record is defined fro the test case in the TEST_OUTPUT table

[-submitAllFirst]

Submit all tests before result check; this options should be used in normal circumstances as speeds up the processing significantly since there is only a single wait for the output before the first result is checked

[-action <str>]

Action(s) seq to process; this can be a single action or a seqnece which can be specified like this '1-6'

[-businessProcessList <str>...]

Business process list to test

[-logData]

Log the data in the database, this option should not be used in normal circumstances as it slows down the processing significantly

[-dataList <str>...]

List of field/value pairs to use; the following format must be used: 'field1:value1,field2:value2' etc.

[-defaultDataList <str>...]

List of default field/value pairs to use;this allows splitting the common base personal data from the 'dataList' such as 'MgmtCode' and 'DlrCode' from the 'dataList' which should hold more dynamic data; dataList overrides defaultData, the same for,mat as for data list must be used

[-show <str>]

Show test case database records

[-outFile <str>]

File name for the output data

[-adHoc]

Ad hoc run with no test cases

[-continueOnError]

Continue even if error encountered

[-errorCodeList <str>...]

Show error code(s) defined in the error table

[-skipTestCaseData]

Don't use data from the table TEST_CASE_DATA

[-create <str>]

Create a record in the database

[-delete <str>]

Delete record(s) in the database

[-update <str>]

Delete record(s) in the database

[-id <str>]

Record ID to copy, delete etc

[-restart]

You must specify a tag or test ids or business ids using the appropriate options

[-noDataUpdate]

Don't update data in the data template but submit it

[-comboCount]

Calculates and prints the total number of test combinations for the data specified

[-recordId <str>]

Variable name used as record id

[-checkSubmitFile <str>]

This option can be used instead of 'waitInSecs' option to tell the program to keep checking for the time specified (in secs, default is 600) if the submitted file is gone from the submission directory and then proceed to check for results

[-makeMatchId <str>]

Create match id from the given prefix

[-runBusinessId <str>]

Run tests under this process id

4.9. FATT Logging & Output

Logging is currently only supported to the screen. Therefore in Unix/Linux environment the output can be redirected to a file. In Windows it currently cannot.

Logging in FATT is rather sophisticated because it needs to fulfill 2 contrary requirements.

When testing is going well as little logging as possible is needed because the user is not interested in successful processing, this is assumed.

When testing is not going well (it's failing) as much logging as possible is needed because the user needs to find out what the problem is.

Obviously the program is not smart enough to decide which type of logging is required this needs to be done by the user. The user controls the level of logging by setting one or two options when the program is run.

There are 2 basic levels of logging and another level which is more a log surppression than logging so 3 levels altogether.

4.9.1. Default Logging

Default loggin is whatever is displayed by default by the program without specifying any logging options. At this level there is not much logging and what there is is mostly generic. However it does display all the file names created but for testing output it only prints a message saying it's checking the output.

4.9.2. Debug Logging

The 'debug' mode is enabled by using the debug option '-debug' or any unique short cut such as '-deb'.

In 'debug' mode very detailed information is printed about the processing including all xml data from each xml file being processed.

This is a lot of data that is being output and therefore this option should really only be used in development and for failing test cases.

Debug should provide sufficient information for a knwoledgable user to be able to figure out the problem and correct it.

4.9.3. Failure Only Logging

To turn on this logging type '-showFailuresOnly' option must be specified.

As already mentioned this is a level of logging achieved by log suppression rather than logging itself.

The idea behind this options is that when a large number of test cases is being tested even the default amount of logging is usually too much and really rather pointless because the user is not interested in anything other than failed tests. For well developed and mature test cases there can be very few to no failures so any other output to be seen is usually undesirable.

With this option the failure logging is rather detailed but everything else is mostly surpressed except a message here and there just to let the user know that the program is not hung but still running.

4.10. Practical Guide & Tips For Using FATT

- Action timeout: Always remember that the system being tested needs time to respond and the time is almost never instantenious. Therefore most test actions need to have a timeout scheduled in the TEST_ACTION.MAX_WAIT_IN_SECS field.

- Result timeout: Also before FATT checks for results, it should pause either for a specific time defined on the command line as 'waitInSecs' or for tests that have a final action a file submission 'checkSubmitFile' option should be used. This ensures the program keeps checking whether the last submitted file or files have been processed (deleted from the directory where they were submitted) before checking for results.

  'checkSubmitFile' option is a lot more practical than waiting a predefined amount of time and should be used whenever possible.

- You should always use the option 'submitAllFirst' to submit all test cases before checking for results. This is significantly faster than submitting a test case wait for results then submitting another test case wait for results and so on.

  For instance running 13 negative multi-value tests using 'checkSubmitFile' option one by one takes 13 minutes. Running the same tests with 'submitAllFirst' option takes less than a minute.

- Always use a single identifier type when selecting test cases: use either 'testIdList' or 'matchIdList' or 'businessIdList' but never their combination as this is undefined and will probably not work correctly.

- When running test cases always run together only test cases that have the same execution path, in other words have the same test steps, in the same order. This applies only if there are batch-type steps defined in the test execution path. If there are no batch steps you can mix various type of test cases together when running them.

5. Field / Element Testing

The field or element testing is about the simplest type of testing available on the back end. At least the test cases are simple but the execution may be a bit more complicated as we will see later.

In field testing we are submitting different values in a specific field and observe the tested system behavior which we compare against the expected behavior of the system. The expected behavior in this type of testing is always such that we expect testing failure for invalid values and non-failure for valid values.

FATT allows two basic ways in which to test elements or fields:

1) Single value testing
2) Multiple value testing

5.1. Single Value (per element) Testing

With single value testing each test case has a single value for which it tests. Since most of the time there are numerous fields and numerous values per field which need to be tested this approach is rather cumbersome and slow as hundreds or thousands of test cases need to be created just to perform the most basic field validation.

The advantage is that this setup is very straightforward and largely mechanical. It doesn't require much thinking. The mechanical nature of the test makes the setup quite quick considering the volume that is usually required.

EXAMPLE

This example shows not just how test cases are developed but how test case classification can change or evolve over time.

The business process 'CONTRACT_FIELD_VALID' covers the test cases from this spreadsheet:
http://compass2012/Departments/SolutionDelivery/QualityAssurance/Library/Public%20Documents/Projects/FundSERV%20Standards%20v24/Functional/Settlement%20Business%20Edits%20-%20BuyCon%20and%20BuyFund%20Test%20Script%20v1.0.xlsx

This process covers all settlement file field validation tests. There are about 2,700 of them and they are all single value tests.

However this didn't start as a specific business process but rather as groups of related test cases as can be seen by looking at the spreadsheet. They were initially all setup and performed manually and then they were imported into FATT. The business process was created later by consolidating a number of sets of tests which were created from the

spreadsheet.

The specific test subsets within this business process can be distinguished by looking at the values in the MATCH_ID field of each test case as well as by looking at 'SECTION' field of the TEST_CASE table. The 'SECTION' field is the 'human' format while the MATCH_ID is for programming puproses. The values below are the first 6 characters of the MATCH_ID field of the test case.

24BCOV – Buy contract field edit (V file output)
24SWIV – Switch contract field edit (V file output)
24SCOV – Sell contract field edit (V file output)
24DCOV – Dilution contract field edit (V file output)
24BFUV – Buy fund field edit (V file output)
24SFUV – Sell fund field edit (V file output)
24SREV – Settlement record field edit (V file output)
24SIEV – Settlement instruction field edit (V file output)

This categorization enables us to test only specific subset of test cases if we want to. For instance if we want to test only settlement 'BuyFund' element and its elements we can do that by typing (while in FATT/bin directory):

fatt.bsh –matchIdList 24BFUV%

The percentage sign'%' at the end indicates that this is not a specific match id but rather a SQL expression where percentage sign '%' stands for 'any list of characters after the initial '24BFUV' sequence of characters (in this case), in other words we are telling FATT to get all test cases that start with '24BFUV'.

The percentage sign'%' doesn't have to be at the end, we can do something like this 24FCOMV%FAIL. The expression must be a valid SQL LIKE expression.

Test Case ID: 5866

Let's have a look at a single test case from the above group and its setup in all the major tables. We will select a random 'negative' test case where 'negative' means we are expecting the tested system to fail in the processing of the case.

1. Here is the test case definition from the TEST_CASE table:

TICKET_ID	undef
STATUS	A
EXPECTED_RESULT	V file
BUSINESS_PROCESS_ID	undef
TEST_CASE_ID	5866

TEST_CONDITION	SellSegMat.SellFund.Dedns.LSIFClawbackProv=123456789.123
REFERENCED_DATA	undef
SECTION	Sell Contract Edit
USER_ID	undef
UPDATED_AT	undef
COMMENTS	undef
SOURCE_ID	undef
MATCH_ID	24SCOV2458FAIL
SUBSECTION	undef
ERROR_CODE	undef
DESCRIPTION	undef
DATA_TEMPLATE	sellSegMatTemplate.xml
MATCH_ID	24SCOV2458FAIL

An important field is a 'TEST_CASE_ID' field which is an auto-generated primary key field.

Another field is 'MATCH_ID', this is an interesting field. It's a free form and user supplied. It allows users to group many related tests together so that only a small number of related records (in other tables such as TEST_ACTION and TEST_OUTPUT) needs to be defined for each test group instead of single record per each primary record as is usual in the relational data design.

In this case we decided to use the testing taxonomy implied in the spreadsheet where the test cases were saved. In the spreadsheet each major settlement element had separate worksheet listing the tests related to the element. This arrangement is expressed by the first 6 letters of the MATCH_ID.

In our case '24SCOV2458FAIL' is a settlement contract test for FundSERV version 24 with a unique id of 2458; the last part 'FAIL' means that the test is expected to fail. The failure in this case is recorgnized if the testing system (NSR) produces a 'V' error file. 'V' error file is a file in the output directory with the same name as the input file except with the second letter 'S' being replaced by 'V'.

Another important field in the TEST_CASE table is DATA_TEMPLATE which holds the data template file (name). This is used in case the data template is not defined at the level of the action which it isn't in this case. The reason the DATA_TEMPLATE is not set for the test action in this case is because we need several different data templates for this set of tests. Because the test action has only a single field for data template the data template must be defined at the level of test case and not the test action.

The logic in the template data field as used in FATT is that TEST_ACTION.DATA_TEMPLATE overrides TEST_CASE.DATA_TEMPLATE field.

This logic may seem counterintuitive because a test action is a record shared by many tests and therefore it would seem logical that if a test case needed a different template its field should override the more generic action field.

Normally this would indeed be the case. However in FATT multiple actions per test case can exist and if the TEST_CASE.DATA_TEMPLATE field were to override the TEST_ACTION.DATA_TEMPLATE field it would not be clear which action is being overridden if two or more actions were to make use of DATA_TEMPLATE field (not every action uses a data template).

If the action field has no value it means the field from the test case needs to be used. If the action has the field set, the test case field is disregarded.

The consequence of this logic is that if a single test case needs to use a specific data template which is different from other test cases using the same action record then all test cases must have their data template set at the level of the test case and hence the action data template field needs to be empty.

The rest of the fields other than TEST_CASE_ID are mainly descriptive, they don't play a role in processing.

2. There are 2 action records defined for the test in TEST_ACTION table:

   The first action record is to create a settlement file, the second action is to submit the file to NSR. This is expressed in TEST_SEQ numbering, see the values below.

TEST_STEP_ID	24
STEP_DATA	undef
STATUS	A
BUSINESS_PROCESS_ID	undef
FILENAME	$FS_FILENAME
SQL_LIST	undef
REPO_NAME	undef
STEP_ELEMENT	file
MAX_WAIT_IN_SECS	undef
OUT_DIRECTORY	$STAGING_DIR
MATCH_ID_RX	24SCOV
IN_DIRECTORY	/data/fsrv/ffs/mgmt/templates
ONE_FILE_FLAG	N
STEP_SEQ	1
COMMAND_LIST	undef

STEP_SCOPE	undef
DESCRIPTION	create settlement file
STEP_ACTION	create
DATA_TEMPLATE	undef

TEST_STEP_ID	42
STEP_DATA	undef
STATUS	A
BUSINESS_PROCESS_ID	undef
FILENAME	$FS_FILENAME
SQL_LIST	undef
REPO_NAME	undef
STEP_ELEMENT	File
MAX_WAIT_IN_SECS	undef
OUT_DIRECTORY	$IN_DIR
MATCH_ID_RX	24...V
IN_DIRECTORY	$STAGING_DIR
ONE_FILE_FLAG	N
STEP_SEQ	2
COMMAND_LIST	undef
STEP_SCOPE	undef
DESCRIPTION	submit settlement file to NSR for processing
STEP_ACTION	submit
DATA_TEMPLATE	undef
TEST_STEP_ID	42

The first action creates the xml file based on the template defined above in the test case. The actual action to take is specified in STEP_ELEMENT (file) and STEP_ACTION (create/submit) fields. Most of the other defined fields use a system variables to setup the file name and the directories.

What is interesting about these 2 records above is that they have different values for the same field MATCH_ID_RX. The file creation action uses ‘24SCOV’ while the file submission action uses '24...V'.

This is because they need to capture or match to different records from the test case table. The file creation action only applies to ‘24SCOV’-type records, which are ‘sell contract field edit’ test cases. These need to use their own specific data template which is different from say ‘24SFUV’ test cases. Therefore they need their own action with their own ‘DATA_TEMPLATE’ field although that is empty in this case in the action record.

The second action which is test submission is the same for all tests that submit a settlelement file to NSR regardless of the actual content of the file (buy or sell or switch etc).

Therefore the regex to capture all these is '24…V'. With this regex in the MATCH_ID_RX field we can capture all records that start with '24' contain any 3 characters followed by a 'V'.

The variables used within the tables are defined in the SYSTEM_VARS table.

3. There are 5 records in the TEST_CASE_DATA table

TEST_DATA_ID	10449
TEST_FIELD_ACTION	=
STEP_SEQ	undef
OPTIONAL_FLAG	1
STATUS	A
BUSINESS_PROCESS_ID	undef
TEST_CASE_ID	-1
TEST_FIELD_NAME	Date
TEST_FIELD_SEQ	undef
TEST_FIELD_VALUE	20131217
LIST_FLAG	0

TEST_DATA_ID	10270
TEST_FIELD_ACTION	=
STEP_SEQ	undef
OPTIONAL_FLAG	1
STATUS	A
BUSINESS_PROCESS_ID	undef
TEST_CASE_ID	-1
TEST_FIELD_NAME	SrcID
ORIG_TEST_FIELD_VALUE	$SOURCE_ID
TEST_FIELD_SEQ	undef
TEST_FIELD_VALUE	AES2
UPDATED_AT	undef
LIST_FLAG	0

TEST_DATA_ID	14425
TEST_FIELD_ACTION	=

STEP_SEQ	undef
OPTIONAL_FLAG	0
STATUS	A
BUSINESS_PROCESS_ID	undef
TEST_CASE_ID	5866
TEST_FIELD_NAME	DlrCode
TEST_FIELD_SEQ	undef
TEST_FIELD_VALUE	2400
LIST_FLAG	0

TEST_DATA_ID	13318
TEST_FIELD_ACTION	=
STEP_SEQ	undef
OPTIONAL_FLAG	0
STATUS	A
BUSINESS_PROCESS_ID	undef
TEST_CASE_ID	5866
TEST_FIELD_NAME	MgmtCode
TEST_FIELD_SEQ	undef
TEST_FIELD_VALUE	YYY
LIST_FLAG	0

TEST_DATA_ID	7541
TEST_FIELD_ACTION	=
STEP_SEQ	undef
OPTIONAL_FLAG	0
STATUS	A
BUSINESS_PROCESS_ID	undef
TEST_CASE_ID	5866
TEST_FIELD_NAME	SellSegMat.SellFund.Dedns.LSIFClawbackProv
TEST_FIELD_SEQ	undef
TEST_FIELD_VALUE	123456789.1
LIST_FLAG	0

The first two are **global variables** because they have TEST_CASE_ID = -1.

The other 3 variables are the actual test case variables as they have their TEST_CASE_ID set to the id of the test.

Out of these ‘MgmtCode’ and ‘DlrCode’ are test case setup variables. They are needed for the test case to work at all but they do not directly participate in the test case, in other words they are not the data which the test case is testing. Because the test case uses varaiables for its setup, these two are the variables needed for this test case; they are used in directory and file names for the test case.

The pure test case data is only the last variable 'SellSegMat.SellFund.Dedns.LSIFClawbackProv'. This variable is the purpose for the whole test case. Looking at the test data value we can see that we are testing whether a 3 digit decimal number is rejected by the system being tested in the ‘LSIFClawbackProv’ field.

4. Finally, the output is defined in the TEST_OUTPUT table:

STEP_SEQ	1
OUTPUT_ACTION	exists
OUTPUT_VALUE_RX	s/^FST/FVT/
STATUS	A
WAIT_FOR	undef
BUSINESS_PROCESS_ID	undef
OUTPUT_TYPE	file
TEST_OUTPUT_ID	4
DATA_SOURCE_NAME	undef
OUT_DIRECTORY	$OUT_DIR
TEST_CASE_ID_REF_FIELD	undef
OUTPUT_FIELD	$FS_FILENAME
MATCH_ID_RX	24...V.*FAIL
UPDATED_AT	undef

The key fields that control the output processing are ‘OUTPUT_TYPE’ and ‘OUTPUT_ACTION’.

In this case we are looking for a file (OUTPUT_TYPE = file and OUTPUT_ACTION = exists) in the OUT_DIRECTORY with the name defined by the variable $FS_FILENAME to which OUTPUT_VALUE_RX field needs to be applied.

This is a regex value (regular expression) and when it’s applied to the filename it changes the second character of the file name from ‘S’ to ‘V’, the rest of the name stays the same.

The test is a success if the defined file is found, otherwise the test is a failure.

To run the specific test case we can type:

fatt.bsh –test 5866

If we were to run the 2,700 test cases 1 by 1 it would take us a long time a lot better option is to specify alist of 'SQL like' expressions like this

fatt.bsh –matchIdList 24BCOV%,24SWIV%,24SCOV%,24DCOV%,24BFUV%,24SFUV%,24SREV%,24SIEV%

We can enclose the list in quotes or not, our choice. Either way this is not a convenient command to remember and use.

We need something simpler. We need to refactor or consolidate our test cases.

5.1.1. Test Case Consolidation & Refactoring

Refactoring is a popular expression in software development; it applies to the process of software reengineering. The reengineering in this case is not really about adding new or even extending current functionality. It's mainly about improving the current functionality, it's about making whatever there is in the software currently work better, faster or to make the software easier to use.

And that's exactly what we need; we need to find a way to put all these related test groups under one roof so that we can treat and run them with a single command using a single identifier, that is, we want the same functionality but execute it differently.

FATT allows us to do that through its use of the 'BUSINESS PROCESS' concept.

We can define a business process for them and run them under that business name. This is the record from the BUSINESS_PROCESS table:

BUSINESS_PROCESS_ID	CONTRACT_FIELD_VALID
BUSINESS_PROCESS_DESCRIPTION	Field validation for contracts

Once we have the business process defined we can relate required records to this process by applying the business process id to the tables with the BUSINESS_PROCESS_ID column such as TEST_ACTION, TEST_CASE_DATA, TEST_OUTPUT.

Once we have the business process set up we can run the process like this:

fatt.bsh –bus CONTRACT_FIELD_VALID –submit –check

which is a short cut for

fatt.bsh –businessIdList CONTRACT_FIELD_VALID –submitAllFirst –checkSubmitFile

5.2. Multiple Value Element / Field Testing

Single value field testing is not very convenient as it requires numerous tests to be setup. To get around that problem we can setup a single test with multiple values per field and with numerous fields per test.

When we setup a test with a list of values per field and 1 or more fields per test the program will create all possible combinations of the data in all the fields. Obviously the more values per field and the more fields there are the higher the number of data combinations.

Each data combination works a lot like a full independent test case except that in FATT it's not treated independently but only as a child of its parent test case. In other words the 'child' test is processed like a real test case but its result is subsumed by the parent test case.

Hence in this case the **parent test case** is actually a collection of child testcases where children get everything from the parent except for the input data which is specific to them and is supplied by the program as 1 piece of the all the data combinations created.

Out-of-memory Problem

The number of combinations can easily reach hundreds of thousands or millions. The memory will run out with the current hardware when the combinations are in hundreds of thousands. There is a way to find out how many combinations will be produced from the current data setup by using '-dataCombo' option.

If the program runs out of memory it won't cause any damage you will just get a message saying 'out of memory' and the program will exit. In that case you have to remove some of the values or fields.

The easiest way to go about it is to set STATUS for some of the data records in the TEST_CASE_DATA field to 'I' which will make the record 'inactive'.

In fact some users may find it more convenient to test each field on its own for a list of valid or invalid values by setting all the other fields to 'inactive' status and running the test field by field each time activating a single field and deactivating the rest.

With multiple value testing it's important to realize that in essence we have many tests or subtests grouped in a single test case. But we can still only have one test result setup for it.

Therefore all the subtests within a multi-value test case must be either 'positive' or 'negative' which means that we need to have a test case where all values we are testing for are expected to be valid so that the testing system doesn't reject them, that's a positive test.

If we were to mix positive and negative values i.e. valid and invalid field values the program would not be able to make a correct decision whether the test case is a success or a failure.

The above is not entirely accurate as we will see shortly. We can have a combined positive and negative testing within a single test case and there is a way to setup the output and result check.

5.2.1. POSITIVE Multiple Value And Element Testing

Test Case ID: 7841

Let's have a look at a specific test case which was setup to test the contract file 'SettlRec' element and its simple-type fields.

There are 14 fields and these are the values as setup in the TEST_CASE_DATA table; element name is on the left and element value on the right (the rest of the columns are left out for clarity), these are all lists of values where individual values are separated by a comma (LIST_FLAG column is set to 'Y'):

DlrAcctID	0,QAZXSWEDCVFRTGB,9O0AAAXXXX54321
FundAcctID	0,QAZXSWEDCVFRTGB,9O0AAAXXXX54321
OrdID	abcd1,MMza19,6109876
OrdStatus	A,R
SrcID	10o9,N,0981653 4BHYTREW
RevnNum	40,01,99
AcctDesig	1,2,3
MgmtCode	AA0,XZ1,XXX
DlrCode	1234,ZZZZ,1Q2W,MTXS

AcctType	01,20
CreateDate	20140101,99991212,10000202
RepCode	0,Z0,9IUHN,02309,DDESA
OrdSrc	D,F,I
OrdType	D,M,W

All the above values were carefully chosen and they are in fact all valid values for their respective fields.

Now if we want to test all possible combinations of the data we should first check how many there are using this command:

fatt.bsh -test 7841 -combo -action 1 -nosubmit

With the above command we are telling FATT to use test id 7841 to find out all possible data combinations for active data records (-combo) and run only the action # 1. The second action is file submission but we are not creating a file we are just counting data combinations so action # 2 is really pointless. We can run it but it will take a bit longer and fail though we will still get the data combination count.

With the above command we get the following output:

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
10:39:30 I OS:          SOLARIS (command line run)
                        SERVER:  qsz24nsr
                        PROGRAM: ./fattRunTest.pl
                        PROCESS: 22629
================================================================

                    FATT GENERAL PARAMETERS

Configuration file path             : /app/fsrv/nsr/fatt/config/fatt.config
Config file entry                   : fatt_user
Database User                       : qtd_usr
Database Server                     : 142.148.44.62
Database Name                       : NSR
Database Schema                     : QTD_OWNR
Database Type, defaults to Oracle   : ORACLE
Test IDs to process(s)              : 7841
Submit all tests before result check    : YES
Action(s) seq to process            : 1
Calculate (test) data combination count : YES
```

```
============================================================
10:39:41 I Found 1 tests
        ************************************************************************
10:39:44 I Processing test case: 7841 match ID: 24FCOMV1PASS
        ************************************************************************
10:39:44 I Checking parameters for the test
10:51:29 I The total test count is: 4723920
        ----------------------------------------------------------------
10:51:50 I Checking test results for test [7841]
10:51:50 I Test is a SUCCESS
============================================================
>>>>>>>>>>>>>>>>>>>>>>>>>>>> TEST MANAGER RECAP<<<<<<<<<<<<
============================================================
Start Time:          10:39:30
End Time:            10:51:50
Processing Type:     Sequential
Test Tag:            20131227103930
Number of Tests:     1

Processed Test List:

1 ) 7841: NA                                  --------> SUCCESS
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

10:51:50 S *** fattRunTest.pl SUCCEEDED ***

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

The total test count is highlighted in yellow and it's 4,723,920. That's almost 5 million tests.

If we were to try and run this we would get:

11:11:50 I Found 1 tests

11:15:02 I The total test count is: 4723920

Out of memory!

This is one kind of out-of-memory error. It's the most obvious one and its advatantage is that it occurs before the real processing starts. Therefore the user doesn't waste too much time.

Another out-of-memory error can occur once the processing starts. As the program is building the XML document tree in memory it runs out of memory. That error may look like the below:

```
12:12:08 I Processing data combination # [204364]
12:12:08 I Processing data combination # [204365]
```

```
12:12:08 I Processing data combination # [204366]
12:12:09 I Processing data combination # [204367]
12:12:09 E ===> tree error : Memory allocation failed : creating buffer
        Entity: line 1: parser error : switching encoding: encoder error
        <SettlInstr xsi:schemaLocation="fsrv Settlement.xsd" xmlns="fsrv"
        xmlns:xsi="htt
                                    ^
        Entity: line 1: parser error : Blank needed here
        <SettlInstr xsi:schemaLocation="fsrv Settlement.xsd" xmlns="fsrv"
        xmlns:xsi="htt
                                    ^
        Entity: line 1: parser error : parsing XML declaration: '?>'
        expected
        <SettlInstr xsi:schemaLocation="fsrv Settlement.xsd" xmlns="fsrv"
        xmlns:xsi="htt
                                    ^
         <===
```

In this case the memory ran out during data processing at about 205K records. This is a worse case scenario because we had to wait for the processing to reach 205K records before we found out it failed.

In either case there is simply too many data combinations for this specific hardware / software setup we are using. To get to a more reasonable number we can set several fields to inactive status or lower the number of values in a field or two.

Very inviting fields for this are those that are big and mostly free form such as ‘DlrAcctID’, ‘FundAcctID’, ‘SrcID’. We can still test these fields either on their own or in combination with other fields later.

If we check the test count now using the same command (after deactivating three fields):

fatt.bsh -test 7841 -combo -action 1

we get:

The total test count is: 174960

This is a lot more reasonable number. We can try and run it to see whether we get the 'out of memory' error again.

Rule of thumb: current hardware and software supports about 200K data combinations, anything above that will likely cause out-of-memory error when actually run as a test.

Use of One Test File Or Many

But before we can proceed we need to consider one very important factor. We need to decide whether we want to stuff all the tests in a single xml file or whether we should use a file for each test / data combination.

It appears that we have a choice in this matter but in fact we don't at least not in this specific case. The reason is that NSR checks for uniqness of each file name submitted and if the file has already been processed the file will be rejected.

Here is the general rule for settlement file submission to NSR:

Based on the file name rules for settlement files NSR allows about 46 thousand unique file names per fund company, dealer and business day. After these unique file names have been used for the day all subsequent file submissions reusing a previously submitted file name will be rejected by NSR.

Since we have over 174 thousand tests (which is almost 4 times the 46K limit) we cannot use a test per file submission but must use a single file with all the tests inside it.

To do that we need to set the TEST_ACTION.ONE_FILE_FLAG = 'Y'.

Another thing to consider when deciding on the type of input we want (one file or many) is the output we are expecting. A single file input produces a single file output if a file is produced at all whereas multiple file input produces multiple file output if there is any output to be produced.

The consequence of this is that for positive tests, it doesn't much matter whether we use a single or multiple files as we don't expect an output, the odd failure we may get can be checked out manually in the error file.

This is different for negative testing. In negative testing we are looking for an output error file and it makes a lot of difference whether we get a single file or multiple files (a file per test).

With multiple files we only check for the existence of the error file, we don't need to read the file itself. With a single file we need to read the file and look for specific record.

Rule of thumb: When deciding whether to use single or multiple files for test submission we need to consider the results. In positive testing we don't need to track test records because we don't need to read the output file. In negative testing we need to read the output file and therefore we need to track the test records. We also need to setup additional output record(s) to check for

specific test records in the output file.

Action Records

Alltogether we have 2 action records set up for the test and they look like this:

Table 1 Create settlement file action record for mutiple data combinations

TEST_STEP_ID	350
MATCH_ID_RX	24FCOMV1PASS
STATUS	A
BUSINESS_PROCESS_ID	SettlRec_VALIDATION
FILENAME	$DEFAULT_FILENAME
STEP_ELEMENT	file
OUT_DIRECTORY	$YYY_STAGING_DIR
IN_DIRECTORY	$SOURCE_DIR
ONE_FILE_FLAG	Y
STEP_SEQ	1
DESCRIPTION	create settlement file
STEP_ACTION	create
DATA_TEMPLATE	settTest.xml

Table 2 Submit settlement file for processing action record

TEST_STEP_ID	351
STATUS	A
BUSINESS_PROCESS_ID	SettlRec_VALIDATION
FILENAME	$DEFAULT_FILENAME
STEP_ELEMENT	file
OUT_DIRECTORY	$YYY_IN_DIR
IN_DIRECTORY	$YYY_STAGING_DIR
STEP_SEQ	2
DESCRIPTION	submit settlement file to NSR for processing
STEP_ACTION	submit

Now we need to think about the output. Even though we are creating close to 175K tests which we will submit in a single file we are expecting all tests to pass and therefore no error file should be created. If an error file were to be created meaning that 1 or more tests failed we can see which tests failed by looking into the error file. It lists all the failed test records.

Output Record

We define the output with the below record:

STEP_SEQ	1
OUTPUT_ACTION	not exists
OUTPUT_VALUE_RX	s/^FST/FVT/
STATUS	A
OUTPUT_TYPE	file
OUT_DIRECTORY	$YYY_OUT_DIR
OUTPUT_FIELD	$DEFAULT_FILENAME
MATCH_ID_RX	24FCOMV.*PASS

This 1 record is good for the results of all 175K of test records we expect to create and submit.

Everything seems to be setup now and we can run the test with this command:

fatt.bsh -test 7841 --checkSubmit

In this case we cannot submit individual files because we would run out of unique file names, but if were to submit less files than the 45K limit we could use single files but the NSR processing could probably run for a couple of hours.

Rule of thumb: When setting up tests always consider whether to use a single file or multiple files per submission; single file is processed faster but multiple files with a file per test are more logical and easier to understand and track. Most importantly consider the output implications as mentioned above.

If we look at the below output we see that the the processing started at about 11:53, the file was created and submitted at about 12:19 so it took about 26 minutes to create 175K test records and put them into a single file.

After the next 20 minutes the error file was not found and therefore the test was deemed successful.

```
12:19:36 I Processing data combination # [174960]
12:19:42 I Created test file:
        [/data/fsrv/ffs/mgmt/YYY/bak/staging/FST7V62400.122713YYY9.Z01]
12:19:56 I Submitting test for processing
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
12:19:58 I Sleeping for [1200] secs before checking results
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ------------------------------------------------------------------------------------------------------
12:39:58 I Checking test results for test [7841]
```

```
12:39:58 I Test is a SUCCESS
============================================================
>>>>>>>>>>>>>>>>>>>>>>>>>>>> TEST MANAGER RECAP <<<<<<<<<<<
============================================================
Start Time:          11:53:38
End Time:            12:39:58
Processing Type:     Sequential
Test Tag:            20131227115338
Number of Tests:     1

Processed Test List:

1 ) 7841: /data/fsrv/ffs/mgmt/YYY/in/FST7V62400.122713YYY9... --------> SUCCESS
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

12:39:58 S *** fattRunTest.pl SUCCEEDED ***

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

We still need to test the remaining three fields which we set to inactive earlier. We can test them together, separately or in combination with other fields as long as the the test count is below 200K tests.

If we are doing a manual testing we can just set the ‘inactive’ records to ‘active’ and vice versa and run the test. However if we wanted to perform the tests automatically we could create a second test case and split the test fields between these 2 test cases so that each field is tested in combination with some other fields.

There are many different ways the testing can be setup so that all fields are tested but no single multi-value test combination causes ‘out of memory’ error.

5.2.2. NEGATIVE Multiple Value Element / Field Testing

Test Case ID: 7861,7881,7901,7902,7903,7904,7905,7906,7907,7908,7909,7910,7911

The test cases listed above are complimentary to the test case # 7841, these are the ‘negative’ value test cases.

Again, ‘negative’ values are values not allowed by the application for the specific fields the values are applied to.

Negative value testing is conceptually different from positive value testing. In positive testing any set of positive values must result in positive processing, i.e., we can be sure that no matter how many fields and values we had used in a single file they are all valid

if the file wasn't rejected.

In negative testing a single invalid value causes a file rejection regardless of how many other values there are positive or negative. Adding a positive or negative value to the test would not make a difference to the outcome once a negative value is already part of the test. If we have multiple test values submitted (using different fields) we can never be sure which value caused the failure (because the failed field/element is not specified in the error file).

Therefore the best way to test for negative values is to test them one-by-one on a field-by-field basis rather than multiple fields and values in a single test as is the case with positive testing.

What this means in FATT is that a negative test should have a single data field with multiple all negative values defined per test case.

The test will be successful when all test records fail processing, any single successful record will cause the overall test to fail.

The below table lists the test case id, the tested field name and the list of tested values. The test data looks like garbage and it is but that's fine because we are testing for negative value.

7861	**OrdType**	A,1,-,01,-1,w,DW
7881	**AcctType**	00,21,100,0,-5,#,A,d,Ac1,AAA
7901	**OrdSrc**	1,0,20,-1,d,DF,i2,DD
7902	**AcctDesig**	0,4,11,A,W,001,-1,aaa
7903	**CreateDate**	20140244,00181302,20151313
7904	**OrdStatus**	a,r,AR,-1,-,2,P
7905	**RevnNum**	0,0123,0001,-234,0A
7906	**DlrCode**	12345,11111111111111,AA 12,A1-^@
7907	**OrdID**	1234,AAAA,12345678,123 a 1!,0,-234,3+45678
7908	**DlrAcctID**	-12 lsdjfkhlhfldsfsa,)ALLL(*&^_MKAIJDHYT63291,0123456678gksajgfks,JHGFJHGDF#$%^*(
7909	**FundAcctID**	-12 lsdjfkhlhfldsfsa,)ALLL(*&^_MKAIJDHYT63291,0123456678gksajgfks,JHGFJHGDF#$%^*(
7910	**SrcID**	-12 lsdjfkhlhfldsfsa,)ALLL(*&^_MKAIJDHYT63291,0123456678gksajgfks,JHGFJHGDF#$%^*(
7911	**RepCode**	-0,)(*&^%$,-,M K

Unlike in the positive testing situation we don't have too many tests here and so we don't need to stuff all the test cases in 1 file to save the filenames for other testing which forces us to read the ouput file and search for specific tests we submitted.

In general it's preferable to use a file per test (test case) as opposed to one file for many tests because it's just easier and more logical to have one-to-one relationship instead of one-to- many.

Therefore in this case we need to setup a new action which will have the ONE_FILE_FLAG set to N.

The test action looks like this:

TEST_STEP_ID	384
MATCH_ID_RX	Undef
STATUS	A
BUSINESS_PROCESS_ID	SettlRec_VALIDATION
FILENAME	$DEFAULT_FILENAME
STEP_ELEMENT	File
OUT_DIRECTORY	$YYY_STAGING_DIR
IN_DIRECTORY	$SOURCE_DIR
ONE_FILE_FLAG	N
STEP_SEQ	1
DESCRIPTION	create settlement file
STEP_ACTION	Create
DATA_TEMPLATE	settTest.xml

We can run these tests with a command below for example:

fatt.bsh -match 24FCOMV%FAIL –check –submit

(this command would include the test # 7912 which is actually a combined test of positive and negative values as described below. In order not to include this test in the above, the test 7912 status was set to inactive)

And the summary of the run may look like this:

```
Start Time:          11:39:41
End Time:            11:43:05
Processing Type:     Sequential
Test Tag:            20140102113941
Number of Tests:     75

Processed Test List:

1 ) /data/fsrv/ffs/mgmt/YYY/in/FSTLHT2400.010214YYY9.Z01      --------> SUCCESS
2 ) /data/fsrv/ffs/mgmt/YYY/in/FSTLHU2400.010214YYY9.Z01      --------> SUCCESS
```

3) /data/fsrv/ffs/mgmt/YYY/in/FSTLHV2400.010214YYY9.Z01 --------> SUCCESS
4) /data/fsrv/ffs/mgmt/YYY/in/FSTLHW2400.010214YYY9.Z01 --------> SUCCESS
5) /data/fsrv/ffs/mgmt/YYY/in/FSTLHX2400.010214YYY9.Z01 --------> SUCCESS
6) /data/fsrv/ffs/mgmt/YYY/in/FSTLHY2400.010214YYY9.Z01 --------> SUCCESS
7) /data/fsrv/ffs/mgmt/YYY/in/FSTLHZ2400.010214YYY9.Z01 --------> SUCCESS
8) /data/fsrv/ffs/mgmt/YYY/in/FSTLI02400.010214YYY9.Z01 --------> SUCCESS
9) /data/fsrv/ffs/mgmt/YYY/in/FSTLI12400.010214YYY9.Z01 --------> SUCCESS
10) /data/fsrv/ffs/mgmt/YYY/in/FSTLI22400.010214YYY9.Z01 --------> SUCCESS
11) /data/fsrv/ffs/mgmt/YYY/in/FSTLI32400.010214YYY9.Z01 --------> SUCCESS
12) /data/fsrv/ffs/mgmt/YYY/in/FSTLI42400.010214YYY9.Z01 --------> SUCCESS
13) /data/fsrv/ffs/mgmt/YYY/in/FSTLI52400.010214YYY9.Z01 --------> SUCCESS
14) /data/fsrv/ffs/mgmt/YYY/in/FSTLI62400.010214YYY9.Z01 --------> SUCCESS
15) /data/fsrv/ffs/mgmt/YYY/in/FSTLI72400.010214YYY9.Z01 --------> SUCCESS
16) /data/fsrv/ffs/mgmt/YYY/in/FSTLI82400.010214YYY9.Z01 --------> SUCCESS
17) /data/fsrv/ffs/mgmt/YYY/in/FSTLI92400.010214YYY9.Z01 --------> SUCCESS
18) /data/fsrv/ffs/mgmt/YYY/in/FSTLIA2400.010214YYY9.Z01 --------> SUCCESS
19) /data/fsrv/ffs/mgmt/YYY/in/FSTLIB2400.010214YYY9.Z01 --------> SUCCESS
20) /data/fsrv/ffs/mgmt/YYY/in/FSTLIC2400.010214YYY9.Z01 --------> SUCCESS
21) /data/fsrv/ffs/mgmt/YYY/in/FSTLID2400.010214YYY9.Z01 --------> SUCCESS
22) /data/fsrv/ffs/mgmt/YYY/in/FSTLIE2400.010214YYY9.Z01 --------> SUCCESS
23) /data/fsrv/ffs/mgmt/YYY/in/FSTLIF2400.010214YYY9.Z01 --------> SUCCESS
24) /data/fsrv/ffs/mgmt/YYY/in/FSTLIG2400.010214YYY9.Z01 --------> SUCCESS
25) /data/fsrv/ffs/mgmt/YYY/in/FSTLIH2400.010214YYY9.Z01 --------> SUCCESS
26) /data/fsrv/ffs/mgmt/YYY/in/FSTLII2400.010214YYY9.Z01 --------> SUCCESS
27) /data/fsrv/ffs/mgmt/YYY/in/FSTLIJ2400.010214YYY9.Z01 --------> SUCCESS
28) /data/fsrv/ffs/mgmt/YYY/in/FSTLIK2400.010214YYY9.Z01 --------> SUCCESS
29) /data/fsrv/ffs/mgmt/YYY/in/FSTLIL2400.010214YYY9.Z01 --------> SUCCESS
30) /data/fsrv/ffs/mgmt/YYY/in/FSTLIM2400.010214YYY9.Z01 --------> SUCCESS
31) /data/fsrv/ffs/mgmt/YYY/in/FSTLIN2400.010214YYY9.Z01 --------> SUCCESS
32) /data/fsrv/ffs/mgmt/YYY/in/FSTLIO2400.010214YYY9.Z01 --------> SUCCESS
33) /data/fsrv/ffs/mgmt/YYY/in/FSTLIP2400.010214YYY9.Z01 --------> SUCCESS
34) /data/fsrv/ffs/mgmt/YYY/in/FSTLIQ2400.010214YYY9.Z01 --------> SUCCESS
35) /data/fsrv/ffs/mgmt/YYY/in/FSTLIR2400.010214YYY9.Z01 --------> SUCCESS
36) /data/fsrv/ffs/mgmt/YYY/in/FSTLIS2400.010214YYY9.Z01 --------> SUCCESS
37) /data/fsrv/ffs/mgmt/YYY/in/FSTLIT2400.010214YYY9.Z01 --------> SUCCESS
38) /data/fsrv/ffs/mgmt/YYY/in/FSTLIU2400.010214YYY9.Z01 --------> SUCCESS
39) /data/fsrv/ffs/mgmt/YYY/in/FSTLIV2400.010214YYY9.Z01 --------> SUCCESS
40) /data/fsrv/ffs/mgmt/YYY/in/FSTLIW2400.010214YYY9.Z01 --------> SUCCESS
41) /data/fsrv/ffs/mgmt/YYY/in/FSTLIX2400.010214YYY9.Z01 --------> SUCCESS
42) /data/fsrv/ffs/mgmt/YYY/in/FSTLIY2400.010214YYY9.Z01 --------> SUCCESS
43) /data/fsrv/ffs/mgmt/YYY/in/FSTLIZ2400.010214YYY9.Z01 --------> SUCCESS
44) /data/fsrv/ffs/mgmt/YYY/in/FSTLJ02400.010214YYY9.Z01 --------> SUCCESS
45) /data/fsrv/ffs/mgmt/YYY/in/FSTLJ12400.010214YYY9.Z01 --------> SUCCESS
46) /data/fsrv/ffs/mgmt/YYY/in/FSTLJ22400.010214YYY9.Z01 --------> SUCCESS
47) /data/fsrv/ffs/mgmt/YYY/in/FSTLJ32400.010214YYY9.Z01 --------> SUCCESS
48) /data/fsrv/ffs/mgmt/YYY/in/FSTLJ42400.010214YYY9.Z01 --------> SUCCESS

```
49) /data/fsrv/ffs/mgmt/YYY/in/FSTLJ52400.010214YYY9.Z01      --------> SUCCESS
50) /data/fsrv/ffs/mgmt/YYY/in/FSTLJ62400.010214YYY9.Z01      --------> SUCCESS
51) /data/fsrv/ffs/mgmt/YYY/in/FSTLJ72400.010214YYY9.Z01      --------> SUCCESS
52) /data/fsrv/ffs/mgmt/YYY/in/FSTLJ82400.010214YYY9.Z01      --------> SUCCESS
53) /data/fsrv/ffs/mgmt/YYY/in/FSTLJ92400.010214YYY9.Z01      --------> SUCCESS
54) /data/fsrv/ffs/mgmt/YYY/in/FSTLJA2400.010214YYY9.Z01      --------> SUCCESS
55) /data/fsrv/ffs/mgmt/YYY/in/FSTLJB2400.010214YYY9.Z01      --------> SUCCESS
56) /data/fsrv/ffs/mgmt/YYY/in/FSTLJC2400.010214YYY9.Z01      --------> SUCCESS
57) /data/fsrv/ffs/mgmt/YYY/in/FSTLJD2400.010214YYY9.Z01      --------> SUCCESS
58) /data/fsrv/ffs/mgmt/YYY/in/FSTLJE2400.010214YYY9.Z01      --------> SUCCESS
59) /data/fsrv/ffs/mgmt/YYY/in/FSTLJF2400.010214YYY9.Z01      --------> SUCCESS
60) /data/fsrv/ffs/mgmt/YYY/in/FSTLJG2400.010214YYY9.Z01      --------> SUCCESS
61) /data/fsrv/ffs/mgmt/YYY/in/FSTLJH2400.010214YYY9.Z01      --------> SUCCESS
62) /data/fsrv/ffs/mgmt/YYY/in/FSTLJI2400.010214YYY9.Z01      --------> SUCCESS
63) /data/fsrv/ffs/mgmt/YYY/in/FSTLJJ2400.010214YYY9.Z01      --------> SUCCESS
64) /data/fsrv/ffs/mgmt/YYY/in/FSTLJK2400.010214YYY9.Z01      --------> SUCCESS
65) /data/fsrv/ffs/mgmt/YYY/in/FSTLJL2400.010214YYY9.Z01      --------> SUCCESS
66) /data/fsrv/ffs/mgmt/YYY/in/FSTLJM2400.010214YYY9.Z01      --------> SUCCESS
67) /data/fsrv/ffs/mgmt/YYY/in/FSTLJN2400.010214YYY9.Z01      --------> SUCCESS
68) /data/fsrv/ffs/mgmt/YYY/in/FSTLJO2400.010214YYY9.Z01      --------> SUCCESS
69) /data/fsrv/ffs/mgmt/YYY/in/FSTLJP2400.010214YYY9.Z01      --------> SUCCESS
70) /data/fsrv/ffs/mgmt/YYY/in/FSTLJQ2400.010214YYY9.Z01      --------> SUCCESS
71) /data/fsrv/ffs/mgmt/YYY/in/FSTLJR2400.010214YYY9.Z01      --------> SUCCESS
72) /data/fsrv/ffs/mgmt/YYY/in/FSTLJS2400.010214YYY9.Z01      --------> SUCCESS
73) /data/fsrv/ffs/mgmt/YYY/in/FSTLJT2400.010214YYY9.Z01      --------> SUCCESS
74) /data/fsrv/ffs/mgmt/YYY/in/FSTLJU2400.010214YYY9.Z01      --------> SUCCESS
75) /data/fsrv/ffs/mgmt/YYY/in/FSTLJV2400.010214YYY9.Z01      --------> SUCCESS
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

11:43:05 S *** fattRunTest.pl SUCCEEDED ***

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

5.2.3. COMBINED Multi-field And Multi-value Testing

To some users it may seem that negative testing in isolation where each field and each value is tested on their own may not seem to be sufficient especially when compared to the positive testing method which tests all possible value combination defined and in all or at least many fields at the same time.

We can combine negative testing with multi-field multi-value positive testing in a test case such as test case # 7912

Test Case ID: 7912

.

We copy the data from the positive data combination test # 7841 to the new test. These are all positive values. Now we can take 1 of those fields, say 'OrdType' and change the value list to 'A,C,D' so that only 'D' is left as positive value, the other two are negative values i.e. invalid in this field.

Since we know we have a negative value in one field out of three values in that field and we also know that all the other values are positive in all the other fields we can project that 1/3 of all data combinations will fail processing. If we had four values in the OrdType field and one was a negative we could expect 25% of tests to fail, 2 out of 4 would produce 50% of failures and so on.

We should do this type of testing with only a single field carrying negative value or values so that our expected failure count remains simple to calculate and can be expressed as:

(**Number of negative values in the field) /**
(Total number of values in the field) *
(Total number of data combinations)

Now when we know how many failures in processing we can expect we can run the tests with all data combinations and check whether the number of failed records is indeed the percentage of the total it should be. If it is we can be fairly confident that the tested system is working as expected.

We cannot be 100% sure becase we would actually have to check each and every failed record to see that it actually contained the negative value. This is also possible in FATT now.

One File Or Many In Multi-field and Multi-value Combined Testing

We need to decide again how we are going to submit test records in one file or in many.

First consideration is how many test records are we going to have. We know that the rule is 46K unique contract files per fund company, dealer and day. If the record count is anywhere over 46K or even near 46K we have no choice but to use a single file. If the expected record count is significantly less we can use a file per record but we probably shouldn't. Day is long and you never know how many test you are going to run. To stay on the safe side and not use up all the file names for the day it's better go with a single file in this case.

Action

If we are using a single file to submit all records then we can reuse the action to create the

contract single file from the test case 7841 which is multi-value positive test case for SettlRec element.

We just update the MATCH_ID_RX field to include our test MATCH_ID so that the field value looks like this:

24FCOMV1PASS|24FCOMV15FAIL

Now this action will cover our test case. Optionally we can create a new test action for this test case if we don't know how to change MATCH_ID_RX to cover our test case, it makes no difference to the system.

To specify a list of MATCH_ID s in a MATCH_ID_RX field separate MATCH_ID values by a pipe like this 'valueA|valueB|valueC'.

Output

This would seem a difficult case to set up as we have 2 contradictory outputs. Some tests are positive and therefore shouldn't produce an output while the rest are negative and should in fact must produce an output (V file).

The way to approach this problem is to use the record count when checking the output. First we we need to disregard the positive tests results or to put it more accurately we will assume it.

We will only check for negative results, the processing failures. This is the exact same thing as we did above with the negative test cases. We are basically looking for an output file in the 'YYY_OUT_DIR' with its original file name changed to a slightly different name where the second character in the name is changed to 'V' (using the regex s/^FST/FVT/).

However we are not done yet, the second step is to check the failed records. Since the output is positive and negative combined we actually do need to check for specific record failures. It's not enough just to check that an error file was created by the tested system because the error file can be created for a single failed record out of 100K records.

Therefore in combined testing we actually need to look into the output file and see what's in it and in some way measure it.

There are 2 ways to go about this.

Result Check By Record Count

We can just check the failed record count. The count is usually sufficient enough for our purposes to be able to tell whether the tested system is working as expected.

We can't be 100% sure that the failures are the failures we expected, for that we would need to examnine each and every record. But if we have say 8,800 test cases, we expect 2,200 of them to

fail and indeed exactly 2,200 does fail it's a pretty good bet those are the records that should have failed.

There is a field 'RECORD_COUNT' in the TEST_OUTPUT table. This field allows us to specify the exact number of failures we expect.

Initially we might be tempted to reuse the existing output record from the failed tests 7861 etc. However now we see that we can't do that because we need to specify the RECORD_COUNT value which only applies to this test case and not to any of the negative tests.

Therefore we must create a new output record for this combined test case. We can copy the record from the negative test cases and make the changes we need.

To calculate the record count we will run the program using 'combo' option to get the total number of data combinations, then we will decide how many negative values we are going to use and in which fields.

Before we get to the calculations we have to think about what we are going to calculate in general terms.

- **Our setup is such that most of our values are positive except for a very few negative values.**
- **However in our output we are looking for failed processing e.i. negative tests only and are assuming successful processing which means we are not going to check for success as there really is nothing to check for (no output).**
- **Therefore our testing result (positive output by FATT) depends on the negative output of the tested system.**
- **In other words a failure in the tested system is a success in FATT and if the numbers match we are good.**

With this in mind we can proceed to the calculation.

Calculation

Let's assume the test data to be this (for test # 7912; all fields hold list data separated by a comma):

AcctType	01,20
DlrCode	1234,ZZZZ,1Q2W,MTXS
RepCode	0,Z0,9IUHN,02309,DDESA
OrdSrc	D,F,I

AcctDesig	1,2,3
RevnNum	40,01,99
OrdType	A,C,D
MgmtCode	YYY
CreateDate	20140101,99991212,10000202

If we run 'fatt.bsh –test 7912 –combo –action 1 –nosubmit' we get the total number of data combinations as 9,720.

In the above data there are only 2 negative values and both are in the OrdType field; they are 'A' and 'C' which is 2/3 of the values for the field. According to our formula we can expect:

9,720 * 2/3 = 6,480

6,480 failed records in the output file.

If we wanted to be more adventurous and make our life more difficult we could enter more negative values and recalculate the expected number of failures.

The calculation would get more complicated and it would go like this; **for each additional field we would calculate the number of failed records from the remainder of the positive tests left. Then we would add this to the previous total of failed (negative) tests**.

Here is an example:

1) Change 'AcctDesig' to hold 1 negative value as '0' and 2 positive ones as '2,3' so that the field value looks like this '0,2,3'.
2) Calculate the remainder of positive tests from the previous value as:
   9,720 – 6,480 = 3,240
3) Calculate the number of negative tests by using 'AcctDesig' field list '0,2,3'. The negative value is 1/3 of all values so:
   1/3 * 3,240 = 1,080
4) Add the new negative test count to the previous value like this:
   6,480 + 1,080 = 7,560

7,560 is the new negative test count. Use this value to update the RECORD_COUNT field in the TEST_OUTPUT table.

The output record will look like this:

STEP_SEQ	1
OUTPUT_ACTION	Exists
OUTPUT_VALUE_RX	s/^FST/FVT/
STATUS	A
OUTPUT_TYPE	File

OUT_DIRECTORY	$YYY_OUT_DIR
OUTPUT_FIELD	$DEFAULT_FILENAME
MATCH_ID_RX	24FCOMV15FAIL
RECORD_COUNT	7560
BUSINESS_PROCESS_ID	SettlRec_VALIDATION

We set MATCH_ID_RX to the match_id of the test (24FCOMV15FAIL) and RECORD_COUNT to 7560.

There is a small problem with the above setup as our test case will match 2 output records. The one above as well as the output record for the negative tests (7841,7901 etc.) which has a MATCH_ID_RX = '24FCOMV.*FAIL'.

This is not a big problem in this specific case. It's just that our result will be checked twice for the same thing, the existinece of error file.

However if we wanted to exclude test # 7912 from being matched to two output records we could update the MATCH_ID_RX field of the negative test output record to:

24FCOMV([2-9]|1^[5])FAIL

The brackets say to regex engine to exclude regex data with the value of '15' after '24FCOMV'.

Now everything is just right, the negative tests have 1 output record and the combined test has 2 unique output records instead of 2 duplicates.

Result Check By Record Check

If we want to go beyong the simple failed record check described above and want to actually check the failed data to make sure it contains at least 1 negative value we can do so.

The generic result check with failed record check goes as follows:

1) Check if output (error) file exists
2) OPTIONAL: Read the output file and search for each record submitted for testing
3) If a record is found check that it contains the value(s) that caused the failure

As you can see above the second step is optional; what this suggests is that there are again 2 ways of doing this type of testing.

Quick & Simple Record Check

The quick and simple way is to check all output records one by one to make sure each contains one of the negative values.

We need to think about how this should be done. We know we have 3 values where any one of them causes processing failure. So we need to check whether the record contains 'value1 OR value2 OR value3'. However there is a problem.

FATT currently doesn't support ALTERNATIVE OUTPUT. This means we cannot set up an 'OR' relationship among the output records.

FATT only supports 'AND' relationships within the output. That means all the output conditions must be true for the result to be a 'SUCCESS' in FATT.

To three alternative data pieces checked and get around this limitation we will create a single record and enter the data as a regex. We will leave OUTPUT_FIELD undefined and this will cause FATT to convert the xml record into a string which we can then match against this regex:

```
/<OrdType>A|<OrdType>C|<AcctDesig>0/
```

FATT knows this is a regex because it's surrounded by slashes, it includes the elements and values delimited by a pipe character which means these are 'OR' type of values, only one needs to exist in a record.

STEP_SEQ	2
OUTPUT_ACTION	=
OUTPUT_VALUE_RX	/<OrdType>A|<OrdType>C|<AcctDesig>0/ – *Use the $SrcID variable for the record as the value*
STATUS	A
WAIT_FOR	Undef
BUSINESS_PROCESS_ID	SettlRec_VALIDATION
OUTPUT_TYPE	file data – *Check the data in the output file*
TEST_OUTPUT_ID	861
MATCH_ID_RX	24FCOMV15FAIL
RECORD_COUNT	7560

We can also use the 'RECORD_COUNT' field and set it to '7560'. If we do we can set this field to null in the output record that checks for the V file as set up above so that we don't check the record count twice.

To run the test we can type in:

fatt.bsh -test 7912 –check –submitall

This is is a very fast way of confirming that the failed records are those that should have failed and no other.

The disadvantage is that it's possible though very unlikely that 1 or more records in the file are

not the records we submitted.

To solve this problem we can do a more thorough record check.

Thorough & Slow Record Check

Thorough and slow record check is done by first finding a record we submitted in the output file and then performing the data check on this record.

To do that we need to keep track of the unique identifiers of the records. FATT won't do that on its own, it must be told to do that as well as which field to use as the indetifier. The option for that is 'recordId' and it has a default value of 'SOURCE_ID' as the field to be used for record tracking.

The reason for this (the fact that 'recordId' option must be used for record tracking) is that FATT gets the output records only when it's ready to check for results. By then it's too late to find out that the record tracking was necessary for the proper output handling. The records had already been released by FATT and there is no way of getting them back.

In addition FATT uses a single SOURCE_ID for all subtests as SOURCE_ID is only refreshed per test. By using the option 'recordId' we tell FATT to get a new SOURCE_ID for each subtest because we need to use this value as the primary key for our test collection and record tracking.

This functionality may change in the future but for now the user has to instruct FATT to keep track of test records for the purpose of result checking.

Once FATT has kept the test records it can be instructed to check the output for the records submitted.

This is done by creating 2 output records (in addition to the file output record which looks almost exactly like the one shown above) in the TEST_OUTPUT.

First record is the identifier record. It tells FATT to look for an output record in the file using SrcID field it collected during record creation.

STEP_SEQ	2 – *IMPORTANT, check for record existence as the second step in result chek*
OUTPUT_ACTION	=
OUTPUT_VALUE_RX	$SrcID – *Use the $SrcID variable for the record as the value*
STATUS	A
WAIT_FOR	Undef
BUSINESS_PROCESS_ID	SettlRec_VALIDATION
OUTPUT_TYPE	file data – *Check the data in the output file*
TEST_OUTPUT_ID	821

DATA_SOURCE_NAME	Undef
OUT_DIRECTORY	Undef
TEST_CASE_ID_REF_FIELD	SrcID – *This tells FATT that SrcID is the primary key for output records*
OUTPUT_FIELD	SrcID – *Use SrcID field as the output field to look for*
MATCH_ID_RX	24FCOMV15FAIL
UPDATED_AT	Undef
RECORD_COUNT	7560

If FATT finds the record in the output file using the above definition it needs to check that the record actually contains the negative value that caused its failure. For that we need to setup another output record.

This is essentially the same record as we created above for the 'Quick and Simple Record Check'. The only difference is the 'STEP_SEQ' number which is now 3 because the # 2 spot is taken by the record search.

STEP_SEQ	3 – *IMPORTANT, check for specific value(s) as the third step in result chek (if file exists and record was found).*
OUTPUT_ACTION	=
OUTPUT_VALUE_RX	/<OrdType>A|<OrdType>C|<AcctDesig>0/ – *Use the $SrcID variable for the record as the value*
STATUS	A
WAIT_FOR	Undef
BUSINESS_PROCESS_ID	SettlRec_VALIDATION
OUTPUT_TYPE	file data – *Check the data in the output file*
TEST_OUTPUT_ID	861
MATCH_ID_RX	24FCOMV15FAIL
RECORD_COUNT	7560

Now we can be confident that the failed data is the data we submitted and not something that the tested system concocted on its own.

The disadvantage is that since we have to search for every single record in the xml file containg thousands of records the speed is rather slow. Compared to the Quick and Fast method this is orders of magnitude slower, in practical terms this specific test case will take 15-20 minutes longer than 'Quick & Simple' method.

Wide Output

In addition FATT will also display all the failed data combinations at the end of the run which allows us to eyeball the data to make sure the failed combos are those expected. This can be done by using 'fullDisplay' option.

We have to remember that the failed data in negative tests is actually the valid or positive data in the tested system (NSR), it's the data that didn't fail and therefore wasn't included in the error file.

In our case it means that each failed record must have 'OrdType = D' and 'AccetDesig = 2' or 'AccetDesig = 3'.

Since there is going to be 2,160 (9720 -7560) failed records they will not fit on the screen and so we cannot properly eye-ball the full output.

To get around this we can either redirect the output to an output log file and check the log file or we can specify an email address and FATT will send the results to this email where we can the check the output. The command to use would be:

fatt.bsh -test 7912 -check -recordid -fulldisp -emaillist russ.profant@fundserv.com

- *check* means wait until submitted file is deleted from the submit directory before checking for results
- *recordid* means that FATT needs to track records by SOURCE_ID/SrcID field (this is the default)
- *fulldisp* means for the recap to include full lines in RECAP so we can see all the data instead of shortend lines normally produced by FATT
- *emaillist* means to send the recap email to the address(es)

These are the last few lines of the output:

```
2159) SrcID=CPIE Date=20140116 RepCode=DDESA OrdSrc=I RevnNum=99 OrdType=D
MgmtCode=YYY CreateDate=10000202 DlrCode=MTXS AcctType=01 AcctDesig=3 --------> FAILURE

2160) SrcID=CPIG Date=20140116 RepCode=DDESA OrdSrc=I RevnNum=99 OrdType=D
MgmtCode=YYY CreateDate=10000202 DlrCode=MTXS AcctType=20 AcctDesig=2 --------> FAILURE

2161) SrcID=CPIH Date=20140116 RepCode=DDESA OrdSrc=I RevnNum=99 OrdType=D
MgmtCode=YYY CreateDate=10000202 DlrCode=MTXS AcctType=20 AcctDesig=3 --------> FAILURE

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


10:34:54 S *** fattRunTest.pl SUCCEEDED ***


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

The reason for the 2161 record count is that the first line is for the actual file whereas the rest of the lines list all the failed data combinations, all 2,160 of them.

5.3. Field /Element Testing Recap

There are **2 basic approaches** to field or element testing in FATT:

1) **Single element** and single field testing

   As the name says it's one field and one value per test case.

   Single-element/field testing requires hundreds to thousands of test cases but otherwise it's straightforward to set up.

   An example is the business process called 'CONTRACT_FIELD_VALID'. This process contains some 2,700 test cases that were created manually and migrated to FATT. It tests all major complex elements within a contract xml record.

   To run all the tests use the command:

   /app/fsrv/nsr/fatt/bin/fatt.bsh -business CONTRACT_FIELD_VALID -check -submit

2) **Multi-element and/or multi-value testing**

   This testing involves multiple fields and/or multiple values per test case.

   The advantage is that in only a handful of test cases we can test hundreds of thousands of data combinations. The data combinations are created automatically from the lists of values the user enters for each field.

   An example of this approach is the process id 'SettlRec_VALIDATION'.

   Within this process there are **three basic testing approaches**:

   i) **Positive data testing** where we test only allowed data values for each field

   In this case we include numerous fields in one test case and assign each field a list of allowed values. The program then creates all possible combinations from the available data.

   We have to watch for the number of data combinations and consequently tests. It shouldn't exceed 200K otherwise the system will run out of memory.

   We should also use a single file for all test submissions which makes testing fast and allows us to conserve unique file names if there is such a need.

   ii) **Negative data testing** where we test only the negative values in each test case.

   Negative data testing is different from positive testing. It should be done on a 1 field per

test case basis with a list of multiple values defined in order to have proper control over the testing outcome.

Testing more than a single negative value per test is not recommended as the outcome is ambiguous as to which value caused the processing failure.

iii) **Combined negative and positive data testing**

This testing type may be used as a complimentary to the previous 2 types. It allows us to test a combination of valid and invalid data. With this approach we have to know exactly which data is incorrect so that we can calculate the failure rate for testing. This allows us to decide the testing outcome, success or failure of a test. Additionally we can also check the failed data itself to confirm that it contains the invalid data items that caused the processing failure.

Checking just the failed record count is fast whereas checking the failed data itself has 2 additional possibilities.

Quick and easy check scans each failed record to confirm it contains at least 1 negative data piece. The scan only takes about a second for 10K of records.

Thorough and slow check requires record identifiers to be saved during the run and use them to check the output and scan the record for failed data. It is significantly slower depending on the size of the output file. If the file has less than 10K records it takes 10-15 minutes but the speed goes down significantly as the output file size grows. This needs to be kept in mind when setting up these types of test cases.

To run all these tests use the below command:

/app/fsrv/nsr/fatt/bin/fatt.bsh -business SettlRec_VALIDATION -check -submit

The output will be somewhat misleading as it will show each test file created for the negative testing as a separate test even though it's just a subtest of a single test case.

6. Business Process Testing

Unlike in element testing here we are going to test an actual business process rather just submit some data files. This is a lot more complex testing task than field/element testing.

We will start with a basic business process test which involves an order submission and a matching contract. This is essentially client process simulation in order to test FundSERV back end proceses, order acceptance and contract loading.

Client in this case is used in the business sense not in the technical sense such as client-server systems.

The client is not a single client but 2 different clients. The first one is a dealer who submits an order the second one is the fund company that receives the order and sends back the settlement instruction file.

6.1. Basic Business Process

This is the process we need to test:

1) Dealer sends an order to FundSERV

2) FundSERV accepts the order and forwards it to the appropriate fund company

3) The fund company accepts the order and replies with a settlement instruction for the order the next business day at the earliest.

This is a fairly simple process that we need to translate into FATT. Before we get there though we need to consider what exactly we can and should test.

One thing that seems to be outside thes scope of this test is the order forwarding from FundSERV to the fund company.

Since this is client testing, it's meant to be active client testing. In other words we want to simulate the client actions as they impact FundSERV not their passive acceptance of data.

Besides, to really test client data forwarding we would need the client participation but this is not in the scope for our test.

Hence we will leave out this part of the process. After this reduction we are left with the following process:

1) Dealer sends an order to FundSERV.

2) FundSERV accepts the order.

3) The fund company replies with a settlement instruction for the order the next business day at the earliest.

We will start with specific type of order namely BUY as this is already setup and we can use it as an example.

There is no real difference in terms of processing or process setup between 'buy' and a 'sell', the only difference is in the data we use during the processing.

6.2. Business Process Mapping to FATT

The mapping between the actual business process and FATT is not complicated but there is no one-to-one relationship either at least most of the time. This is because the step granularity in business is a lot courser than the step granularity in FATT. FATT usually needs more than one step to fulfill a single businesss step. And sometimes we need additional testing steps to simulate certain business conditions such as next business day.

Even a simple business step like file creation is in FATT (which can be an order or contract submission by the client) a 2-step process: 1) create the file in the staging directory 2) submit/copy the file to the processing directory.

Therefore the business step #1 is translated into 2 actual steps: 1) create order file 2) submit order for processing. The below tables show the test action records for file creation.

Create an order file action record:

TEST_STEP_ID	64
STEP_SEQ	1
STATUS	A
BUSINESS_PROCESS_ID	BUY
STEP_ELEMENT	File
STEP_ACTION	Create
FILENAME	$ORDER_FILENAME
OUT_DIRECTORY	$STAGING_DIR
IN_DIRECTORY	$SOURCE_DIR
DESCRIPTION	create TFS order file
DATA TEMPLATE	buyOrderTemplate.xml

Submit an order file for processing action record:

TEST_STEP_ID	106
STEP_SEQ	4

STATUS	A
BUSINESS_PROCESS_ID	BUY
STEP_ELEMENT	ftp
STEP_ACTION	put
IN_DIRECTORY	$STAGING_DIR
OUT_DIRECTORY	$ORDER_DIR
FILENAME	$ORDER_FILENAME
STEP_SCOPE	batch
REPO_NAME	FFS
DESCRIPTION	submit TFS order file

Next Business Day Problem

However there is a complication in this process and it's the fact that the settlement file can be accepted only the next business day at the earliest by FundSERV.

Now we have 2 options; we can either really wait till the next business day before continuing or we can try and find a way of simulating the next business day within the FundSERV environment.

FATT can accommodate both options but we will concern ourselves only with the second option for now; we need to find a way to simulate the next business day in FundSERV environment.

The business day in terms of fund processing is controlled by the cutoff time defined for each fund. The current business day is defined up until the cutoff time and anything after that is the next business day regardless of the actual current date.

Therefore to change FundSERV processing to next business day we need to update the fund cutoff time to be in the past vis-a-vis current time. To get back to the current day we need to change the fund cutoff time to be in the future relative to current time.

We will setup a test action to do that and we will reuse it a number of times. The main action field is the SQL statement that changes the cutoff time for a fund.

TEST_STEP_ID	105
MATCH_ID_RX	Undef
STATUS	A
BUSINESS_PROCESS_ID	BUY
SQL_LIST	UPDATE tfs_ownr.fund@qtd_tfs set cut_off_time = '$CUT_OFF_TIME ' WHERE mgmt_company_code = '$MgmtCode' AND fund_number = '$FundID'
STEP_ELEMENT	data
STEP_SCOPE	batch

ONE_FILE_FLAG	N
STEP_SEQ	2
DESCRIPTION	update fund cutoff time to future time
STEP_ACTION	update

As we can see this is a 'batch' type action since it's updating the fund it needs to run only once for all orders using the specific fund id (and MgmtCode).

It's not enough to just update the cutoff time however because NSR is caching static data including fund data for performance purposes. We need to force NSR to reload its static data otherwise NSR will never see the fund cutoff time change which means it will reject settlement files as being submitted the same business day.

The static data reload is done by a small shell script called 'refresh.bsh'. The content of the script is as follows:

#!/usr/bin/bash

rm -rf /data/fsrv/ffs/mgmt/REFRESH

sleep 10

mkdir /data/fsrv/ffs/mgmt/REFRESH

exit 0

In the script the 'sleep 10' command is very important as it was observed that NSR is not able to detect 'REFRESH' folder deletion and recreation with less than 3-5 seconds in-between the two actions. If NSR doesn't observe 'REFRESH' folder change it doesn't reload the data.

This task is wrapped up in another test step that is setup in FATT:

TEST_STEP_ID	106
MATCH_ID_RX	Undef
STATUS	A
BUSINESS_PROCESS_ID	BUY
COMMAND_LIST	./refresh.bsh
STEP_SCOPE	batch
STEP_SEQ	3
DESCRIPTION	Refresh NSR

But why did we need to setup fund cutoff time change at this point? This should be relevant only when submitting settlement files and not order files.

We do it because we don't know what the current cutoff time is as we are constantly changing it. So to be on the safe side we update the cutoff time for the fund we intend to use with our order before we actually submit the order to make sure the current business day is still current, in other

words the cutoff time hasn't been set to an earlier time today which would make today at this time already next business day (in FundSERV environment).

We are not going to go though each step in the test process in detail but we will look at the full process as currently setup.

Below is the full business process translated into testing steps within FATT:

1	**create** TFS order file	We use a file template and database data to create an order
2	**update** fund cutoff time to future time	We update the fund cutoff time to a future time today
3	**submit** TFS order file	We ftp order to the submission directory of FFS
4	**check** if order was created	We check the TFS to see whether the order was created
5	**transfer** order to NSR	If order was created we transfer it from TFS to NSR
6	**update** fund cutoff time to earlier time to make now the next business day within FundSERV	We update the fund cutoff time to the past time today to simulate the next business day
7	**refresh** NSR	We refresh the NSR so that it sees the cutoff time change
8	**create** settlement file	We create settlement file from a template and dynamic data from the database
9	**submit** settlement file to processing directory	We submit the settlement data to NSR for processing

The fill colors in the second column indicate the mapping of FATT test steps to the 3-step business processes described earlier:

1) Dealer creates the order file and submits it to FundSERV
2) FundSERV accepts order
3) Fund company creates settlement file and submits it to FundSERV

Output

Even though this is a more complex testing than the field testing the output is quite similar except in this case the output is not the 'V' file but rather the 'K' file.

'K' file is also an error file but it is a business error that the file indicates. Its name is derived the same way as the 'V' file name where the second character of the file name prefix 'FST' is replaced by 'K' to become 'FKT'.

But unlike the 'V' file which only carries the incorrect record, the 'K' file actually carries the error associated with the processing failure.

Therefore we not only need to check for the existence or lack of the 'K' file we also need to check the content of the file whether the error is the expected error or not.

The following is the output file for test cases we expect to pass which are the test cases that have their match id end in 'PASS'. The record sets up a K file that is not supposed to exist. If it doesn't the test case is a success, if it does the test case fails.

STEP_SEQ	1
OUTPUT_ACTION	not exists
OUTPUT_VALUE_RX	s/^FST/FKT/
STATUS	A
OUTPUT_TYPE	File
OUT_DIRECTORY	$OUT_DIR
OUTPUT_FIELD	$FS_FILENAME
MATCH_ID_RX	24BBEK.*PASS
BUSINESS_PROCESS_ID	BUY

The following is the ouput record for the processing failures when the 'K' file is created. This record is mapped to the test cases that have their MATCH_ID end in 'FAIL'.

STEP_SEQ	1
OUTPUT_ACTION	exists
OUTPUT_VALUE_RX	s/^FST/FKT/
STATUS	A
OUTPUT_TYPE	file
OUT_DIRECTORY	$OUT_DIR
OUTPUT_FIELD	$FS_FILENAME
MATCH_ID_RX	24BBEK.*FAIL
BUSINESS_PROCESS_ID	BUY

Additionally we need to create the error checking records. In FundSERV v24 the error consists of 2 separate records; en error code and an error detail. Hence for every expected processing failure we actually need to create 2 separate records.

For example for the test id # 7445 we have these 2 records setup:

STEP_SEQ	2
OUTPUT_ACTION	=
OUTPUT_VALUE_RX	805

STATUS	A
OUTPUT_TYPE	file data
OUTPUT_FIELD	ErrorCode
MATCH_ID_RX	24BBEK2FAIL
BUSINESS_PROCESS_ID	BUY

STEP_SEQ	3
OUTPUT_ACTION	=
OUTPUT_VALUE_RX	A
STATUS	A
OUTPUT_TYPE	file data
OUTPUT_FIELD	ErrorCodeDtl
MATCH_ID_RX	24BBEK2FAIL
BUSINESS_PROCESS_ID	BUY

In essence we are expecteding a failure with an error record that will have an element called ErrorCode = 805 and an element called ErrorCodeDtl = A. If these two elements are not found the test fails.

6.3. More Complex Business Process Mapping to FATT

In this chapter we will look at the most complex process currently set up in FATT. It's not much different than the BUY process dissected earlier but it does include additional level of complexity.

We will map In-Cash-Transfer (ICT) business process to FATT. The business process goes like this:

1. Dealer sends in an ICT order
2. FundSERV accepts the order and immediately creates a sell order for a 'from' fund id and forwards the order to the fund company
3. The fund company accepts the sell order and sends a contract file to FundSERV the next business day (at the earliest)
4. FundSERV accepts the contract
5. FundSERV creates a buy order the same business day the sell contract was accepted and sends it to the fund company
6. Fund comapy accepts the buy order and sends back to FundSERV the contract for it the next business day (at the earliest)

As we can see ICT is nothing more than a combination of 2 separate orders in a single order, a sell followed by a buy. It has twice as many bsuiness steps as a simple buy process outlined earlier.

The below is what the process looks like in FATT:

1	**create** TFS order file	We use a file template and database data to create an order
2	**update** fund cutoff time to future time	We update the fund cutoff time to a future time today
3	**submit** TFS order file	We ftp order to the submission directory of FFS
4	**check** if SELL order was created	We check the TFS to see whether the order was created. We are not checking the type of order only that an order was created.
5	**transfer** SELL order to NSR	If order was created we transfer it from TFS to NSR
6	**update** fund cutoff time to earlier time to make now the next business day within FundSERV	We update the fund cutoff time to the past time today to simulate the next business day
7	**refresh** NSR	We refresh the NSR so that it sees the cutoff time change
8	**create** SELL contract file	We create settlement file from a template and dynamic data from the database
9	**submit** SELL contract file to processing directory	We submit the settlement data to NSR for processing
10	**check** if SELL contract was created in NSR	We check if the contract was loaded by NSR
11	**generate BUY** order part of ICT in TFS	We run a FundSERV process to generate BUY orders from ICT orders
12	**transfer BUY**order to NSR from TFS	We transfer the BUY order to NSR
13	**create BUY** contract file	We create the BUY contract
14	**submit BUY** contract file to FFS	We submit the buy contract to NSR

Most of the above steps are the common ones that we have already seen previously. One notable exception is the step # 11.

This step uses the FundSERV process that generates ICT buy orders once the settlement contract for the SELL portion of the order has been loaded into NSR. This process is wrapped in a shell script called ‘generate.bsh’ in FATT. For more detailed explanation of what this program does contact the developer Steven Kong.

There is an already a working test case for this process with the number 7451. The rest of the ICT test cases for BUY testing are in ‘inactive state’. They need need to be properly setup (test case data corrected) and activated.

TO DO: Correct the ICT test cases (TEST_CASE.BUSINESS_PROCESS_ID = 'ICT') and activate them.

To run the sample test case type in:

fatt.bsh –test 7451 –check

The output should look like this:

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
14:43:16 I OS:      SOLARIS (command line run)
         SERVER:  qsz24nsr
         PROGRAM: ./fattRunTest.pl
         PROCESS: 8758
================================================================================

                    FATT GENERAL PARAMETERS

Configuration file path            : /app/fsrv/nsr/fatt/config/fatt.config
Config file entry                  : fatt_user
Database User                      : qtd_usr
Database Server                    : 142.148.44.62
Database Name                      : NSR
Database Schema                    : QTD_OWNR
Database Type, defaults to Oracle  : ORACLE
Test IDs to process(s)             : 7451
Submit all tests before result check : YES
Check if submitted file is gone    : 600

================================================================================
14:43:17 I Found 1 tests
14:43:17 I Getting data and actions for each test case; this may take a
         while...
         ********************************************************************
14:43:17 I Processing test case: 7451 match ID: 24BBEK8PASS
         ********************************************************************
14:43:17 I Checking parameters for the test
14:43:17 I Created test file:
         [/data/fsrv/ffs/mgmt/YYY/bak/staging/WOC3YYY9.B1142400]
14:43:17 I Processed SQL command: [UPDATE tfs_ownr.fund@qtd_tfs set
         cut_off_time = '144716' WHERE mgmt_company_code = 'YYY' AND
         fund_number = '720']
14:43:17 I Processed SQL command: [UPDATE tfs_ownr.fund@qtd_tfs set
         cut_off_time = '144716' WHERE mgmt_company_code = 'YYJ' AND
```

```
        fund_number = '610']
14:43:17 I Will upload these files to the processing server:
        /data/fsrv/ffs/mgmt/YYY/bak/staging/WOC3YYY9.B1142400
14:43:28 I Record FOUND in the table tfs_ownr.order_msg@qtd_tfs
14:43:28 I Processed SQL command: [@../db/order_load.sql YYY 20140211 1542361]
14:43:28 I Processed SQL command: [UPDATE tfs_ownr.fund@qtd_tfs set
        cut_off_time = '144316' WHERE mgmt_company_code = 'YYY' AND
        fund_number = '720']
14:43:28 I Processed SQL command: [UPDATE tfs_ownr.fund@qtd_tfs set
        cut_off_time = '144316' WHERE mgmt_company_code = 'YYJ' AND
        fund_number = '610']
14:43:28 I Running command [./refresh.bsh]
14:45:38 I Created test file:
        [/data/fsrv/ffs/mgmt/YYY/bak/staging/FSTHDM2400.021114YYY9.Z01]
14:45:38 I Submitting test for processing
14:46:43 I Record FOUND in the table NSR_OWNR.contract_master
14:46:43 I Running command [./generate.bsh]
14:47:20 I Processed SQL command: [@../db/order_load.sql YYJ 20140211 1542361]
14:47:20 I Created test file:
        [/data/fsrv/ffs/mgmt/YYY/bak/staging/FSTHDM2400.021114YYY9.Z01]
14:47:20 I Submitting test for processing
        ------------------------------------------------------------------
14:47:20 I Checking test results for test [7451]
14:47:50 I Test is a SUCCESS
================================================================================
>>>>>>>>>>>>>>>>>>>>>>>>>>>> TEST MANAGER RECAP <<<<<<<<<<<<<<<<<<<<<<<<
================================================================================
Start Time:          14:43:16
End Time:            14:47:50
Processing Type:     Sequential
Test Tag:            20140211144316
Number of Tests:     1

Processed Test List:

1 ) 7451: /data/fsrv/ffs/mgmt/YYY/in/FSTHDM2400.021114YYY9... --------> SUCCESS
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

14:47:50 S *** fattRunTest.pl SUCCEEDED ***

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

7. TEST DEVELOPMENT

In this section we will see how to develop test cases in FATT.

The business problem we will use to demonstrate this is the testing of business edits in a contract file.

Again we will look at a specific business process namely 'Buy', this is a subset of the business edit contract testing.

7.1. FATT As a Database Interface Tool

This functionality is not fully finished and contains numerous bugs (mostly minor) as there was not enough time to finish it properly.

FATT is not just a tool to run the predefined test cases. It's also a tool to develop those test cases.

Since test cases are always defined in the FATT database (in this context FATT database is just a logical name for QTD_OWNR table space residing on the NSR 24 server) for FATT to be a good development tool it needs to be able to interface with its database.

And it does that in all major database functional areas:

1) It can retrieve and display records
2) It can create records
3) It can update records
4) It can delete records

Before anybody gets the idea that FATT is a full-fledged Oracle client let us dispel that right now.

FATT can do all the basic database actions but nothing more. Its purpose is not to replace or replicate SQL Developer or even SQL*Plus but to enable and make really simple the basic database operations needed for test case setup and maintenance.

FATT can always be used only in single functional role at a time:

1) It can run test case(s)
2) It can perform one database operation

In FATT database actions take precedence over test case processing. The key database actions are designated by the parameter with the same name as the action followed by the table name such as:

fatt.bsh –**show** table

fatt.bsh –**create** table

fatt.bsh –**update** table

fatt.bsh –**delete** table

'table' is not the table name but the table name shortcut.

The table name shortcuts are designed to make their use easy (to remember). There are often several shortcut names available for a single table.

There is an '**all**' name which covers all tables in FATT db.

TABLE NAME	SHORTCUT NAME
TEST_CASE	test case
	case
TEST_CASE_DATA	test case data
	case data
	data
TEST_ACTION	action
BUSINESS_PROCESS	business
	process
TEST_PROCESSING_DATA	data
	processing data
TEST_OUTPUT	output
TEST	result
FS_ERROR_CODE	error
TEST_VARS	vars
	variables
	sys_vars
TEST_DATA	run data

Shortcut names consisting of 2 or more words need to be enclosed in quotes (double or single).

In addition to specifying the action and the table, the user also has to specify indetifier(s) to which the action applies. The full generic command looks like this:

fatt.bsh –action table –idType id

Show Command

The identifiers for the show command can be any of the identifiers for the test case execution:

1) testIdList
2) matchIdList
3) businessIdList

With 'show' parameter you cannot use 'id' parameter, you must use one of the above identifier parameters.

Below are some examples of the use of show command.

Show all business processes:

fatt.bsh –show business

Show test steps/actions for business process called 'BUY':

fatt.bsh –show action –bus BUY

Show all records set up for the test case id 5555 in all tables:

fatt.bsh –show all -test 5555

Show only specific test cases:

fatt.bsh –show case -test 5555,5556

Show test results based on a test tag:

fatt.bsh -show result -tag 20140213105819

Find out what actions will match a specific test case:

fatt.bsh –show action -test 5555

Find out what output will match a specific business process:

fatt.bsh –show output -bus BUY_CONTRACT

Show error codes:

fatt.bsh –show error

Show error code 815 and code detail A:

fatt.bsh –show error –err 815A

Create Command

THis command creates records in FATT database. As already mentioned a few times FATT doesn't allow direct record creation. It only supports record creation by copying existing records. The reason for this is to make record creation easier. Most of the time the user doesn't even need to supply a single field value to create a new record.

The generic command is:

fatt.bsh -create table –id XXX

There are 2 exceptions to this; the case and business records are created by specifying 'testIdList' and 'businessIdList' identifiers respectively.

Below are some examples:

Create a new test case by copying the test case id 5555:

fatt.bsh -create case -test 5555

This will create a new test case as well as new test case data (copied over from the test case id 5555:

```
================================================================================
09:35:05 I Found 1 tests
09:35:05 I Getting data and actions for each test case; this may take a
        while...
09:35:06 I Created new test with id [7943]
09:35:06 I Created new data record with ID [17253]
09:35:06 I Created new data record with ID [17254]
09:35:06 I Created new data record with ID [17255]
09:35:06 I Created new data record with ID [17256]
09:35:06 I Created new data record with ID [17257]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Create a new business process:

fatt.bsh -create business –bus BUY_CONTRACT

When we create a new business process FATT also creates related business actions:

```
==========================================================================
09:59:40 I Created test action(s) with business process id [BUY_CONTRACT_G1UT]
09:59:40 I Created new business process [BUY_CONTRACT_G1UT] with associated
        actions
```

To check the new business actions you can type in:

fatt.bsh –show action –bus BUY_CONTRACT_ G1UT

When creating new records the ‘create’ command can be combined with the ‘dataList’ parameter to create a new record with the specific field values from the ‘dataList’ parameter.

fatt.bsh –create business –bus BUY_CONTRACT –data “BUSINESS_PROCESS_ID:BUY_TEST”

To create a new output record by copying the record with the ouput id = 702

fatt.bsh –create output –id 702

To create a new test case data record by copying the record with the ouput id = 10270

fatt.bsh –create data -id 10270

To create new test case data records by copying test case data records from the test case id = 5555

fatt.bsh –create data –test 5555

Record creation for tables is not currently supported

Update Command

The update command has exactly the same format as the ‘create’ command. We mostly use ‘id’ parameter to indetify the records to update.

fatt.bsh -update case -test 5555 -data "DESCRIPTION:NULL;EXPECTED_RESULT:pass"

To set the field to null specify ‘NULL’ as the value.

Delete Command

Just like above, use ‘id’ to specify the record to delete.

Test case deletion using ‘testIdList’ will delete the test case as well as its related test case data.

Business process deletion will delete business process and the related business action when using the ‘businessIdList’.

The ‘create, update, delete’ database actions do not support list values in list type identifiers ‘testIdList’, ‘ matchIdList’ and ‘businessIdList’. If a list is given only the first id from the list will be used.

7.2. FATT As a Test Case Development Tool

FATT makes it easy to develop test cases. Although the recommended case development is by copying and updating existing test cases as we will see in the next chapter, it’s possible to develop test cases from scratch.

To make the test case development really easy and fast there are already a few predefined business processes in FATT database. These business processes are meant to be used for test case development but they are no different from regular business processes used to run already defined test cases. In fact except for their names (in some cases) you cannot tell them apart from “regular” business processes.

They are in fact business sub-processes of a larger business process. FATT however doesn’t have the concept of a subprocesses and therefore these subprocesses are treated and look like any other business process.

There is one important difference between running a predefined test cases through a business process and using a business process to develop such a test case and it’s the ‘adHoc’ parameter.

To run the FATT tool without a test case the ‘adHoc’ parameter must be specified on the command line.

‘adHoc’ tells FATT not to look for any test cases but perform the business process with default

or dummy test case. The dummy test case has the TEST_CASE_ID of '-1'.

Even though there is no test case and therefore no test case data there is system data and processing data still available during the run. This is very important and will be explained a little later.

There is no need or even a point in setting up output for 'adHoc' testing.

7.2.1. Order Submission Development

We don't have a generic order submission process in FATT at least not yet. But we do have a test buy order submission process based on the buy order template. The name is actually 'TESTBUY'.

To see the business process record we can do:

fatt.bsh -show business -bus TESTBUY

That doesn't tells us much so we need to dig deeper and look at the test actions:

fatt.bsh -show action -bus TESTBUY

This will produce:

```
$ACTION1 = {
          'BUSINESS_PROCESS_ID' => 'TESTBUY',
          'ORIG_FILENAME' => '$ORDER_FILENAME',
          'STEP_ELEMENT' => 'file',
          'OUT_DIRECTORY' => '$STAGING_DIR',
          'FILENAME' => 'WO5M${MgmtCode}9.B064${DlrCode}',
          'IN_DIRECTORY' => '$SOURCE_DIR',
          'IN_DIRECTORY' => '/data/fsrv/ffs/mgmt/templates',
          'OUT_DIRECTORY' => '/data/fsrv/ffs/mgmt/${MgmtCode}/bak/staging',
          'ONE_FILE_FLAG' => 0,
          'STEP_SEQ' => '1',
          'DESCRIPTION' => 'create TFS order file',
          'STEP_ACTION' => 'create',
          'DATA_TEMPLATE' => 'buyOrderTemplate.xml'
        };

$ACTION2 = {
          'BUSINESS_PROCESS_ID' => 'TESTBUY',
          'ORIG_FILENAME' => '$ORDER_FILENAME',
          'REPO_NAME' => 'FFS',
          'STEP_ELEMENT' => 'ftp',
```

```
          'OUT_DIRECTORY' => '$ORDER_DIR',
          'FILENAME' => 'WO5M${MgmtCode}9.B064${DlrCode}',
          'IN_DIRECTORY' => '$STAGING_DIR',
          'STEP_SEQ' => '2',
          'DESCRIPTION' => 'submit order file',
          'STEP_ACTION' => 'put',
         };

$ACTION3 = {
          'SQL_LIST' => [
                        'SELECT * FROM tfs_ownr.order_msg@qtd_tfs WHERE
mgmt_company_code = '$MgmtCode' AND dealer_code = $DlrCode AND source_id   =
'$SrcID' AND source_type = '$SrcType' AND order_accepted_date = '$TODAY'"
                       ],
          'BUSINESS_PROCESS_ID' => 'TESTBUY',
          'STEP_ELEMENT' => 'data',
          'MAX_WAIT_IN_SECS' => '60',
          'STEP_SEQ' => '3',
          'COMMAND_LIST' => undef,
          'STEP_SCOPE' => undef,
          'DESCRIPTION' => 'check if order was created',
          'STEP_ACTION' => 'exists',
         };

$ACTION4 = {
          'BUSINESS_PROCESS_ID' => 'TESTBUY',
          'FILENAME' => '$ERROR_ORDER',
          'REPO_NAME' => 'FFS',
          'STEP_ELEMENT' => 'ftp',
          'OUT_DIRECTORY' => '$ORDER_OUT_DIR',
          'IN_DIRECTORY' => '$STAGING_DIR',
          'STEP_SEQ' => '4',
          'DESCRIPTION' => 'download file',
          'STEP_ACTION' => 'get',
         };

$ACTION5 = {
          'BUSINESS_PROCESS_ID' => 'TESTBUY',
          'ORIG_FILENAME' => '$ERROR_ORDER',
          'STEP_ELEMENT' => 'file',
          'IN_DIRECTORY' => '$STAGING_DIR',
          'STEP_SEQ' => '5',
          'DESCRIPTION' => 'show error file',
          'STEP_ACTION' => 'show',
         };
```

```
$ACTION6 = {
        'BUSINESS_PROCESS_ID' => 'TESTBUY',
        'SQL_LIST' => [
                    '@../db/order_load.sql $MgmtCode $TradeDate $INPUT_BATCH_ID'
                  ],
        'REPO_NAME' => 'TFS',
        'STEP_ELEMENT' => 'data',
        'STEP_SEQ' => '6',
        'STEP_SCOPE' => 'batch',
        'DESCRIPTION' => 'transfer order to NSR',
        'STEP_ACTION' => 'run',
       };
```

In simple terms the steps are as follows:

5) Create an order file
6) Update fund cutoff time to future time today
7) Ftp the order file to TFS
8) Check if order was created in TFS
9) Download the error file
10) Display the error file
11) Transfer the order to NSR

At first glance this process looks quite weird; we try to download and display the error file even if the order was successfully loaded.

This is because FATT doesn't currently support alternative processing based on the outcome of the previous step. If it did the step number 4 would be a fork based on the outcome of the step # 3. If the order was loaded the process would go to step number 6, if not it would continue to steps 4 and 5.

To get around this limitation we can setup a linear sequential process and tell FATT to continue on error by using 'continueOnError' parameter.

What this accomplishes is that if the order was rejected FATT will continue in processing and it will download the error file and then display it. And when it tries to execute step number 6 it fails. This way the user gets to see the cause of the failure with touching the keyboard at all.

If the order is processed and loaded, FATT tries to download and show the error file. It fails but it still continues to load the order to NSR because it was instructed to continue on failure.

This way we can have both worlds, we can see the error on failure and we can transfer the order to NSR on success.

Using this development process we can have an order file be it positive or negative for a specific test case developed very quickly. We simply select a file to use as data template and then supply

the testing data needed on the command line in the ‘dataList’ parameter. We run the test process until the desired result, which most of time would mean TFS order acceptance.

Let’s start by entering a basic command; remember we already have template file defined in the first step as 'DATA_TEMPLATE' = 'buyOrderTemplate.xml'.

fatt.bsh –business TESTBUY -adhoc

we get an error:

09:44:27 I Processing test case: Ad Hoc

09:44:27 I Checking parameters for the test

09:44:27 E ===> Action data [WO5V${MgmtCode}9.B074${DlrCode}] still contains unupdated variable(s) <===

09:44:27 E ===> Action data [/data/fsrv/ffs/mgmt/${MgmtCode}/bak/staging] still contains unupdated variable(s) <===

We need 2 basic variables defined. This is were ‘defaultDataList’ parameter comes very handy. We don’t need to enter these 2 values every time on the command line we can just put them in the config file:

defaultDataList=”MgmtCode:YYY;DlrCode:2400”

Remember that default data will always override whatever test case data is in the template or the database.

Because this data is out-of-sight you may forget that it’s overriding the data you think is being used.

This can lead to unexpected behavior which is difficult to trace. Therefore don’t put ‘defaultDataList’ in you config file unless you are certain that it should be used every time FATT runs.

We can also do this:

fatt.bsh –business TESTBUY –adhoc – data ”MgmtCode:YYY;DlrCode:2400”

You have options in FATT in how to go about accomplishing a certain result.

Now when we run the adhoc test it works:

fatt.bsh -bus TESTBUY -ad -data "MgmtCode:YYY;DlrCode:2400" –cont

---

```
10:15:44 I Processing test case: Ad Hoc
        ----------------------------------------------------------------
10:15:44 I Checking parameters for the test
10:15:44 I Created test file:
        [/data/fsrv/ffs/mgmt/YYY/bak/staging/WO5ZYYY9.B0742400]
10:15:44 I Will upload these files to the processing server:
        /data/fsrv/ffs/mgmt/YYY/bak/staging/WO5ZYYY9.B0742400
10:15:55 I Record FOUND in the table tfs_ownr.order_msg@qtd_tfs
10:15:55 I Will download these files from the processing server:
        /data/fsrv/ffs/mgmt/YYY/bak/staging/WK5Z2400.B074YYY9
10:15:55 W Remote file size not available; msg: WK5Z2400.B074YYY9: No such file
        or directory.
        ; will attempt a transfer anyway
10:15:55 E ===> Failed to get remote file WK5Z2400.B074YYY9 - message:
        WK5Z2400.B074YYY9: No such file or directory.
         <===
10:15:55 E ===> File [/data/fsrv/ffs/mgmt/YYY/bak/staging/WK5Z2400.B074YYY9]
        doesn't exist, can't show it <===
10:15:55 I Processed SQL command: [@../db/order_load.sql YYY 20140207 1540038]
10:15:55 I It appears that no output is setup for the test [-1] in TEST_OUPUT
        table, the result defaults to SUCCESS
============================================================
>>>>>>>>>>>>>>>>>>>>>>>>>>> TEST MANAGER RECAP <<<<<<<<<<<<
============================================================
Start Time:          10:15:43
End Time:            10:15:55
Processing Type:     Sequential
Test Tag:            20140207101543
Number of Tests:     1

Processed Test List:

1 ) -1: /home/fundserv/tfs/home/o2400t01/bak/WK5Z2400.B074... --------> SUCCESS
```

Very important issue when submitting orders is that at the minimum there are 2 'MgmtCode' elements. If we are changing this from the default which is in the template we need to instruct FATT to change both fields otherwise our order will be rejected by TFS.

Example

Let's look at a basic example, we want to find a management company and fund id combination that works in TFS.

We know our template is good as setup because it went to success when we ran it above. We can go into the database and select a few fund ids for a management code. Let the management code be ‘YYZ’ and let’s use fund id ‘102’.

Notice the ‘AcctLookup.MgmtCode’ to address the problem mentioned above.

fatt.bsh -bus TESTBUY -ad -data "MgmtCode:YYZ;DlrCode:2400;AcctLookup.MgmtCode:YYZ;FundID:102" -cont

This is a bust with the error file looking like this:

```
<OrdSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="tfs" xsi:sc
                                                                ^
<?xml version="1.0" encoding="UTF-8"?>
<OrdSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="tfs"
xsi:schemaLocation="tfs tfs.xsd" Version="24">
  <Msg>
    <MsgCreate>
      <Date>20140207</Date>
      <Time>105659</Time>
      <Src>
        <SrcType>F</SrcType>
        <MgmtCode>YYZ</MgmtCode>
      </Src>
      <Target>
        <TargetType>D</TargetType>
        <DlrCode>2400</DlrCode>
      </Target>
    </MsgCreate>
    <Ntwrk>
      <NtwrkID>FSRV</NtwrkID>
      <Date>20140207</Date>
      <Time>105629</Time>
      <OrdID>6406664</OrdID>
      <TradeDate>20140207</TradeDate>
```

```
    <SettlDate>20140207</SettlDate>
   </Ntwrk>
   <MsgType>
    <OrdRspn>
     <ActnCode>NEW</ActnCode>
     <SrcID>EZKY</SrcID>
     <DlrCode>2400</DlrCode>
     <RspnSrc>N</RspnSrc>
     <RtnCode>99</RtnCode>
     <Reject>
      <ErrorCode>005</ErrorCode>
     </Reject>
    </OrdRspn>
   </MsgType>
  </Msg>
</OrdSet>
```

Let's try fund id = 100.

fatt.bsh -bus TESTBUY -ad -data "MgmtCode:YYZ;DlrCode:2400;AcctLookup.MgmtCode:YYZ;FundID:100" -cont

This works. Now we know that if we need to use management id 'YYZ' in our test cases we can use fund id 100.

If we don't want to transfer the order to NSR we can leave out the last test action and run it like this:

fatt.bsh -bus TESTBUY -ad -data "MgmtCode:YYZ;DlrCode:2400;AcctLookup.MgmtCode:YYZ;FundID:100"

If the processing doesn't fail, no error file is created and therefore without the 'continueOnError' parameter FATT will exit.

This is a fairly simple example showing how to develop a test case starting with an order submission.

We keep submitting orders with various data until our order is accepted. Then we create a test case by cloning an existing test case. Then we take the file we just submitted and make it the new

process/test template. We can enter the template name in the action DATA_TEMPLATE field if all our future test case can share this very template. If not we enter it as a data template only for this test case.

Then we decide what data if any we need to enter as test case data. For the first test case we can keep all data we need in the template. For each subsequent test case we would reuse this template and enter the specific test case data into the database.

7.2.2. Contract Submission Development

Now that we have and order submission part figured out and test case setup we need to setup the contract processing part of the test case.

As luck would have just like there is a 'TESTBUY' process already defined in the FATT db for order development there is also a 'TESTSETTLEMENT' process defined for contract development.

Our mission is to generate a contract and there are 2 ways of going about it within FATT.

1) We use internal FundSERV contract generation tool to generate a full contract record based on the order that we submitted.
2) We use a predefined contract template and contract data to create a specific contract record for our purposes.

These 2 ways of contract generation are not completely interchangeable, in other words their purpose and use is slightly different.

We use contract generator tool while developing test cases, we use template submission while running the existing test cases or so we should.

The contract generator is very usefull in that it can create for us a valid contract from our order whereas we could spend significant amount of time guessing the right data by using a contract template and plugging in the various data pieces.

Let's have a look how test steps are defined for this development business process by typing in:

fatt.bsh –show business –bus TESTSETTLEMENT

fatt.bsh –show action –bus TESTSETTLEMENT

This second command displays the following test steps (only relevant data is shown):

```
$ACTION1 = {
        'TEST_STEP_ID' => '444',
        'STATUS' => 'A',
```

```
        'BUSINESS_PROCESS_ID' => 'TESTSETTLEMENT',
        'ONE_FILE_FLAG' => 0,
        'STEP_SEQ' => '1',
        'COMMAND_LIST' => [
                    './generateContracts.bsh $TradeDate $MgmtCode $DlrCode'
                  ],
        'DESCRIPTION' => 'generate contracts for the trade date, mgmt code and dealer code',
       };

$ACTION2 = {
        'TEST_STEP_ID' => '445',
        'STATUS' => 'A',
        'BUSINESS_PROCESS_ID' => 'TESTSETTLEMENT',
        'FILENAME' => '$FS_FILENAME',
        'ORIG_DATA_TEMPLATE' => '$GEN_CON_FILE',
        'STEP_ELEMENT' => 'file',
        'OUT_DIRECTORY' => '$STAGING_DIR',
        'IN_DIRECTORY' => '$GEN_DIR',
        'ONE_FILE_FLAG' => 0,
        'STEP_SEQ' => '2',
        'DESCRIPTION' => 'create settlement file',
        'STEP_ACTION' => 'create',
       };

$ACTION3 = {
        'TEST_STEP_ID' => '446',
        'STATUS' => 'A',
        'BUSINESS_PROCESS_ID' => 'TESTSETTLEMENT',
        'FILENAME' => '$FS_FILENAME',
        'STEP_ELEMENT' => 'file',
        'OUT_DIRECTORY' => '$IN_DIR',
        'IN_DIRECTORY' => '$STAGING_DIR',
        'ONE_FILE_FLAG' => 0,
        'STEP_SEQ' => '3',
        'DESCRIPTION' => 'copy settlement file to processing directory',
        'STEP_ACTION' => 'submit',
       };

$ACTION4 = {
        'TEST_STEP_ID' => '447',
        'STATUS' => 'A',
        'BUSINESS_PROCESS_ID' => 'TESTSETTLEMENT',
        'SQL_LIST' => [
                    'select * from NSR_OWNR.contract_master where order_id = '$OrdID''
                  ],
        'STEP_ELEMENT' => 'data',
```

```
        'MAX_WAIT_IN_SECS' => '120',
        'ONE_FILE_FLAG' => 0,
        'STEP_SEQ' => '4',
        'DESCRIPTION' => 'check if contract was created',
        'STEP_ACTION' => 'exists',
       };

$ACTION5 = {
        'TEST_STEP_ID' => '459',
        'STATUS' => 'A',
        'BUSINESS_PROCESS_ID' => 'TESTSETTLEMENT',
        'FILENAME' => '$CONTRACT_K_FILE',
        'STEP_ELEMENT' => 'file',
        'IN_DIRECTORY' => '$OUT_DIR',
        'ONE_FILE_FLAG' => 0,
        'STEP_SEQ' => '5',
        'DESCRIPTION' => 'show error file',
        'STEP_ACTION' => 'show',
       };
```

If we look at the above process we can see 5 steps defined:

1) Generate contracts using the generator tool.
2) Create the contract file by selecting the appropriate contract from the above-generated file
3) Submit the contract file
4) Check if the contract was accepted and created in NSR
5) Display the error file

The logic for the above process is just like the logic for the order submission; since there is no alternative processing in FATT we need to create a a linear process and use ‘continueOnError’ parameter to display the error file should the contract be rejected by the NSR.

But before we do anything we must consider the rule number #1 when submitting contracts; the contract can be submitted the next business day after the trade date at the earliest, no sooner.

Since we are using the same fund as we did in order submission, we know that the fund cutoff time was set to something like the current time plus 4 extra minutes. We can have a look into the db to see what it is to make sure it’s past or regardless of what it is we can just update it to a past time.

Example

Suppose we have created the order portion in the previous step (chapter). The order has been loaded into TFS and transferred to NSR. Now we need to create a contract for it.

If we look at the above steps the step # 1 './generateContracts.bsh $TradeDate $MgmtCode $DlrCode' requires a few parameters. We have to supply these either in the 'dataList' parameter or the 'defaultDataList' or the combination of the two.

In addition we have to remember that when developing a test case we need to use'adHoc' parameter which tells FATT that no specific test case is to be used for the run.

Let's start from the beginning and create an order.

fatt.bsh -bus TESTBUY -data "MgmtCode:YYY;DlrCode:2400" -adhoc -cont –deb

'cont' parameter is needed so that the order is transferred to NSR after the error file download fails (as there is no error file if the template and data are correctly setup).

'deb' means 'debug', show all information available. This option allows us to see the SrcID for the order. We need this value for contract generation. The reason for it is that the contract generator generates contracts for all orders for the specific fund company (MgmtCode), fund dealer (DlrCode) and trade date (TradeDate). However we are only interested in our specific order, we don't care or need any other orders. We use the SrcID to select our contract from the contract file.

If you look at the above process it doesn't include fund cutoff time update. But there is a process called 'CUTOFF_TIME' that we can use to update the time before we actualy submit an order.

fatt.bsh -bus 'CUTOFF_TIME' -show action

To run the process we do this:

fatt.bsh -bus 'CUTOFF_TIME' –data "MgmtCode:YYY;FundID:R100;Cutoff:12000"

The 'Cutoff' variable is the 6-digit time which needs to be in the past in regards to current time; any past time is good.

Now that we have created an order and updated the cutoff time we can create a contract:

fatt.bsh -bus TESTSETTLEMENT -data "MgmtCode:YYY;DlrCode:2400;SrcID:FNFJ;TradeDate:20140210" -adhoc -cont

In addition to 'MgmtCode, DlrCode and TradeDate we also supply the SrcID parameter.

If we did everything right the contract should be loaded by NSR.

We can now take the contract file and make it our template for BUY contracts by entering the file name to the TEST_ACTION.DATA_TEMPLATE field. We should copy the file itself into the '/data/fsrv/ffs/mgmt/templates' directory which is setup in FATT as such. Of course we can put it anywhere else but then we need to setup the directory variable for it in FATT.

The reason we need to have a contract template is that we don't use contract generator for already setup test cases because that would be too slow and cumbersome. It's a lot easier and faster to use the contract generator ad hoc as we just did to create a valid contract and then use that for related test cases as the contract template with test data being added on as needed basis from the database to fulfil the test's objective.

We can also skip the contract generation using the ad hoc development 'TESTSETTLEMENT' process. We do this once we already have a conreact template that fits our purpose,. Again, no need to run the contract generator which always regenerates the same orders for the trade date, fund company and the dealer.

For instance we can do this:

fatt.bsh -bus TESTSETTLEMENT -ad -data "MgmtCode:YYY;DlrCode:2400;" -cont -action 2-6 -template /data/fsrv/ffs/mgmt/YYY/bak/f/reject/FVTHG22400.021114YYY9.Z01

7.3. Other Preconfigured Development Processes

There are a few preconfigured processes in FATT db.

If we do:

fatt.bsh –show business

We get a full list. Of that the below are development processes:

```
$BUSINESS PROCESS10 = {
                      'PROCESS_DESCRIPTION' => 'Update cut off time for a fund',
                      'PROCESS_STATUS' => 'A',
                      'BUSINESS_PROCESS_ID' => 'CUTOFF_TIME'
                     };

$BUSINESS PROCESS11 = {
                      'PROCESS_DESCRIPTION' => 'Find contract in NSR',
                      'PROCESS_STATUS' => 'A',
                      'BUSINESS_PROCESS_ID' => 'GET_CONTRACT'
                     };
```

```
$BUSINESS PROCESS12 = {
                'PROCESS_DESCRIPTION' => 'Get the order from TFS',
                'PROCESS_STATUS' => 'A',
                'BUSINESS_PROCESS_ID' => 'GET_ORDER'
               };

$BUSINESS PROCESS18 = {
                'PROCESS_DESCRIPTION' => 'Transfer order from TFS to NSR',
                'PROCESS_STATUS' => 'A',
                'BUSINESS_PROCESS_ID' => 'TRANSFER'
               };
```

7.3.1. CUTOFF_TIME

The ‘CUTOFF_TIME’ has 2 actions defined:

```
$ACTION1 = {
        'TEST_STEP_ID' => '224',
        'STEP_DATA' => undef,
        'STATUS' => 'A',
        'BUSINESS_PROCESS_ID' => 'CUTOFF_TIME',
        'SQL_LIST' => [
                    'UPDATE tfs_ownr.fund@qtd_tfs set cut_off_time = '$Cutoff' WHERE
mgmt_company_code = '$MgmtCode' AND fund_number = '$FundID"
                  ],
        'REPO_NAME' => undef,
        'STEP_ELEMENT' => 'data',
        'MAX_WAIT_IN_SECS' => undef,
        'FILENAME' => undef,
        'MATCH_ID_RX' => undef,
        'IN_DIRECTORY' => undef,
        'OUT_DIRECTORY' => undef,
        'ONE_FILE_FLAG' => 0,
        'STEP_SEQ' => '1',
        'COMMAND_LIST' => undef,
        'STEP_SCOPE' => 'batch',
        'DESCRIPTION' => 'update fund cutoff time to future time',
        'STEP_ACTION' => 'update',
        'DATA_TEMPLATE' => undef
       };

$ACTION2 = {
        'TEST_STEP_ID' => '225',
        'STEP_DATA' => undef,
```

```
  'STATUS' => 'A',
  'BUSINESS_PROCESS_ID' => 'CUTOFF_TIME',
  'SQL_LIST' => undef,
  'REPO_NAME' => undef,
  'STEP_ELEMENT' => undef,
  'MAX_WAIT_IN_SECS' => '120',
  'FILENAME' => undef,
  'MATCH_ID_RX' => undef,
  'IN_DIRECTORY' => undef,
  'OUT_DIRECTORY' => undef,
  'ONE_FILE_FLAG' => 0,
  'STEP_SEQ' => '2',
  'COMMAND_LIST' => [
                      './refresh.bsh'
                    ],
  'STEP_SCOPE' => 'batch',
  'DESCRIPTION' => 'refresh NSR',
  'STEP_ACTION' => undef,
  'DATA_TEMPLATE' => undef
 };
```

We already used the process when we were creating order and contract files above.

The second action in the process is the NSR refresh. This is only necessary if we are submitting contracts which must be submitted to NSR. If we are submitting orders to TFS/FFS there is no need to run this command as it will have no effect on these systems.

Notice the SQL_LIST statement and the variables required: MgmtCode, FundID, Cutoff. These must be supplied on the command line in the ‘dataList’ or ‘defaultDataList’ or the combination of the two.

To change the fund cutoff time in TFS we can just run the first step like this:

fatt.bsh –bus CUTTOFF_TIME –adhoc –data “MgmtCode:YYY;FundID:R100;Cutoff:120000” -action 1

To run the whole process we leave out the ‘action’ parameter.

fatt.bsh –bus CUTTOFF_TIME –adhoc –data “MgmtCode:YYY;FundID:R100;Cutoff:120000”

7.3.2. GET_CONTRACT

With this process we can check and see if a settlement contract was loaded in the NSR.

fatt.bsh –bus GET_CONTRACT –show action

```
$ACTION1 = {
        'TEST_STEP_ID' => '209',
        'STEP_DATA' => undef,
        'STATUS' => 'A',
        'BUSINESS_PROCESS_ID' => 'GET_CONTRACT',
        'SQL_LIST' => [
                    'select * from NSR_OWNR.contract_master where order_id = '$OrdID"
                  ],
        'REPO_NAME' => undef,
        'STEP_ELEMENT' => 'data',
        'MAX_WAIT_IN_SECS' => undef,
        'FILENAME' => undef,
        'MATCH_ID_RX' => undef,
        'IN_DIRECTORY' => undef,
        'OUT_DIRECTORY' => undef,
        'ONE_FILE_FLAG' => 0,
        'STEP_SEQ' => '1',
        'COMMAND_LIST' => undef,
        'STEP_SCOPE' => undef,
        'DESCRIPTION' => 'check if contract was created',
        'STEP_ACTION' => 'exists',
        'DATA_TEMPLATE' => undef
       };
```

As you can see this process has only a single step in it. To run it we need the order id. The order id is provided by TFS upon loading an order.

To see records displayed with GET_CONTRACT and GET_ORDER we need to use the 'debug' option, otherwise FATT will only say that it found the record but will not show it.

To see an order in NSR we can run the 'GET_ORDER' process below.

7.3.3. GET_ORDER

If we ever need to look up an order in NSR we can use this process. It also has only a single step:

```
$ACTION1 = {
        'TEST_STEP_ID' => '208',
        'STEP_DATA' => undef,
        'STATUS' => 'A',
```

```
        'BUSINESS_PROCESS_ID' => 'GET_ORDER',
        'SQL_LIST' => [
                'SELECT * FROM tfs_ownr.order_msg@qtd_tfs WHERE source_id   =
'$SrcID"
                ],
        'REPO_NAME' => undef,
        'STEP_ELEMENT' => 'data',
        'MAX_WAIT_IN_SECS' => undef,
        'FILENAME' => undef,
        'MATCH_ID_RX' => undef,
        'IN_DIRECTORY' => undef,
        'OUT_DIRECTORY' => undef,
        'ONE_FILE_FLAG' => 0,
        'STEP_SEQ' => '1',
        'COMMAND_LIST' => undef,
        'STEP_SCOPE' => undef,
        'DESCRIPTION' => 'check if order was created',
        'STEP_ACTION' => 'exists',
        'DATA_TEMPLATE' => undef
       };
```

To run this process we need to provide the ‘SrcID’ on the command line.

fatt.bsh –bus GET_ORDER –data “SrcID:12345”

7.3.4. ORDER_TRANSFER

If we need to transfer an order from TFS to NSR we can use this process. Again, it’s only a single step process. It uses MgmtCode, TradeDate and INPUT_BATCH_ID as input parameters. These need to be supplied on-line and can be read from the order using the ‘GET_ORDER’ process.

```
$ACTION1 = {
        'TEST_STEP_ID' => '226',
        'STEP_DATA' => undef,
        'STATUS' => 'A',
        'BUSINESS_PROCESS_ID' => 'TRANSFER',
        'SQL_LIST' => [
                '@../db/order_load.sql $MgmtCode $TradeDate $INPUT_BATCH_ID'
                ],
        'REPO_NAME' => 'TFS',
        'STEP_ELEMENT' => 'data',
        'MAX_WAIT_IN_SECS' => undef,
        'FILENAME' => undef,
```

```
  'MATCH_ID_RX' => undef,
  'IN_DIRECTORY' => undef,
  'OUT_DIRECTORY' => undef,
  'ONE_FILE_FLAG' => 0,
  'STEP_SEQ' => '1',
  'COMMAND_LIST' => undef,
  'STEP_SCOPE' => 'batch',
  'DESCRIPTION' => 'transfer order to NSR',
  'STEP_ACTION' => 'run',
  'DATA_TEMPLATE' => undef
};
```

7.3.5. How To Create New Processes

Very often it's not necessary to create a new process, it may be enough to run the subset of an existing process.

For instance if we wanted to just create a new contract file from an existing order using the contract generator tool we could run just the first 2 steps of the 'TESTSETTLEMENT' business process to get the file generated like this:

fatt.bsh –bus TESTSETTLEMENT –adhoc –action 1-2 –data "MgmtCode:YYY;DlrCode:2400;TradeDate:20140201;SrcID:1ZXSW"

An observant user might have noticed by now how easy it is to create a new process especially if the process already has steps defined in some other process.

All that needs to be done is to copy the existing steps and give them a new business process name.

Developing a brand new task is somewhat more involved but usually not too difficult.

If the user is not able to configure a business and/or step he/she needs to ask somebody in the development team for help.

In general there should be a developer assigned to FATT to configure new test steps as requested by the users.

7.4. Test Case Development By Cloning Existing Process or Test Case

Rule of thumb: Always look into the FATT database to see what processes and test cases are already defined there. More likely than not there is already a process defined that is either the same or similar to our requirement. In such a case we either reuse the existing process (by

using its BUSINESS_PROCESS_ID label for our test cases) or clone it to a new process and update it to fit our needs.

Find Similar Business Process Already Defined

To see all the business processes defined type in:

fatt.bsh –show business

It can be very useful to have a precise description of the process in the 'PROCESS_DESCRIPTION' field so that the user can tell immedtialy what the process does exactly and whether it's reusable for him/her.

One of the business processes we see is 'BUYINT_CONTRACT'. This is contract testing for buy transactions with an intermediary. This business process is what we may need, it has 2 actions: create settlement file, submit settlement file.

To confirm the business process contains the test steps we need we look at the test actions defined:

fatt.bsh –show action –bus BUYINT_CONTRACT

Initially we might be tempted to use this very process but at closer look we see that there are details that don't fit our needs.

Specifically this process uses intermediary code within the file name and file directories. Our process doesn't have intermediaries and so our file names and directories are different. Therefore we cannot reuse the process. We must create our own process.

Clone An Existing Process To The New Process

What we can do is to copy this process to our new process. We do this specifying the source id in the 'id' field and optionally the new business process id in the data parameter:

fatt.bsh –create business –id BUYINT_CONTRACT -data "BUSINESS_PROCESS_ID:BUY_CONTRACT"

When we clone a business process FATT creates a new record in the BUSINESS_PROCESS table and copies existing actions defined in the TEST_ACTION table for the business into the new business.

If the new BUSINESS_PROCESS_ID is specified as above then this id will be used. If nothing is specified then FATT will use the business id to copy and append '_' (underscore) followed by the next SrcID as defined in the variables table.

The business id should be meaningful and therefore the user should either specify the id as in the above command or update the id immediately after the business record has been created by running:

fatt.bsh –update business –id CURRENT_ID –data BUSINESS_PROCESS_ID:BUY_CONTRACT"

fatt.bsh –update action –id CURRENT_ID –data "BUSINESS_PROCESS_ID:BUY_CONTRACT"

If the business process has no specific actions defined for its BUSINESS_PROCESS_ID nothing will be cloned for the new process.

FATT will not copy output records defined for the business process in the TEST_OUTPUT table if there are any. This is because the output records are either too generic and apply to numerous business processes or too specific and apply to specific test cases. Therefore there shouldn't really be specific business process output records most of the time. If there are and they need to be cloned for the new process this has to be done separately by the user.

Now we need to adjust the actions that we just copied. In this case we know that intermediary orders (and contracts) are different from those of a dealer. This is because the intermediary code is used in the file name and directory name. We need to replace this with the dealer code.

After the updates our two actions will look like this:

TEST_STEP_ID	464
MATCH_ID_RX	undef
STATUS	A
BUSINESS_PROCESS_ID	BUY_CONTRACT
FILENAME	$FS_FILENAME
STEP_ELEMENT	file
OUT_DIRECTORY	$STAGING_DIR
IN_DIRECTORY	$SOURCE_DIR
ONE_FILE_FLAG	N
STEP_SEQ	1
DESCRIPTION	create settlement file
STEP_ACTION	Create
DATA_TEMPLATE	undef

The above action will create a settlement file. Notice that we are not defining the data template here. This is because this process will use several templates and therefore they need to be defined at the level of the test case (TEST_CASE.DATA_TEMPLATE).

The submission action record will look like this:

TEST_STEP_ID	464
MATCH_ID_RX	undef
STATUS	A
BUSINESS_PROCESS_ID	BUY_CONTRACT
FILENAME	$FS_FILENAME
STEP_ELEMENT	file
OUT_DIRECTORY	$IN_DIR
IN_DIRECTORY	$STAGING_DIR
ONE_FILE_FLAG	N
STEP_SEQ	2
DESCRIPTION	Submit settlement file to NSR
STEP_ACTION	submit
DATA_TEMPLATE	undef

Now we have the test actions defined we need to look at the output. We know there are output records already defined for other processes that do exactly the same thing as our process will. They look for a 'K' file.

If we look into the database we see that there are 2 records that will work for us. Or we can use FATT to find out if there is an existing record that will work by using a specific test id from the business process like below:

fatt.bsh -test 7543 -show case

We run the above line just to make sure we have a valid test case for the process id:

```
$TEST CASE1 = {
          'TICKET_ID' => undef,
          'STATUS' => 'A',
          'EXPECTED_RESULT' => 'Pass',
          'BUSINESS_PROCESS_ID' => 'BUY_CONTRACT',
          'TEST_CASE_ID' => '7543',
          'TEST_CONDITION' => 'BuyAAFund.BuyFund.Currency=EUR and
BuyAAFund.SettlSrc=F  1) Settlement Method=1   2) EUR Trade Related Bank Account for
Fund company exist',
          'REFERENCED_DATA' => undef,
          'SECTION' => undef,
          'USER_ID' => undef,
          'UPDATED_AT' => undef,
          'COMMENTS' => undef,
          'SOURCE_ID' => undef,
          'BRD_REF' => 'FST5042400.071013YYY9.Z01',
```

```
  'MATCH_ID' => '24BBEK100PASS',
  'SUBSECTION' => '20130621-hyu570',
  'ERROR_CODE' => undef,
  'DESCRIPTION' => undef,
  'DATA_TEMPLATE' => 'settlementBUYAA.xml'
 };
```

Once confirmed we can run:

fatt.bsh -test 7543 -show output

This will show the following output record:

```
$TEST OUTPUT1 = {
   'OUTPUT_ACTION' => 'not exists',
   'OUTPUT_VALUE_RX' => 's/^FS/FK/',
   'STATUS' => 'A',
   'OUT_DIRECTORY' => '$OUT_DIR',
   'WAIT_FOR' => undef,
   'BUSINESS_PROCESS_ID' => undef,
   'OUTPUT_TYPE' => 'file',
   'TEST_OUTPUT_ID' => '354',
   'STEP_SEQ' => '1',
   'RECORD_COUNT' => undef,
   'OUTPUT_FIELD' => '$FS_FILENAME',
   'DATA_SOURCE_NAME' => undef,
   'TEST_CASE_ID_REF_FIELD' => undef,
   'MATCH_ID_RX' => '24BBEK.*PASS',
   'UPDATED_AT' => undef
  };
```

The output record is fine for our purposes. But we also need to check if we have an output record for processing failure, a record with EXPECTED_RESULT = fail.

We get another test case from the process in this case it's 7545 and do the same thing:

fatt.bsh -test 7545 -show output

We get:

```
$TEST OUTPUT1 = {
   'OUTPUT_ACTION' => 'exists',
   'OUTPUT_VALUE_RX' => 's/^FS/FK/',
   'STATUS' => 'A',
```

```
 'OUT_DIRECTORY' => '$OUT_DIR',
 'WAIT_FOR' => undef,
 'BUSINESS_PROCESS_ID' => undef,
 'OUTPUT_TYPE' => 'file',
 'TEST_OUTPUT_ID' => '353',
 'STEP_SEQ' => '1',
 'RECORD_COUNT' => undef,
 'OUTPUT_FIELD' => '$FS_FILENAME',
 'DATA_SOURCE_NAME' => undef,
 'TEST_CASE_ID_REF_FIELD' => undef,
 'MATCH_ID_RX' => '24BBEK.*FAIL',
 'UPDATED_AT' => undef
};
```

This record is good for our purposes hence we don't need to create an output record to match failed processing.

For failures we usually need to define additional output records for the expected error code. For the test case id 7545 we are expecting error '034' and error detail 'B'. We need to create these two output records in the database.

In FATT we don't create records directly but by copying an existing record and updating the data fields as needed.

If we don't know an existing record that we can copy we can select all records and look for a ny record that has 'OUTPUT_FIELD' = 'ErrorCode' then we copy that record.

fatt.bsh –show output

Let's say we copy record with test_output_id = 64.

fatt.bsh –create output –id 64 –data "OUTPUT_VALUE_RX:034;MATCH_ID_RX: 24BBEK100PASS"

As seen above we only need to change 2 fields, the value and the match id so that it matches to only this record and no other.

We do the same thing for 'ErrorCodeDtl' record.

Test Case Cloning

We could be done if we didn't jump over a small detail above, namely we assumed that there was a test case already created for the business process id.

We create test cases just like any other record in FATT, we copy an existing test case. It doesn't matter which record we copy as long as we update it properly to reflect the new test case because the new test case has all fields set to null (other than the test case id field).

Do not clone a test case using 'id' parameter such as 'fatt.bsh –create test –id 5555'. This behavior is not supported and will not work.

Use 'testIdList' or 'matchIdList' parameter in which case the first test case record will be cloned, as a source test case for cloning.

The test case data will be cloned as well to the new test case.

MATCH_ID In Test Case Cloning

Cloning a test case poses a special issue, what to do about MATCH_ID field.

As explained above MATCH_ID is a key field that is used thoughtout FATT database to associate test cases to test steps/actions, test output and occasionally to test case data.

Despite its importance there are no rules governing the values used for MATCH_ID. They don't even have to be unique though they should. In fact MATCH_ID doesn't even need to be set though it should.

To help the user in creating MATCH_ID there is an option called 'makeMatchId'. This option accepts a string of up to 30 chars in length which is then used to create MATCH_ID for the test case like this:

"matchIdParameter" + TEST_CASE_ID + "(FAIL|PASS)"

If no string is given the value defaults to 'FATT'. If a string is given and it ends in either 'PASS' or 'FAIL' this value will be appended to the new MATCH_ID. There is no default for 'PASS|FAIL' functionality.

The match ids currently setup don't use test case_id in their value but they should.

TO DO: Update the existing match ids by replacing the numeric value within the match id with the test case TEST_CASE_ID value for the sake of consistency. It's not necessary or required but it would be nice to have.

7.5. Reclassifying Test Cases

Sometimes we create a test case but can't make it work. It seems like the test case just can't work as defined.

Test Case # 7671

One such case was the test case # 7671. It was designed as part of the full 'BUY' business process run. However no matter what was tried the test case just never failed with various errors when it was expected to pass.

Eventually it became clear that the test case couldn't get though the whole cycle of BUY process. Specifically it wasn't designed to include the order steps but only the contract part.

In FATT it's easy to reclassify a test case without doing much more than changing the process id of the test case.

First however we must check that an alternative business process already exists for it if not we need to create it first.

The alternative business process in this case is 'BUY_CONTRACT'. It consists of only the 2 steps we need, contract creation and contract submission.

We simply change the business process id with this command:

fatt.bsh –update case –test 7671 –data "business_process_id:BUY_CONTRACT"

Now if we run the test case we find out that it's still not working . The reason for that is that the settlement date we assign to the contract is not valid for the fund we are using.

Because we are submitting only contracts in this business process we have to supply the settlement and trade dates to each contract. This is normally done by FundSERV when accepting orders. For the sake of simplicity in this business process we use 1 fund which requires a specific trade and settlement dates which we create using system variables.

However in the case # 7671 this fund cannot be used and we have to use another fund that needs a different settlement date and trade date.

The problem is that selttlement date is set up as the variable for the process.

So we change a tactic and we will not update the settlement date but rather the trade date. We will keep the settlement date to be the current date and set the trade date to a date that the specific fund requires as it is trade date/settlement date combination that must be correct rather than each date individually.

The trade date currently set up for this test case looks like this:

fatt.bsh –show data –test 7671

$DATA4 = {

```
'TEST_DATA_ID' => '17203',
'TEST_FIELD_ACTION' => '=',
'OPTIONAL_FLAG' => 1,
'STATUS' => 'A',
'BUSINESS_PROCESS_ID' => 'BUY_CONTRACT',
'TEST_CASE_ID' => undef,
'STEP_SEQ' => undef,
'TEST_FIELD_NAME' => 'TradeDate',
'ORIG_TEST_FIELD_VALUE' => '$TRADE_DATE',
'TEST_FIELD_SEQ' => undef,
'TEST_FIELD_VALUE' => '20140131',
'UPDATED_AT' => '20130730000000',
'LIST_FLAG' => 0
};
```

To get around this issue we need to setup a test case variable that will override this trade date.

First we need to setup the variable in the SYS_VARS table and we will call it 'PREVIOUS_B_DAY'. This will give us the actual value we need which we will assign to the test case variable. This is because the fund we are using has next business day settlement after the trade date.

```
$SYSTEM VARIABLES31 = {
                'ORIG_VAR_NAME' => 'PREVIOUS_B_DAY',
                'STATUS' => 'A',
                'VAR_NAME' => 'PREVIOUS_B_DAY',
                'CURRENT_VALUE' => undef,
                'VAR_TYPE' => 'function',
                'VAR_VALUE' => 'getNextBusinessDate(-1)',
                'REFRESH_INTERVAL' => 'R',
                'UPDATED_AT' => '20130924000000',
                'MATCH_ID_RX' => undef
              };
```

Once the variable is set up we create a test case data record for our test case:

```
$DATA7 = {
        'TEST_DATA_ID' => '17210',
        'TEST_FIELD_ACTION' => '=',
        'OPTIONAL_FLAG' => 0,
        'STATUS' => 'A',
        'BUSINESS_PROCESS_ID' => undef,
        'TEST_CASE_ID' => '7671',
        'STEP_SEQ' => undef,
        'TEST_FIELD_NAME' => 'TradeDate',
        'ORIG_TEST_FIELD_VALUE' => '$PREVIOUS_B_DAY',
```

```
  'TEST_FIELD_SEQ' => undef,
  'TEST_FIELD_VALUE' => '20140204',
  'UPDATED_AT' => '20130812000000',
  'LIST_FLAG' => N
};
```

We still have the business trade date variable from the business process but it will be overridden by the test case variable.

This solves our problem and the test case now passes without failure.

Sometimes if we are transferring a test case from one business process to another we need to create additional test case data records . This is especially the case if we are reducing the number of steps and leaving out a significant part such as order creation/submission. Order file may have some elements that may be required by the contract file.

These elements are automatically transferred to the contract file by the use of variables defined in the PROCESSING_DATA table.

If the order is not created there is nothing to be transferred and therefore these variables if required need to be entered into the TEST_CASE_DATA table.

7.6. Test Setup Tips & Tricks

In testing we often have special cases that we need to deal with. In those case the test setup needs to be inventive. Below are a few cases that demonstrate this.

Test Case ID: 4372

In the test case id 4372 we need to test the situation where DlrCode element is empty. The problem arises because DlrCode is used to setup the contract file name. If there is no value FATT will complain that it cannot construct the file name and exit.

To get around this problem we will setup 2 test case data records for DlrCode. The first one will have a valid value such as ‘2400’ while the second one will be empty. This will allow FATT to use the first value to create the file name and to use the empty value as the data element in the test file itself.

The only thing we need to make sure of is that ‘empty’ dealer code is after the dealer code with a value. We do this by setting up the ‘TEST_FIELD_SEQ’ field. We normally don’t need to setup this field, we only do it for special cases like this. We set the ‘DlrCode = 2400’ record to 1 and ‘DlrCode is empty’ to 2 so that the records look like this:

fatt.bsh –show “case data” –test 4372

```
.
.
.
$DATA3 = {
          'TEST_DATA_ID' => '16239',
          'TEST_FIELD_ACTION' => '=',
          'OPTIONAL_FLAG' => 0,
          'STATUS' => 'A',
          'BUSINESS_PROCESS_ID' => undef,
          'TEST_CASE_ID' => '4372',
          'STEP_SEQ' => undef,
          'TEST_FIELD_NAME' => 'DlrCode',
          'TEST_FIELD_SEQ' => '1',
          'TEST_FIELD_VALUE' => '2400',
          'UPDATED_AT' => '20131004125025',
          'LIST_FLAG' => 0
        };

$DATA4 = {
          'TEST_DATA_ID' => '6933',
          'TEST_FIELD_ACTION' => 'empty',
          'OPTIONAL_FLAG' => 0,
          'STATUS' => 'A',
          'BUSINESS_PROCESS_ID' => undef,
          'TEST_CASE_ID' => '4372',
          'STEP_SEQ' => undef,
          'TEST_FIELD_NAME' => 'DlrCode',
          'TEST_FIELD_SEQ' => '2',
          'TEST_FIELD_VALUE' => undef,
          'UPDATED_AT' => '20130731085213',
          'LIST_FLAG' => 0
        };
.
.
.
```

Now when we run the test case it works as it should.

This is a common trick that can be used numerous times whenever we have a field that FATT needs for processing but the test itself concerns the lack or unavailability of the very same field.

TO DO: More special cases need to be discussed.

8. Appendix

8.1. Regular Expression Use In FATT

Regular expressions are a complex subject about which countless books have been written, courses taught, tutorials published. We will not cover even the basics here just enough to make the user aware that regexes exist and how they are used within FATT.

What is a regular expression or regex as the term is used throughout FATT? Regular expression is a sequence of characters that forms a search pattern or so says the Wikipedia (http://en.wikipedia.org/wiki/Regular_expression).

There are two main types of operations that can be done using regexes.

String or pattern matching – the pattern is enclosed in a set of (usually) slashes like this **/string/**
String or pattern replacement – the pattern to be replaced is enclosed in slashes followed by the replacement string also enclosed in slashes like this **s/pattern/replacement/**; the ‘s’ at the beginning of the expression is optional, it may or may not be included.

The characters within a regex string can have one of two meanings; a character is either a literal value or it can be a meatcharacter.

8.2. User Exercises

1. General FATT Usage

a) Login to the server ‘142.48.44.62’ as ‘nsrapp’ user or with your own login and use ‘su’ to switch to ‘nsrapp’.

b) Go to the ‘~fatt/bin’ directory:

cd /app/fsrv/nsr/fatt/bin

c) Look at the the profile files defined:

more ../config/.fatt_profile
more ../config/fatt.config

d) Look at the the bash script wrapper:

more fatt.bsh

e) To see the options available type in:

fatt.bsh –help

f) Create a new section in the fatt.config file and name it using your own name so that the section looks like this:

[russ_user]
dbServer=142.148.44.62
dbName=NSR
dbUser=qtd_usr
.
.
.
debug=Y

As you can see we included an extra line 'debug=Y' to show how the config file can be customized. Now whenever FATT is run with this config section 'debug' option will be set by default.

g) To use your new custom section you have 2 options:

- You can run the the perl script directly

  perl ./fattRunTest.pl -configSection russ_user

- You can create your own wrapper script by copying **fatt.bsh** to something like **russ_fatt.bsh** and replace the 'fatt_user' with 'russ_user' in the script.

It's not necessary for each user to have his/her own configuration but it may be useful for some users. Especially useful should be the 'default data' functionality that alows the user to permanently override the data as setup in the database or the data template.

h) Type in invalid parameter such as:

fatt.bsh -dummy

Observe the output, scroll up and see the message you received saying the parameter was invalid.

Type in:

fatt.bsh –test 5866 -full

Again, scroll up and see the message saying the 'full' option is ambiguous as it can be 'fullDisplay' or 'fullDate'.

8.3. Currently Defined Variables In SYS_VARS Table

VAR_NAME	VAR_TYPE	VAR_VALUE
STAGING_DIR		/data/fsrv/ffs/mgmt/${MgmtCode}/bak/staging
ENCODING		ISO-8859-1
TRADE_DATE	function	getNextBusinessDate(-3)
HOLIDAYS		0101,1224,0701,20130805
SOURCE_DIR		/data/fsrv/ffs/mgmt/templates
IN_DIR		/data/fsrv/ffs/mgmt/${MgmtCode}/in
TAG	function	getNextSeqNumberB36
SETTLEMENT_DATE	function	getNextBusinessDate
TODAY_MMDDYY	function	getTodaysDate()
DEALER_CODE		2400
MANUFACTURER_CODE		${MgmtCode}9
FILE_EXT		Z01
SEQUENCE	function	getNextSeqNumberB36(3)
FS_FILENAME		FST${SEQUENCE}${DlrCode}.${TODAY_MMDDYY}${MgmtCode}9.${FILE_EXT}
CUT_OFF_TIME	function	getTime()
CUT_OFF_FUTURE	function	getTime(240)
DEFAULT_MgmtCode		YYY
DEFAULT_DlrCode		2400
SETTLEMENT_MINUS_1	function	getNextDateFromDate(${SettlDate},-1)
IN_TWO_DAYS	function	getNextBusinessDate(+2)
IN_FOUR_DAYS	function	getNextBusinessDate(+4)
NEXT_SUNDAY	function	getNextDate(Sunday)
SWITCH_TRADE_DATE	function	getNextBusinessDate(-1)
SOURCE_ID	function	getNextSeqNumberB36
YESTERDAY	function	getNextBusinessDate(-1)
TODAY	function	getTodaysDate(YYYYMMDD)
TOMORROW	function	getNextBusinessDate(+1)
ORDER_FILENAME		WO${ORDER_FILENAME_SEQ}${MgmtCode}9.${TODAY_MDDY}${DlrCode}
TODAY_MDDY	function	getTodaysDate(MDDY)
ORDER_FILENAME_SEQ	function	getNextSeqNumberB36(2)
ORDER_DIR		/home/fundserv/tfs/home/o${DlrCode}t01/in
YYY_STAGING_DIR		/data/fsrv/ffs/mgmt/YYY/bak/staging
YYY_IN_DIR		/data/fsrv/ffs/mgmt/YYY/in
YYY_OUT_DIR		/data/fsrv/ffs/mgmt/YYY/out
DEFAULT_DIR		/tmp
DEFAULT_FILENAME		FST${SEQUENCE}2400.${TODAY_MMDDYY}YYY9.Z01
INVALID_MGMT_CODE		YYYY
OUT_DIR		/data/fsrv/ffs/mgmt/${MgmtCode}/out

INT_ORDER_DIR		/home/fundserv/tfs/home/o${IntCode}t01/in
INT_ORDER_FILENAME		WO${ORDER_FILENAME_SEQ}${MgmtCode}9.${TODAY_MDDY}${IntCode}
INT_FS_FILENAME		FST${SEQUENCE}${IntCode}.${TODAY_MMDDYY}${MgmtCode}9.${FILE_EXT}
ORDER_OUT_DIR		/home/fundserv/tfs/home/o${DlrCode}t01/bak
ERROR_ORDER		WK${ORDER_FILENAME_SEQ}${DlrCode}.${TODAY_MDDY}${MgmtCode}9
MMDDYY	function	getTodaysDate(MMDDYY)
GEN_DIR		/tmp/qa/mgmt/${MgmtCode}/in
NEXT_SATURDAY	function	getNextDate(Saturday)

8.4. Possible Enhancements

✓ *Develop default data functionality – this will allow users to have their own default data such as MgmtCode, FundID, DlrCode etc. This will also cut down on the amount of test case data that needs to be setup for each test. DONE*

- Allow test options to be stored at the test level in the TEST_CASE table.
- Allow alternative test result so that one can create oputput records that are combined using logical ‘OR’ in addition to the current logical ‘AND’.
- Allow to specify whether the data is positive or negative. Calculate the expected failure rate when combining positive and negative data in the test.

8.5. FAQ

1. Can I run a test case or test cases defined for one business process under another business process?

   Yes, use the parameter ‘runBusinessId’ to specify which business process you want to run the test cases under:

   fatt.bsh –bus BUY_CONTRACT –runbus BUY_CONTRACT_PC

2. How is the above useful? It can be useful if for example you want to run you test cases from a different environment such as your PC.

   In a case like that most of time the test cases will not work as they are setup for a specific environment. To make them work the user can copy the existing process to a new process, update the processing steps as required and then run the desired test cases with the above parameters.

3. Can I run just a subset of steps from a business process?

   Yes, to run a single step you can type in :

   fatt.bsh –bus TESTBUY –action 1

   To run more then one step:

   fatt.bsh –bus TESTBUY –action 1-3

   fatt.bsh –bus BUY –action 9-11

   The end number for the action doesn't need to be correct. For example if the process 'BUY' has only 10 steps the above command is still valid.

4. Can I actually run the process 'the next business day' rather than use the simulation?

   Yes, you have to have a predefined process, then you run only the first number of steps the first day. You have to use 'logData' option the first day so that FATT saves the run-time data in the database.

   fatt.bsh –bus BUY –action 1-6 –logdata

   The next day you can run the rest of the actions. In our case we are using the process that simulates the next business day, since we are running actually the next business day we don't need to simulate it but if we do no harm will be done.

   To run the rest of the process without simulation we can type in:

   fatt.bsh –bus BUY –action 9-11 –restart

   To run the rest of the process with simulation we can type in:

   fatt.bsh –bus BUY –action 7-11 –restart

   The difference is the two steps skipped in the first command.

5. What if I want to run a test case with a preconfigured file, in other words a file I have already created previously while doing testing, can I do that?

Yes, but depending on the type of test you are doing an existing file may not be of much use because unless you are testing for file format only the file still needs to be updated with at least some current data such as TradeDate and SettlDate otherwise it will fail.

But in general you can run a preexisting file if you replace the DATA_TEMPLATE entry with the file name and if the file contains several records you need to specify the SrcID of the correct record in the SOURCE_ID field of the test case. FATT will extract the right record from the source file and create a new file from it.

9. FATT Roadmap

- User Testing & Training

  FATT contains tremendous amount of functionality but this hasn't been seriously tested in one of its key functionalitites namely as a development tool.

  Before the users can test it though they need to be trained. The training would need to be rigorous and hands-on to achieve the desired proficiency by the users.

  The functionality of test-processing engine on the other hand has been regressively tested on a daily basis and should be stable.

- FATT Refactoring

  Because the code was developed without much if any specific requirements, the code at least in its detail is quite "loose" and benefit from a round or two of refactoring to further de-spaghettify it and make it more structured rather than linear.

  This could be done in tandem with user training and testing.

- Documentation Completion

  The documentation is quite thorough at over 120 pages but is not fully finished. The Apendix is a hodgepodge of chapters that have been started but not finished.

  There was no time for development documentation. This may require 20-30 additional pages.

  To users the documentation may also seem too "thick" in the sense that too much information is crammed into it. The documenmtation may need to be spread out into a lot more pages to make it more readable.

  In short the documentation itself would need a round of refactoring.

- Web Testing Extension

  The existing Selenium web-testing project hasn't been incorporated into FATT. This could and should be done in the future.

  The current status of the Selenium project is little more than a proof of concept. It's not clear whether the current status could be used as abase for future development or whether it would need to be rewritten.