A Design Pattern for Flexible Script Solutions Using the Eresia FIP, NIP and XIP Portals

CML09005-01

Code Magus Limited (England reg. no. 4024745)
Number 6, 69 Woodstock Road
Oxford, OX2 6EY, United Kingdom
www.codemagus.com

June 14, 2010

Contents

1 Description

This course describes a design pattern as a means to solve design issues encountered around the creation of toolkits based on the Eresia File Injection Portal, the Eresia Network Injection Portal and the Eresia XML Injection Portal.

The pattern presented in the text, although intended for, is not limited for use with these tools.

2 Objectives

As with any test scripting tool set, effort spent scripting test solutions using Eresia and the FIP, NIP and XIP needs to deliver solutions to the users of those scripts which meet their objectives without the burden of requiring the users to supply volumes of non-scenario data and/or elements functionally dependent on non-scenario specific requirements. This allows the scripts and associated artefacts to address the users changing needs within the confines of the channel opened by those script without continued and recurrent reliance on the script writer to populate and execute the test pack. And it allows the users to succinctly state their scenario content requirements. The only reliance on, and hence return to, the script writer is when those items taken into account when providing the channel changes, items such as meta-data, format specifications, access methods, and network protocols. There should be no reliance on script writers for content.

This course teachers a tried and tested pattern which delivers on these objectives, allowing the test analyst users to contribute content to maximise coverage and for the testers to be able to execute the tests themselves. This relieves the script writer from the burden of configuring scenarios and of having to execute the tests themselves, allowing them to concentrate on enabling other test channels to be opened.

3 Prerequisites

The Prerequisites for this course are as follows:

- Object Types[3]
- Thistle
- Eresia Visual Test Environment
- The Eresia File Injection Portal (Eresia FIP)[8]
- The Eresia Network Injection Portal (Eresia NIP)[9]

- The Eresia XML Injection Portal (Eresia XIP)[10]
- Network Control Programs (NCP's)
- Access Methods
- XML/XSD/WSDL

4 Introduction

Developing any sort of software or solution can be a difficult and tedious process and can be even more so when one aims at providing solutions that are reusable, extensible and general at the same time.

The aim of this course is to empower developers and testers alike, to be able to create and use versatile and robust solutions with similar interfaces without the need to focus on the underlying design and thus being able to apply their energies on solving the problem.

4.1 What is a Design Pattern

Design Patterns are a means of solving recurring problems in a similar manner all the while creating a solution that is unique and particular to the initial problem statement.

Design Patterns are fundamentally different from algorithms in that algorithms solve computational problems whereas design patterns solve the design issues encountered in the implementation of these ideas.

Design Patterns offer an approach which can be used over and over in solving problems of similar nature, which implies that the problem statement will to a large degree determine the type of Design Pattern that one would use to solve a particular problem.

In the context of this course, a design pattern is introduced to aid developers and testers alike, by providing a unique and intuitive approach to designing and using Thistle based solutions for common problems encountered in batch and online systems, amongst others.

4.2 Why Should we use Design Patterns

The development and automation of file creation and online messaging toolkits has been a field in which Code Magus has been involved for several years.

What was required were toolkits that despite being very different in their underlying detail behaved in a consistent manner.

An important aspect of this is that, for the end users, a common feel accross multiple solutions across several different types of tools be provided, unifying them with a common interface and understanding.

This allows for an expectation and familiarity to be created where the user's expectation of familiarity or intuitive understanding is fulfilled.

From a development perspective, first principles are done away with, Developers will now have a definite pattern spelling out the requirements of usability of the solution, knowing that the effort is being spent on a user community that will find it usable and familiar without extensive retraining.

Users can now allow their efforts to be spent at their domain and expert knowledge supplying appropriate content to enrich the quality of testing.

The material presented in this course will aid in the development of future toolkits in that the development process will be streamlined in terms of design, future enhancements will not require a complete rework of the existing code and that the end users of the toolkits will not have to modify their testing approach each time a new toolkit is developed.

5 Pattern Elements

5.1 Intent

The intention of this course and design patterns as a field of study to allow developers to provide robust solutions that have not been worked from first principles, in other words, not having to rethink a design approach each time a new solution is required.

This provides a means for developers to build on tried and tested design solutions that have worked in the past, which promotes elegant and extensible code and consistency of interfaces across solutions.

5.2 Applicablility

The Patterns presented in this course have been developed in such a way as to apply to a wide variety of scenarios, in that they offer a means of grouping components, interfaces and mechanisms in order to provide a solution which will be familiar to the end user.

Going forward developers will be able to apply the techniques and outcomes provided in this course to wide variety of situations when using tools such as the Eresia Network Injection Portal, the Eresia File Injection Portal and the Eresia XML Injection Portal.

5.3 Scalability

Scalability, in terms of software engineering, by definition is a desirable property of any system. It indicates an ability to handle growing amounts of work and can be readily and easily enlarged.

A Pattern is said to scale if it is suitably efficient and practical when applied to large situations.

In the context of this course the pattern that will be presented has been designed with this in mind.

6 Benefits of Design Patterns

The development of testing solutions in a fast moving environment where the SUT[1] is constantly changing and evolving to meet the needs of business, often pose enormous challenges in meeting critical deadlines while at the same time having to provide a reusable and bug free solution.

Developers are often required to analyse the requirements, design an efficient and effective solution as well as test the solution in ever shortening spaces of time.

The aim then of having an adaptable and scalable design pattern is to allow developers to focus on the requirements that need to be met rather than spend time focusing on a unique design approach for each new solution.

End users will be able to use the provided toolset and solutions confidently, in that they will not have to rethink their testing approach as well as not have to be trained to use new testing solutions and tool sets each time a new product requires testing.

At the highest level, the institute or organisation that will mandate the particular testing solution both from a development and usage point of view will benefit from the decreased development time, decreased training costs and an increased testing scope.

7 Pattern Components

The Design Pattern for Flexible Script Solutions approach to developing solutions can be thought of in terms of a inter-connected chain of components working together in a structured manner to achieve a well structured design, free from any particular implementation details.

[1] System Under Test

All the components as well as the interfaces to those components will be common across all solutions. All the components will be discussed in detail in subsequent sections.

8 Pattern Structure

Any solution, from design to implementation, will typically be made up of several components. These components are constantly being revised and updated in order to keep up with ever changing business requirements, updated functionality and fixes that inevitably need to be applied.

The Developers, support staff as well as the end users may change over time, it is therefore of the utmost importance to structure and organise all the components in a meaningful and consistent manner, so that time is not wasted trying to locate components.

Once a suitable structure and organization of components has been established, it is then critical to track changes and store a version history of each component in an appropriate version control system.

Versioning provides future developers and support staff with an accurate history of the changes made to components as well as the motivation for the change. A versioning system will allow developer to roll back to previous versions should the need arise.

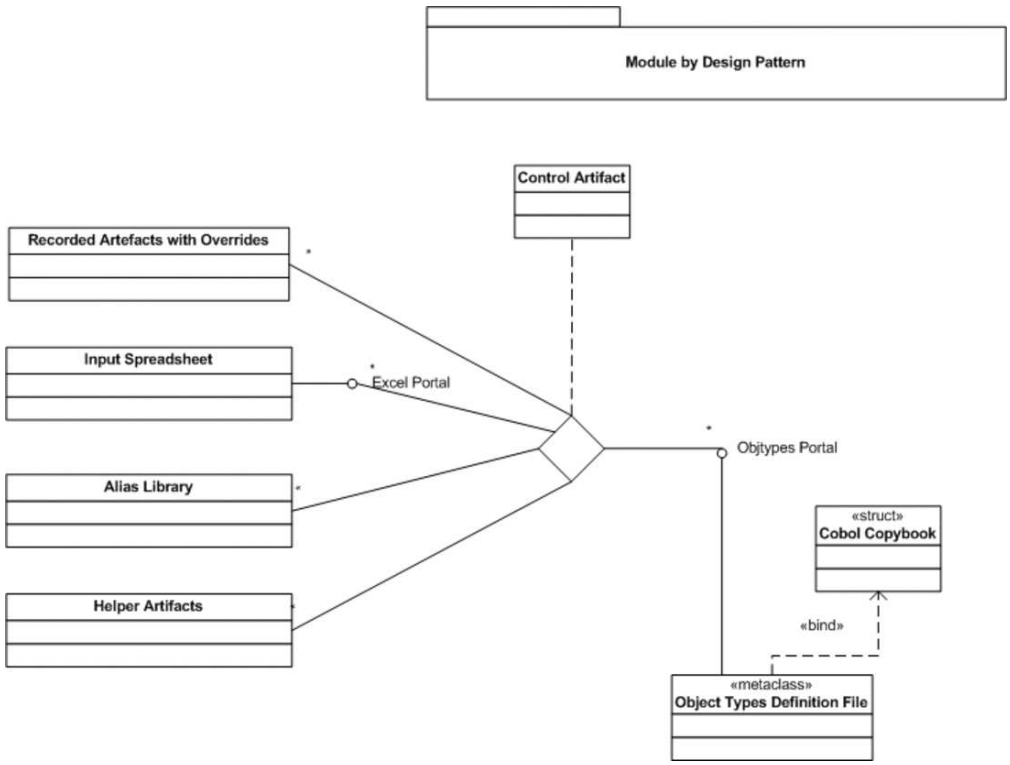


Figure 1: The Pattern Component Structure

A high level view of how the pattern components fit together can be seen in Figure 1.

8.1 Structure and Organization of Components

The illustration below shows the recommended convention to be used when arranging components into a logical structure, this ensures that all components are stored in consistent manner, which will aid end users and developers in locating the components as and when they are required.

```
01  System Root Directory (C:\)
02    |
03    |->CodeMagus
04    |      |
05    |      |->CVSModule
06    |           |
07    |           |-> XXXXFormats
08    |           |      \ testdata
09    |           |         |> configs
10    |           |         |> copybooks
11    |           |         |> objtypes
12    |           |
13    |           |-> XXXXPackage1
```

```
14    |           |      \ testdata
15    |           |         |> configs
16    |           |         |> data
17    |           |         |> scripts
18    |           |         |> spreadsheets
19    |           |
20    |           |-> XXXXPackage2
21    |           |      \ testdata
22    |           |         |> configs
23    |           |         |> data
24    |           |         |> scripts
25    |           |         |> spreadsheets
```

The system root directory shown on line 1 is the disk drive users wish to make use of for their development or testing environment, this is typically the `C:` in a Windows based environment.

The folder shown on line 3 is an anchoring folder which should be appropriately named in order to reflect a logical grouping of files and folders. The defined standard is to name the folder `CodeMagus`. This is appropriate since the files that will be contained within the structure will be files associated with the Code Magus Limited Tools.

The `CVSModule` shown on line 5 represents a grouping of files and folders under CVS control. A discussion on the purpose and function of CVS is deferred to subsequent sections. CVS Modules such as these are not likely to be created by users or developers, it will be the responsibility of a CVS Administrator to create and maintain access to CVS Modules as determined by the needs of the particular organization.

Within CVS Modules a distinction should be made between metadata and the artefacts or files that make use of that metadata. This distinction has to be made and they should always be maintained separately, each existing within their own structure. The reasoning behind this convention is that the relationship that exists is not two way. Whereas artefacts are determined by the metadata, the metadata is in no way determined by the artefacts. It is also common for metadata to be shared among different artefacts which further justifies this distinction.

The standard for naming folders which contain metadata should imply the specification as well as infer that the elements are metadata type elements. The agreed convention is to prefix the folder name with the associated specification or target system followed by the word `Formats`. In the above example this can be seen on line 7 where "`XXXX`" is the associated specification or system. The name must be expressed in camel case except in the case of an acronym which is always expressed in uppercase.

The `testdata` folder is a standard covering folder that must be present by convention. The `copybooks`, `objtypes` and `configs` folders are self explanatory since they will contain the relevant components. It is important that they are expressed as plural.

The artefacts that make use of the metadata, are grouped in separate folders and will represent an entire solution or package or several solutions or packages. The folders

which group the solutions together are named by implying some functionality associated with the underlying specification or system. In the above example the "`XXXX`" is the associated specification or system and should be followed by some implied functionality.

The `testdata` folder is a standard covering folder that must be present by convention. The `data`, `scripts` and `spreadsheets` will contain the appropriate components and must be expressed as plural.

8.2 Version Control

The version control system which will be used is CVS[2]. CVS allows several developers to work on one project at the same time, while keeping track of all changes to each source file.

Developers work on their own local file copies and can do so without producing conflicts when other developers are modifying the same files. A full discussion on CVS and all of the operations one can perform are beyond the scope of this course, although a basic introduction is necessary and will be sufficient for most users.

Figure 2 illustrates a typical CVS client/server configuration. It can be seen from the figure that both end users and developers alike will make use of CVS. The developers and end users will each typically work within there own repositories. A repository is a designated location on the CVS server which will contain and maintain all the required versioning information related to the files stored therein.

End users will also be granted differing levels of access to the files depending on their role within the business or development environment. For example, a business analyst might only have read access to a repository where as a developer would have write and read access. A CVS administrator would normally set up the required repositories and assign access to the end users.

8.2.1 Basic CVS Operation

The basic operations that an end user should be comfortable with in order to use CVS efficiently are the following:

- CVS Checkout - Retrieves modules or files from a particular repository on the CVS server.
- CVS Update - Retrieves the latest versions of files or modules already on a local machine from a repository on the CVS server.

[2]Concurrent Versioning System

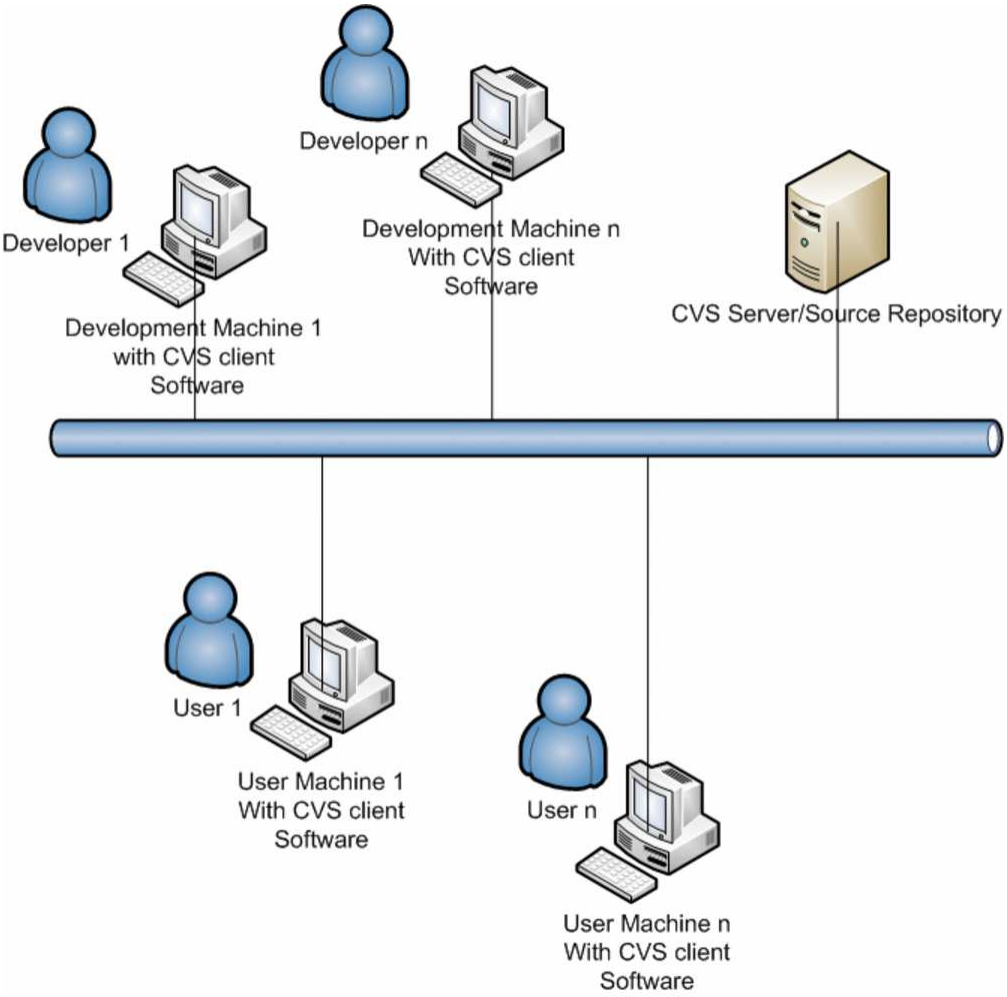


Figure 2: CVS Clients and Server

- CVS Add - Add files or modules to a particular repository on the CVS server. It is important to note that this operation does not upload the file to the server, it only schedules the file for addition. In order to successfully upload a file, a CVS commit should be performed after a CVS Add operation.
- CVS Commit - Stores the file, any associated comments and version history on CVS server.

The above listed operations are only a subset of all CVS operations, and it is advisable for users to consult the multitude of online resources in order to make better use of CVS.

9 Metadata

Metadata are a specific representation of groups of data presented in one form, (in this case a specification) in a medium which best conveys the information therein for the specific task at hand.

This data about data could be represented in so many ways that it would be impossible to discuss each underlying specification, however, the most relevant types of metadata pertaining to Eresia FIP, NIP and XIP Portals are the COBOL copybook, the XML Schema Definition Languate (XSD) and the Web Services Description Language (WSDL).

Throughout this course COBOL copybooks will be used as the underlying representation of a particular specification although a brief description of XSD and WSDL will be provided.

9.1 COBOL Copybooks

A concrete mechanism for specifying the metadata is required. This mechanism or type system could be a suitable proprietary system, but as you will see, as recurring theme in our toolkit, we abstract away from this concrete expression of the metadata requirements when come to actual use the metadata.

For example, such attributes of element representation (character set encoding, binary formats, endianness, etc) and location information (position and length) are not relevant when values are interchanged (assignment and reference) to and from conforming buffers, records and messages. In a similar manner we distance ourselves from the concrete representation showing that it could be one or more of any number of suitable representations.

But there has to be at least one representation and the industry we find ourselves in has a familiarity with the expression of types as described by the COBOL Standard.

This familiarity is often implicit and can be observed in the manner in which members of the financial industry describe types for elementary items (e.g. by expressions similar to the content of picture clauses). Hence our chosen concrete representation mechanism is that of COBOL with the container being the COBOL copybook.

Our abstraction away from this mechanism is delivery through our object types artefacts and it is to these artefacts that our tools connect.

Because of the popularity of COBOL in this industry there is vast collection of concrete metadata that is immediately available and that can be shared across organizations, and between various objectives within organizations such as development and testing.

Our approach allows us to hook into this collection and consume existing metadata without requiting it to be reworked or modified. It is our express desire to hook into pre-existing metadata wherever that it considered to be definitive rather than tweak copies for our own purposes irrespective of how tempting and easy this might be.

The reason we want to avoid this is that we cannot take over the responsibility of the authority of the metadata, and as soon as we make a modified copy it loses its authority.

Sometimes an authorative copybook might cover only a partial piece of a buffer, message or record that we would like to interpret or generate, with other pieces being described by other copybooks.

The COBOL copybook is then the first building block in the design patterns for flexible script solutions, this is because it is derived and must conform to some specification or layout.

It is assumed that the reader is comfortable with COBOL copybooks. This implies that the reader must understand the structure of a copybook, any COBOL syntax that is applicable, the data types that are available as well as the symbol extensions that have been provided.

Below is an example of a COBOL copybook using the symbol extensions derived from the record specification in the Code Magus Limited Proprietary Financial Record Layout[11] section 2.7, 2.8 and 2.9. The line numbers should be interpreted only as a guide and do not form part of the actual copybook.

```
0001  ****************************************************************
0002  *
0003  * Code Magus Limited.
0004  * Proprietry Data Record Layout.
0005  *
0006  * About: To be used with demonstrations and
0007  *        as a training tool.
0008  *
0009  * Copyright (c) 2008 Code Magus Limited. All rights reserved.
0010  * Contact: Stephen Donaldson [stephen@codemagus.com].
0011  *
0012  ****************************************************************
```

```
0013  *
0014  * $Author: justin $
0015  * $Date: 2010/03/10 07:07:38 $
0016  * $Id: samplebook.cpy,v 1.1 2010/03/10 07:07:38 justin Exp $
0017  * $Name:  $
0018  * $Revision: 1.1 $
0019  * $State: Exp $
0020  *
0021  ****************************************************************
0022  *
0023   01  CMLDATAR.
0024  *
0025       03  TRANSACTION-CODE                       PIC 9(4).
0026  *
0027  *begin attribute TRANSACTION-CODE["DESCRIPTION"].
0028  * Field: Transaction Code.
0029  *
0030  * The transaction code will be 1502 - Code Magus Data Record.
0031  *
0032  *end attribute.
0033  *
0034            88  DATA-RECORD                       VALUE 1502.
0035  *
0036       03  TRANSACTION-CLASSIFIER                 PIC 9(1).
0037  *
0038  *begin attribute TRANSACTION-CLASSIFIER ["DESCRIPTION"].
0039  * Field: Transaction Classifier.
0040  *
0041  * The class of the given transaction.
0042  *
0043  * Values     Description
0044  *
0045  *  1         Payment Record
0046  *  2         Repayment Record
0047  *  3         Reject Record
0048  *
0049  *end attribute.
0050  *
0051            88  PAYMENT-RECORD                    VALUE 1.
0052  *
0053            88  REPAYMENT-RECORD                  VALUE 2.
0054  *
0055            88  REJECT-RECORD                     VALUE 3.
0056  *
0057       03  PROCESS-TIME                            PIC 9(6)
0058                                          COLLATING IS ASCII.
0059  *
0060  *attribute set PROCESS-TIME["MASK"]="HHMMSS".
0061  *
0062  *begin attribute PROCESS-TIME["DESCRIPTION"].
0063  * Field: Process Time.
0064  *
```

```
0065  * The time at which the settlement file
0066  * was processed.
0067  *
0068  *end attribute.
0069  *
0070       03  PROCESS-DATE                          PIC 9(6)
0071                                        COLLATING IS ASCII.
0072  *
0073  *attribute set PROCESS-DATE["MASK"]="YYMMDD".
0074  *
0075  *begin attribute PROCESS-DATE["DESCRIPTION"].
0076  * Field: Process Date.
0077  *
0078  * The date on which the settlement file
0079  * was processed.
0080  *
0081  *end attribute.
0082  *
0083       03  CURRENCY-CODE                         PIC 9(3).
0084  *
0085  *begin attribute CURRENCY-CODE["DESCRIPTION"].
0086  * Field: Currency Code.
0087  *
0088  * A ISO three byte numeric currency code of the
0089  * transaction.
0090  *
0091  *end attribute.
0092  *
0093       03  TRANSACTION-AMOUNT                    PIC 9(15) COMP-3.
0094  *
0095  *begin attribute CURRENCY-CODE["DESCRIPTION"].
0096  * Field: Transaction Amount.
0097  *
0098  * The transaction amount.
0099  *
0100  *end attribute.
0101  *
0102       03  TRANSACTION-DECIMALIZATION            PIC 9(1).
0103  *
0104  *begin attribute TRANSACTION-DECIMALIZATION["DESCRIPTION"].
0105  * Field: Transaction Decimalization.
0106  *
0107  * The number of decimal places implied in the transaction
0108  * amount.
0109  *
0110  *end attribute.
0111  *
0112       03  MESSAGE-NUMBER                        PIC 9(15).
0113  *
0114  *begin attribute MESSAGE-NUMBER["DESCRIPTION"].
0115  * Field: Message Number.
0116  *
```

```
0117  * The sequential message number within the
0118  * current file.
0119  *
0120  *end attribute.
0121  *
0122       03  UNIQUE-ID                              PIC 9(6) COMP.
0123  *
0124  *begin attribute UNIQUE-ID["DESCRIPTION"].
0125  * Field: Unique Identifier.
0126  *
0127  * The unique identifier assigned by the terminal to
0128  * trace the record through the life cycle.
0129  *
0130  *end attribute.
0131  *
0132       03  ENCRYPTION-KEY                         PIC X(8) COMP-X.
0133  *
0134  *begin attribute ENCRYPTION-KEY["DESCRIPTION"].
0135  * Field: Encryption Key
0136  *
0137  * A public encryption key for the file.
0138  *
0139  *end attribute.
0140  *
0141       03  TERMINAL-DESCRIPTION-DATA.
0142  *
0143  *begin attribute TERMINAL-DESCRIPTION-DATA["DESCRIPTION"].
0144  * Field: Terminal Description Data.
0145  *
0146  * This field describes the device on which the
0147  * original transaction was performed.
0148  *
0149  *end attribute.
0150  *
0151            05  TERMINAL-ID                       PIC X(5).
0152  *
0153  *begin attribute TERMINAL-ID["DESCRIPTION"].
0154  * Field: Terminal Identifier.
0155  *
0156  * This field describes the institution assigned terminal
0157  * identifier on which the transaction was performed.
0158  *
0159  * This field may not be space filled or all zeroes.
0160  *
0161  *end attribute.
0162  *
0163            05  TERMINAL-TYPE                     PIC X(4).
0164  *
0165  *begin attribute TERMINAL-TYPE["DESCRIPTION"].
0166  * Field: Terminal Type.
0167  *
0168  * This field describes the type of terminal on which
```

```
0169  * the transaction was performed.
0170  *
0171  * Values     Description
0172  *
0173  * POSD       Point-of-Sale Terminal.
0174  * ATMD       ATM Machine.
0175  * 3270       3270 Terminal.
0176  *
0177  *end attribute.
0178  *
0179                88  POINT-OF-SALE                 VALUE "POSD".
0180  *
0181                88  ATM-MACHINE                   VALUE "ATMD".
0182  *
0183                88  3270-TERMINAL                 VALUE "3270".
0184  *
0185       03  MERCHANT-DESCRIPTION-DATA.
0186  *
0187  *begin attribute MERCHANT-DESCRIPTION-DATA["DESCRIPTION"].
0188  * Field: Merchant Description Data.
0189  *
0190  * This field describes the merchant at which
0191  * the original transaction was performed.
0192  *
0193  *end attribute.
0194  *
0195           05  MERCHANT-NAME                     PIC X(20).
0196  *
0197  *begin attribute MERCHANT-NAME["DESCRIPTION"].
0198  * Field: Merchant Name.
0199  *
0200  * This field contains the name of the merchant at which
0201  * transaction was performed.
0202  *
0203  *end attribute.
0204  *
0205           05  MERCHANT-CLASSIFIER                PIC X(8).
0206  *
0207  *begin attribute MERCHANT-CLASSIFIER["DESCRIPTION"].
0208  * Field: Merchant Classifier.
0209  *
0210  * This field describes the class of merchant at which
0211  * transaction was performed.
0212  *
0213  * Values     Description
0214  *
0215  * RESTRNT    Restaurant Merchant.
0216  * RETAILS    Retail Merchant.
0217  * AIRLINE    Airline Merchant.
0218  *
0219  *end attribute.
0220  *
```

```
0221                88  RESTAURANT-MERCHANT           VALUE "RESTRNT".
0222  *
0223                88  RETAIL-MERCHANT               VALUE "RETAILS".
0224  *
0225                88  AIRLINE-MERCHANT              VALUE "AIRLINE".
0226  *
0227           05  MERCHANT-ADDRESS.
0228  *
0229  *begin attribute MERCHANT-ADDRESS["DESCRIPTION"].
0230  * Field: Merchant Address.
0231  *
0232  * The physical address of the merchant.
0233  *
0234  * This field contains the following subfields:
0235  * Street Address
0236  * Suburb
0237  * Town/City
0238  * Postal Code.
0239  *
0240  *end attribute.
0241  *
0242               07  MERCH-STREET-ADDRESS          PIC X(27).
0243               07  MERCH-SUBURB                  PIC X(20).
0244               07  MERCH-TOWN                    PIC X(20).
0245               07  MERCH-POSTAL-CODE             PIC X(4).
0246       03   DEBIT-CREDIT-INDICATOR               PIC X(2).
0247  *
0248  *begin attribute DEBIT-CREDIT-INDICATOR["DESCRIPTION"].
0249  * Field: Debit or Credit Indicator.
0250  *
0251  * Indicates whether the net effect of the transaction is a
0252  * debit or credit. Valid Values are as follows:
0253  *
0254  * "DR" = Debit Transaction
0255  * "CR" = Credit Transaction
0256  *
0257  *end attribute.
0258  *
0259       03   LOCAL-INTERNATIONAL-FLAG             PIC X(13).
0260  *
0261  *begin attribute LOCAL-INTERNATIONAL-FLAG["DESCRIPTION"].
0262  * Field: Local or International Flag.
0263  * Flag indicating whether the transaction is local or
0264  * international. Valid Values are as follows:
0265  *
0266  * "INTERNATIONAL" = International Transaction
0267  * "DOMESTIC"      = Local or Domestic Transaction
0268  *
0269  *end attribute.
0270  *
0271            88  INTERNATIONAL-TRAN
0272                                          VALUE "INTERNATIONAL".
```

```
0273            88  LOCAL-DOMESTIC-TRAN
0274                                        VALUE "DOMESTIC".
0275       03   ACCOUNT-SCORE                          PIC +99v99.
0276  *
0277  *begin attribute ACCOUNT-SCORE["DESCRIPTION"].
0278  * Field: Account Score.
0279  *
0280  * A score assigned by crediting agencies to determine the
0281  * credit worthiness of the account holder. The format of
0282  * this field is as follows:
0283  *
0284  * * The first byte of this field must contain a "+" or "-".
0285  * * The next two bytes of this field must contain an integer
0286  *   between 01 and 99.
0287  * * The next part of this field must contain a floating point.
0288  * * The last two bytes of this field must contain an integer
0289  *   between 00 and 99.
0290  *
0291  *end attribute.
0292  *
0293       03   POINTS-SCORE                          PIC +99v99.
0294  *
0295  *begin attribute POINTS-SCORE["DESCRIPTION"].
0296  * Field: Points Score.
0297  *
0298  * A score assigned by banking institution to determine the
0299  * credit worthiness of the account holder. The format of
0300  * this field is as follows:
0301  *
0302  * * The first byte of this field must contain a "+" or "-".
0303  * * The next two bytes of this field must contain an integer
0304  *   between 01 and 99.
0305  * * The next part of this field must contain a floating point.
0306  * * The last two bytes of this field must contain an integer
0307  *   between 00 and 99.
0308  *
0309  *end attribute.
0310  *
0311       03  ACCOUNT-POINT-RATIO                    PIC +99v99.
0312  *
0313  *begin attribute ACCOUNT-POINT-RATIO["DESCRIPTION"].
0314  * Field: Account Point Ratio.
0315
0316  * This field is the ratio between fields 14 and 15, which
0317  * represents the ratio between the banking and credit agency
0318  * assigned score to determine the overall credit worthiness
0319  * of the account holder.
0320  *
0321  * * The first byte of this field must contain a "+" or "-".
0322  * * The next two bytes of this field must contain an integer
0323  *   between 01 and 99.
0324  * * The next part of this field must contain a floating point.
```

```
0325  * * The last two bytes of this field must contain an integer
0326  *   between 00 and 99.
0327  *
0328  *end attribute.
0329  *
```

9.2 XML

EXtended Markup Language or XML is a simple text based format derived from SGML as a recommendation from the World Wide Web Consortium (W3C), It is in widespread use today as a mechanism for describing and sharing data. XML documents consist of tags and attributes that describe the data between the tags and the XML documents themselves conform to a much stricter set of syntax rules unlike some other well known markup languages such as HTML.

Many people that are initially presented with XML try to draw comparisons between XML and HTML since HTML is a very well known markup language. They are however used for very different purposes, whereas HTML is a markup language with predefined tags and attributes intended for specifying how data is to be displayed, XML on the other hand a self describing document with user-defined tags and attributes that conveys information about the data it contains.

9.3 XML Schema Definition Languate (XSD)

XML Schema definition language provides or XSD is a means of defining classes of XML documents. XSD's imply a structure, set of rules and any constraints that developers may wish to convey in their XML documents. XSD's are written in xml which makes it easier for developers familiar with XML to learn and by implication makes them easier to read.

XML Schemas provide support for data types, this means that an XML schema may constrain the types that given data may take on. This makes it a simpler task for developers and their applications to validate the data and perform data conversion routines.

An XML document is said to conform to a particular XSD if it structurally matches the requirements layed out in the XML Schema and the data conforms to any associated or implied rule.

9.4 Web Services Description Language (WSDL)

Web Services Description Language (WSDL) is an xml-based format developed as a collaboration between IBM, Ariba and Microsoft. WSDL is used to describe networked XML-based services in straight-forward manner such that the messages or requests and

responses are defined ... so that the detail of the underlying protocol and message encoding need not be a factor at the point of definition.

WSDL provides developers with a standard that can be used when defining services. The standard brings together the definition of the individual messages as well as a means of grouping these messages together into sets of related services.

XML Schema's are used in the WSDL to provide the low-level data-typing required for the message content. Data-type specifications may also be imported where they are stored as seperate resources as would be the case where the messages are of a complex nature.

WSDL port types are the means by which messages are grouped together to form a single logical operations. WSDL supports unidirectional (one-way) and bidirectional (two-way) port types.

WSDL is said to bind together the physical (data typing) and the logical (messages and port types), ie. the binding provides the connection between the physical and the logical as well as describes the transmission details (SOAP,HTTP,MIME).

WSDL will also define the comminutcation end-point or physical location of the service, which is simply a Web address or URI.

9.5 Types Collections

As type systems go, the COBOL copybook is not as complete as it needs to be for our purposes. We use our object types mechanism to augment this information to complete the required typing for the preparation and interpretation of buffers, records and messages.

For example, in a COBOL system there is an implicit assumption that certain representations are those of the computer system hosting the COBOL system.

Such attributes include the character set representation, the binding of items to memory, the endianness of binary items and the sign conventions of certain numeric items.

Because our tools need to interpret and prepare test data on one platform destined for or between other platforms in which different assumptions are made, this missing information has to be supplied.

These additional attributes are supplied in our object types artefacts. Object types also allow the laying out of various copybooks and the picking and choosing of portions of those copybooks as required to describe a particular buffer, message or record.

Finally, object types are a useful mechanism of naming a type buffer, message or record and for supplying a description to the type, together with a predicate over the contents of the buffer, message or record, which is required to evaluate to true over its contents in order to identify that item as belonging to the named type.

The object types definition file below is based on the record specification in the Code Magus Limited Proprietary Financial Record Layout section 2.7, 2.8 and 2.9. The line numbers should be interpreted only as a guide and do not form part of the actual object types definition file.

```
0001
0002 path "C:/CodeMagus/CodeMagus/CMLFormats/testdata/copybooks/%s.cpy";
0003 options ebcdic, omit_fillers;
0004
0005 ------------------------------------------------------------------------------
0006 -- Code Magus Limited.
0007 -- Proprietary Object Types.
0008 --
0009 -- About: To be used with demonstrations and
0010 --        as a training tool.
0011 --
0012 -- Copyright (c) 2008 Code Magus Limited. All rights reserved.
0013 -- Contact: Stephen Donaldson [stephen@codemagus.com].
0014 --
0015 ------------------------------------------------------------------------------
0016
0017 type CML_PROP_DATA_RECORD_PAYMENT
0018   title "Code Magus Limited: Proprietary Data Record Layout - Payment"
0019   book CMLDMDAT
0020   map CMLDATAR
0021      include CMLDATAR
0022      when (CMLDATAR.TRANSACTION_CODE = 1502)
0023       and (CMLDATAR.TRANSACTION_CLASSIFIER = 1);
0024
```

A given object types definition file comprises two sections, namely the Preamble statements and the type list.

Lines 2 and 3 make up the preamble statements which introduce the global options and attributes which cannot be inferred from the metadata artefacts which are referenced from the type definitions.

The `path` statement in line 2 is the fully qualified path to the location where the metadata resides on the local machine.

The `options` statement on line 3 defines the attributes that cannot be inferred from the metadata, or for which the local machine attributes do not suffice.

Lines 5 through 15 illustrate the use of inline comments which will always start with `--`.

From line 17 to 23 illustrates a type list containing only one type.

The Object types artefacts form a central role in the design pattern for flexible script solutions. The identification of messages or records through the use of object types artefacts is essential to the function of the design pattern for flexible script solutions

since this functionality is key to the pattern and will be used extensively throughout the creation and implementation of the various pattern components.

10 Canned Data Collections

The Code Magus Limited Tools, Eresia NIP[5], Eresia FIP[4] and Eresia XIP[**?**] are specialist tools that provide powerful functionality whose aim is to analyse and manipulate canned collections of data.

These canned collections of data may be in the form of logs, files or xml messages which are in one way or another the inputs and outputs of the systems that will typically require test packages.

Canned data collections provide dual functionality in that as well as being a basis for new records or messages can also provide aid in checking the integrity of the defined metadata for a particular system.

Canned data collections are obtained from a variety of sources, the data might be provided by software vendors as an aid to testing, messages might be saved to a log which could be read, or the data may be sanitized production data that may be re-used.

It is important to note that it is not important how this data is acquired as long as it is an accurate representation of the specification.

Section 11 describes a means of tying the metadata, the object type definition files and these canned data collections together in order to make sense of as well as constructively use all these components together to create runnable and modifiable artefacts that will form core components of the pattern in question.

11 Workspaces

Workspaces are powerful tools provided by Code Magus Limited and extend the functionality provided by `objtypes`[3] by allowing end users to perform a variety of operations on files and logs, which will all be thoroughly examined throughout this section.

The portals which currently have workspaces available to end users are the Eresia Network Injection Portal, the Eresia File Injection Portal and the Eresia XML Injection Portal.

Although the workspaces all provide similar interfaces with common set of operations, it is worth looking at them independently.

12 The Eresia NIP Workspace

The Eresia Network Injection Portal (Eresia NIP) provides end users with the ability to perform several functions on collections of data in the form of log files.

These files are typically collections of typed network messages which can be analysed, manipulated and resent over a network.

Once an Eresia NIP workspace has been configured to read a particular set of network messages, the end users will have access to all available information pertaining to the message collection.

Provided the metadata for a given set of network messages is correct the Eresia NIP will automatically interpret and type all the network messages in a particular log.

End users will be able to view all the messages contained in the log as well as all the fields contained in the individual messages. Any available descriptions pertaining to the individual fields may be viewed at any time, along with any available attributes and masks. The messages and fields may also be examined at the buffer level in the form of buffer dumps.

The Eresia NIP also provides the user with an extensible query language in order to facilitate queries over these collections of messages.

13 The Eresia FIP Workspace

The Eresia File Injection Portal (Eresia FIP) provides end users with the ability to perform several functions on collections of data in the form of files.

These files are typically collections of typed records which can be analysed, manipulated and reorganized into new files.

Once an Eresia FIP workspace has been configured to read a particular kind of file, the end users will have access to all available information pertaining to the particular file.

Provided the metadata for a given file is correct the Eresia FIP will automatically interpret and type all the records in a particular file.

End users will be able to view all the records contained in the file as well as all the fields contained in the individual records. Any available descriptions pertaining to the individual fields may be viewed at any time, along with any available attributes and masks. The records and fields may also be examined at the buffer level in the form of buffer dumps.

The Eresia FIP also provides the user with an extensible query language in order to facilitate queries over these collections of records.

14 The Eresia XIP Workspace

The Eresia XML Injection Portal (Eresia XIP) provides end users with the ability to perform several functions on collections of data in the form of XML documents.

Once an Eresia XIP workspace has been configured to read the XML documents, the end users will have access to all available information related to the contained XML documents.

End users will be able to view all the individual XML records contained in the collection as well as all the fields and attributes that are defined.

The Eresia XIP also provides the user with an extensible query language in order to facilitate queries over these collections of XML documents.

15 Recording Example Using The Eresia FIP Workspace

The Eresia NIP, Eresia FIP and the Eresia XIP all provide recording interfaces which work in a manner similar to one another and whose outputs are runnable thistle artefacts for use with the Eresia Visual Test Environment[6].

These runnable thistle artefacts form the basis for the default value artefacts which are discussed in Section 16. The recorded artefacts will undergo a series of manipulations in order to prime them for use as default value artefacts.

This section provides all the details and guidelines necessary for obtaining a recorded thistle script through the configuration of the pattern components discussed thus far into an Eresia FIP workspace.

15.1 Workspace Elements

The File Injection Portal's workspace is essentially a tree structure with the `File Portal Workspace` element or node as it's base or root, the tree structure is a convenient way of displaying the workspace and showing the relationships between the various nodes.

The `File Portal Workspace` node has it's own menu which can be accessed by hovering the mouse over the node and right clicking. This action will bring up a pop-up menu with a `new` menu item. This is seen in the Figure 3.

When the `new` menu item is highlighted it will extend the menu to include the following options:

- Candidate Data File
- Object Types Definition File

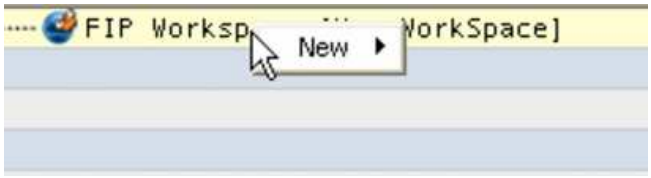


Figure 3: To Select Options on a New Workspace

- Environment Variables
- Description
- Query Collection
- Worksheet Collection

This is illustrated in Figure 4. All the listed options create child nodes with differing functions belonging to the `File Portal Workspace` node. The child nodes may in turn contain their own child nodes or may have values assigned to the nodes. Each of the options will be examined the sections to follow.

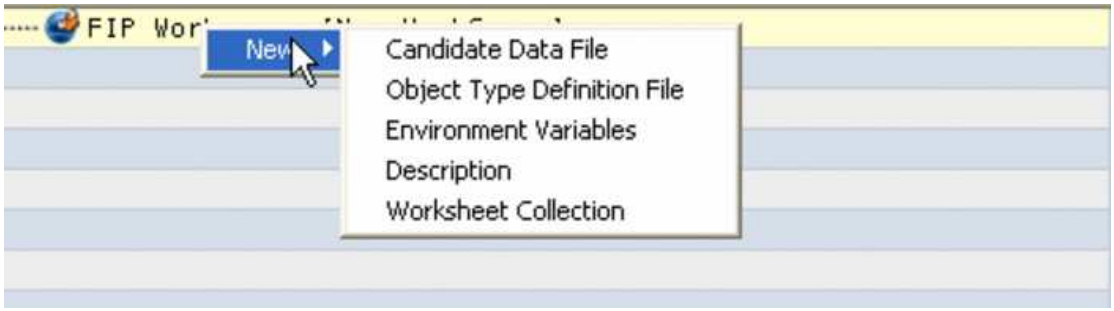


Figure 4: Viewing Available Options on a New Workspace

15.2 Adding Object Type Definition Files to a Workspace

It is assumed at this point that the reader is familiar with Object Types and their role in identifying and recognizing the objects to which they are applied.

The Object Types file that is opened will be applied to all collections of data within the particular workspace.

Only one Object Types file will be permitted in any workspace at any one time, the Object Types file may however be replaced with a new Object Types file at any time.

To add the Object Types element or node, the `Object Type Definition File` option needs to selected from the `File Portal Workspace` node's menu. The resulting `Object Type Definition File` node will added as a child node under the `File Portal Workspace` node as shown in Figure 5.

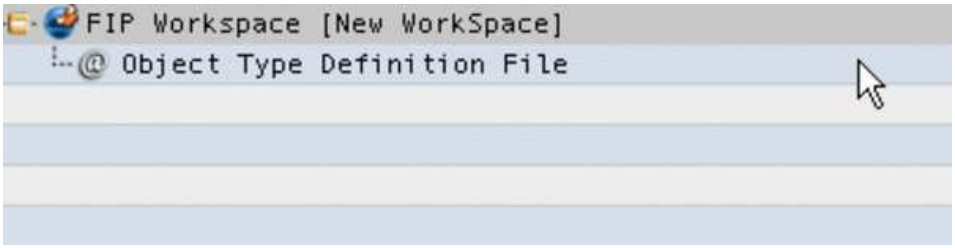


Figure 5: A New `Object Type Definition` File Node

Once the `Object Type Definition File` node has been added to the workspace the value cell (the cell opposite the node) will become active and the user may now double click the value cell in order to enable the browse button. The browse button is as shown in Figure 6, it is the grey button in the value cell containing ellipses.

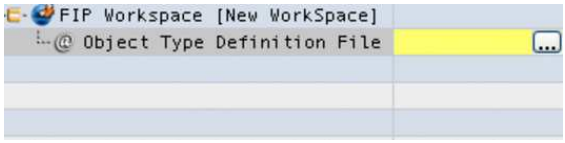


Figure 6: How to Browse to an Object Types Definition File

When an end user clicks on the browse button the `Open` window will appear allowing the user to navigate to and select the required Object Types file for the current workspace, this can be seen in Figure 7. Once the Object Types file has been successfully opened, the path to the Object Type file and the Object Type filename will be displaced in the value cell. In addition to this the `Record Type Collection` node will be created which is discussed in Section 15.2.1.

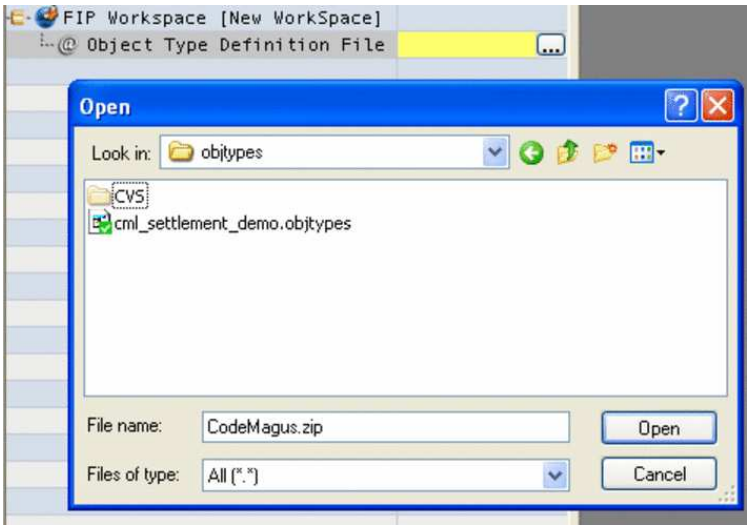


Figure 7: Browsing to an Object Types Definition File

15.2.1 The `Record Type Collection`

The `Record Type Collection` is a node that will be created after an Object Types file has been successfully loaded into a particular workspace. The `Record Type Collection` node has a direct relationship with the Object Types file and contains and gives access to all types that have been defined in the corresponding Object Types file.

In addition to this the `Record Type Collection` node provides direct access to the fields contained in the types defined in the Object Types file. The end user will generally use the `Record Type Collection` node as a reference to the types in the Object Types file and the fields contained in each of the corresponding types. Users will be able to view the various attributes of the fields, such as masks, descriptions and picture clauses. In addition to this users will be able to export all the field names to a token separated file.

An expanded `Record Type Collection` node can be seen in Figure 8. The `Record Type Collection` node can be expanded by clicking on the "plus" sign on the immediate left of the node. Every type defined in the Object Types file will be listed under the `Record Type Collection` node.

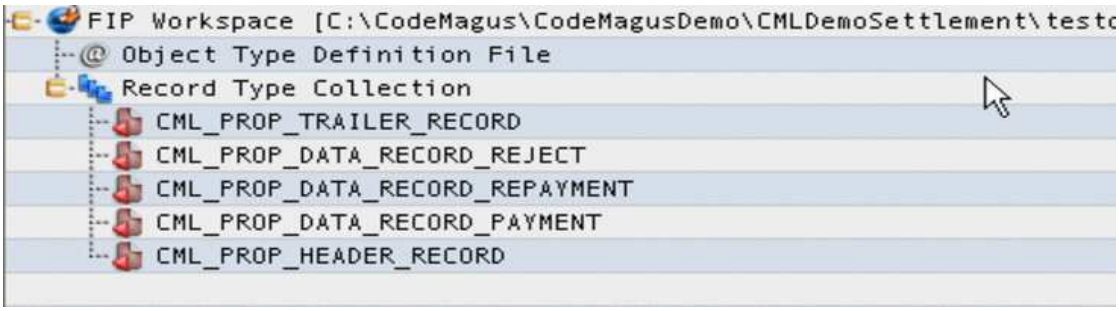


Figure 8: The Record Collection

15.3 Opening Files in a Workspace

The `Candidate Data File` node facilitates the opening and reading of a particular file, and makes the file available to the File Injection Portal. Users may then perform various analysis tasks on the file. Users may inspect the file visually, check that the file layout is as required and that the fields within the records align to the metadata that defines them.

The users may also check whether the file contains as many records as are expected to be present and perform queries on the file to meet any additional requirements. This section describes the addition of a single `Candidate Data File`[3] node to the workspace as well as setting up an `Open Spec` and `Record Collection` and illustrating the options available to the user for analysing the records within the `Record Collection`.

15.3.1 Adding a `Candidate Data File`

A new `Candidate Data File` node can be added by selecting the `Candidate Data File` menu item from the `File Portal Workspace` node's menu. This is illustrated in Figure 9 .

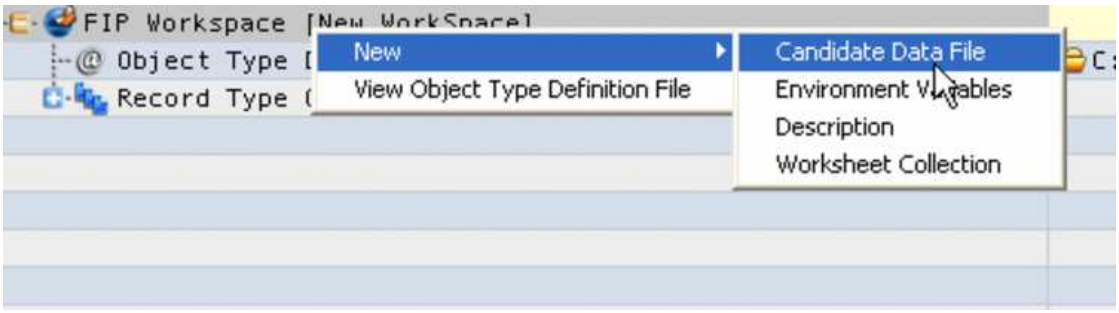


Figure 9: Selection of a New Candidate Data File.

[3] A given workspace may contain as many `Candidate Data File` nodes as required.

The result will be an empty `Candidate Data File` node as shown in Figure 10. In the figure the `Candidate Data File` node has been expanded to illustrate the empty `Record Collection`[4] node that has been added as a result of adding the `Candidate Data File` node, there is still some configuration required via the `Open Spec` to allow a file to be read in to the `Candidate Data File`'s `Record Collection`.

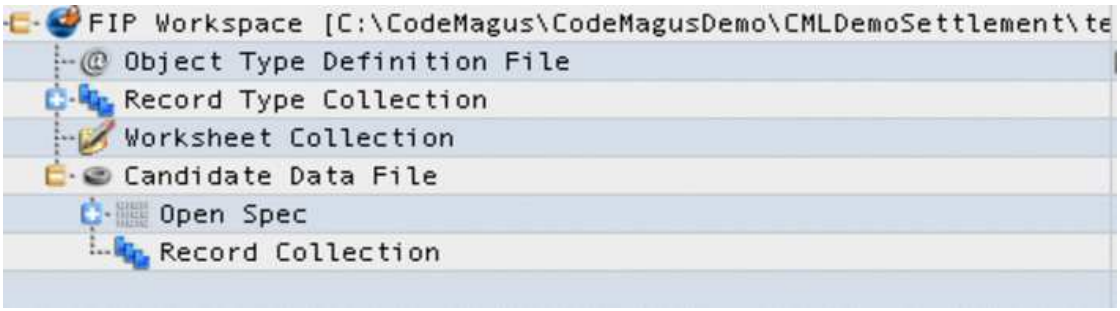


Figure 10: An Empty Candidate Data File

15.3.2 The `Candidate Data File` and Available Operations

By right-clicking on the `Candidate Data File` node a pop-up menu will appear with the menu items `New` and `Delete`. The `Delete` menu item will when selected remove the `Candidate Data File` node from the workspace. The `Candidate Data File` node's menu items are as shown in Figure 11.

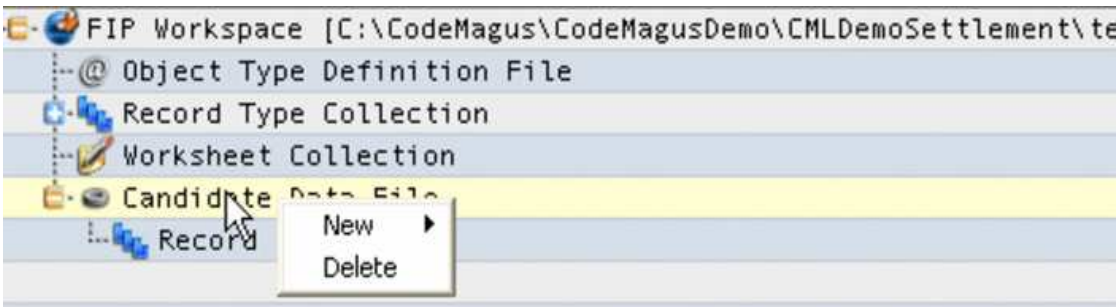


Figure 11: Options Available via the Candidate Data File

By highlighting the `New` menu item on the `Candidate Data File` node's pop-up menu a further two menu items will appear, namely, `Open Spec` and `Query Collection`. The document Function and Operation of the Eresia File Injection Portal may be referred to should the reader not be familiar with these constructs.

[4] The `Record Collection` as discussed here must be differentiated from the `Record Type Collection` that appears as a result of successfully loading the Object Types file. The `Record Collection` is a result of the `Record Types Collection` being applied over a given collection of data for interpretation.

15.3.3 Reading Files with an Access Method Open Specification

The `Open Spec` will however be discussed briefly in this section in order to examine a populated `Record Collection`. Selection of the `Open Spec` menu item is as shown in Figure 12.

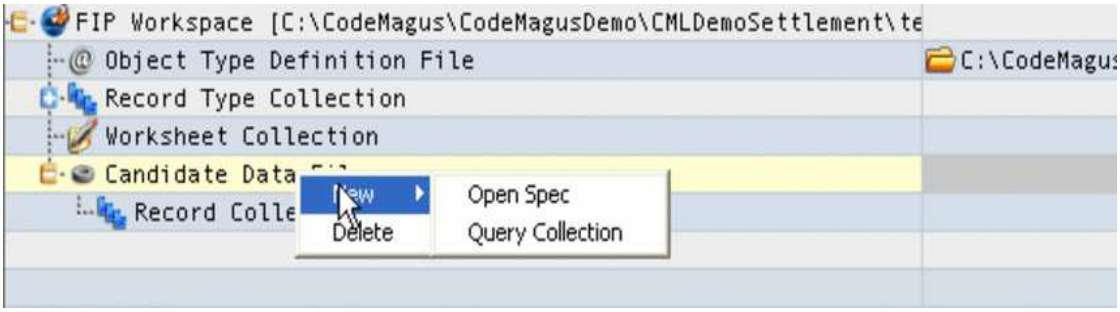


Figure 12: Selection of the Open Spec under a Candidate Data File

An expanded `Open Spec` node is as shown in Figure 13, it can be seen that there are three nodes which will need to be populated in order for a file to be loaded as part of the `Record Collection`, they are:

- `Access Method` - The name of the access method required to open the file.
- `Object` - The name of the object or file to be opened.
- `Options` - The options that the access method requires in order to open the file.

The `Access Method`[2] , and the `Options` value cells will receive input by double-clicking on the cell and typing in the appropriate text for the required access method. The `Object`'s value cell can be populated by double-clicking and selecting the browse button in order to browse to the object or file that is required to populate the `Record Collection`.

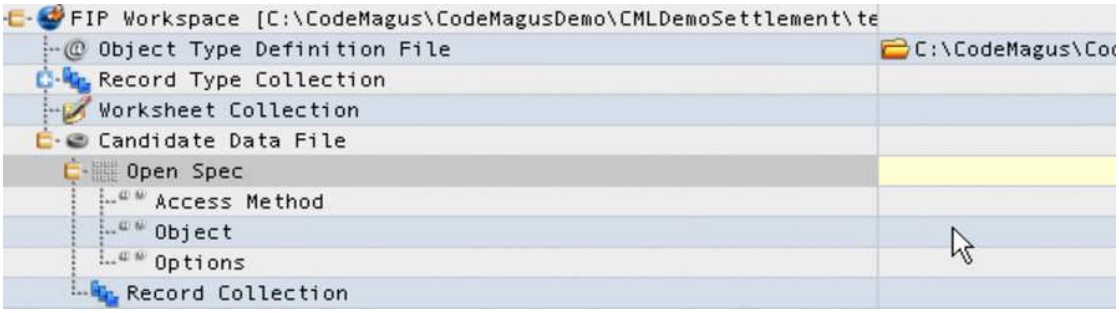


Figure 13: An Empty Open Spec under a Candidate Data File

In Figure 14, the `Access Method`, and the `Options` value cells have been populated with the relevant access method and options required for the access method to open the file or object.

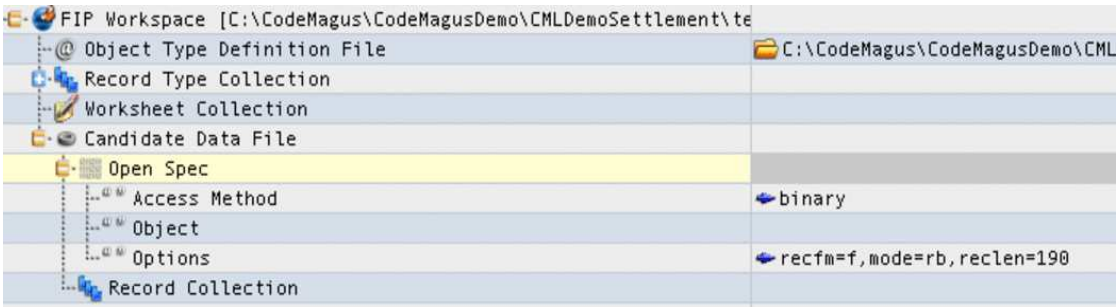


Figure 14: Population of Fields in the Open Spec

Figure 15 illustrates how the `Objects` value cell is populated by browse to the required file or object that is to populate the `Record Collection`.

Once the object or file has been selected and successfully opened, the `Record Collection` will be populated as shown in Figure 16. It can be seen from the figure that the records

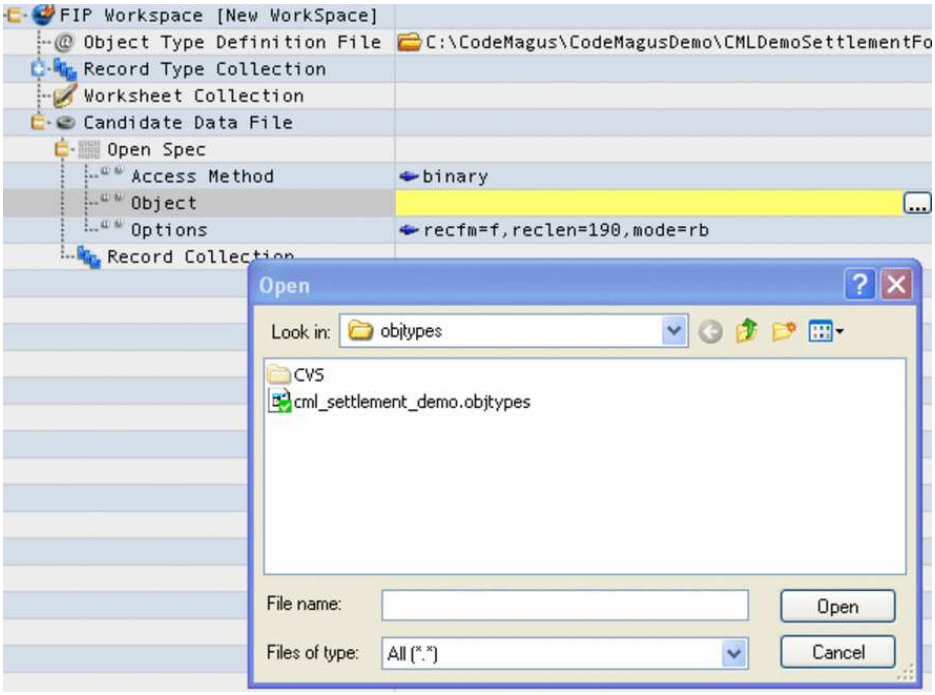


Figure 15: Browsing to an Object for the Open Spec

in the `Record Collection` look very much like the Object Types that appear in the `Record Types Collection`.

The reason for this is that as the data is provided to the `Record Collection`, the Object Types that were provided to the workspace will be applied to the data. The records will therefore map to the appropriate Object Type and the data can now be easily interpreted and analysed.

15.4 Creating Worksheets for Manipulation, Recording and Saving of Records and Files

In the preceding text it has been shown that through the `Candidate Data File`, large collections of records from files or objects can be read into a `Record Collection` and analysed either at the record level or the field level. At the same time having access to the buffers that make up the records, the properties and attributes that define the fields and the fields and values themselves provide the end user with an invaluable means of access to the contents of these files or objects.

The ability to analyse and interpret very large collection of data at a micro level adds immeasurable value to the end user. This functionality alone could make up an entire toolset, however, the need often arises to be able to reuse the files or objects under analysis. Worksheets solve this problem by providing various mechanisms to reuse

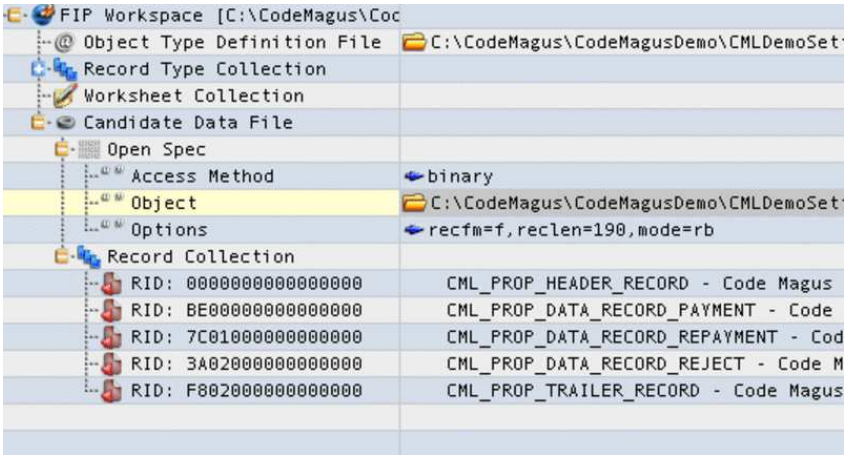


Figure 16: Record Collection Successfully Opened

existing files or objects to create new solutions as well as new files or objects.

Worksheets provide end users with a means to manipulate customized collections of records, save records to file, perform queries on collections of records and also provides an interface to the Eresia Visual Test Environment which allows users to record runnable thistle scripts based on the records in a particular worksheet.

Worksheets provide the end user with direct access to records within a worksheet record collection, in addition to having all the attributes and properties available within a record collection, records in worksheets can be edited.

There may be as many worksheets within a worksheet collection as are required. The worksheet nodes may be renamed, although the names may contain no spaces or special characters.

15.4.1 The `Worksheet Collection`

In order to add worksheets, a `Worksheet Collection` node will need to added to the workspace, which as the name implies, is a collection of worksheets. The `Worksheet Collection` node can be added by right-clicking the `File Portal Workspace` node in order to access it's pop-up menu and selecting the `Worksheet Collection` menu item. Figure 17 illustrates the selection of a new `Worksheet Collection`.

The result of selecting the new `Worksheet Collection` menu item from the `File Portal Workspace` node will be a `Worksheet Collection` node that will be added to the workspace as seen in Figure 18.

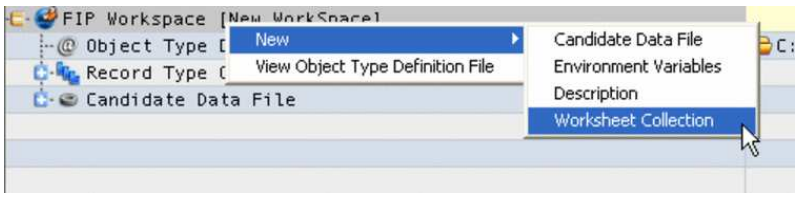


Figure 17: Creating a new Worksheet Collection

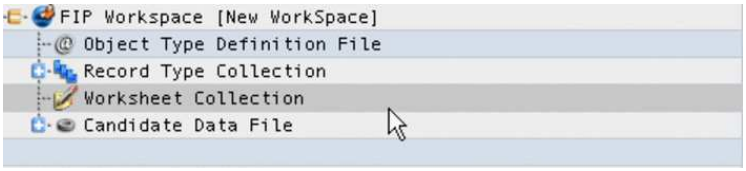


Figure 18: The New Worksheet Collection

15.4.2 The `Worksheet Collection`

The `Worksheet Collection`'s pop-menu which can be accessed by right-clicking on the node is shown in Figure 19 with the `New` menu item expanded. If the `Delete` menu item is selected the `Worksheet Collection` node will be removed from the workspace.

The are two other menu items available when the `New` menu item is expanded, namely the `Default Open Spec` and the `Work Sheet` menu items.

15.4.3 The `Work Sheet` Menu Item

The `Work Sheet` menu item will create a `worksheet` node under the `Worksheet Collection` node which can be configured for the various functions that may be performed using worksheets.

15.4.4 The `Default Open Spec` Menu Item

The `Default Open Spec` menu item will create the `Default Open Spec` node under the `Worksheet Collection` node. The `Default Open Spec` has precisely the same interface as any other `Open Spec` node, the difference with this particular open specification is the manner in which it is applied, in that the `Default Open Spec` as well as any other `Open Spec` with the exception of the `Output Open Spec` are used only when recording. They do not open the file for reading as is the case with the `Candidate Data File` but are written to the Thistle script when recording takes place. This will become clear in Section 15.5

As will be seen when the individual `Work Sheet`'s are dealt with, they too have open specifications associated with them. A worksheet may have an `Open Spec` and an `Output Open Spec` defined, but need not.

Therefore, the `Default Open Spec` will come into play should a worksheet not have an `Open Spec` defined. This allows end users to not have to create a new `Open Spec` for each new worksheet provided they have a `Default Open Spec` defined.

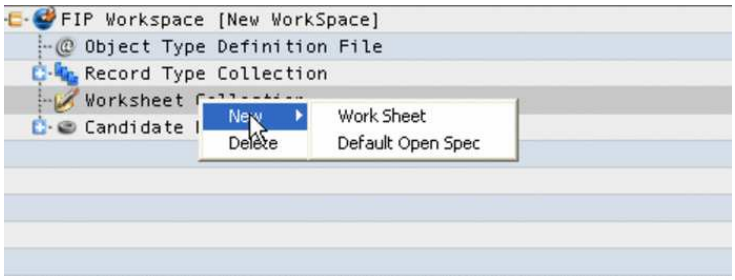


Figure 19: Menu items available for a `Worksheet Collection`.

Figure 20 shows the workspace after both the `Default Open Spec` and `Work Sheet` menu items have been selected and the input fields of the `Default Open Spec` have been populated.

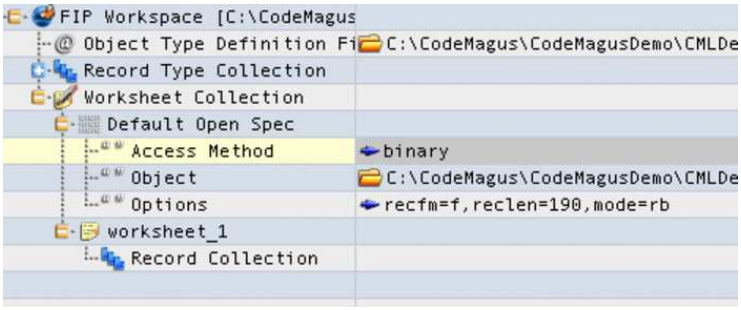


Figure 20: `WorkSheet Collection` node with a `worksheet` and `Default Open Specs`.

15.5 Recording from a Worksheet

In order to record using the Eresia FIP, the Eresia Visual Test Environment and the Eresia FIP must to be connected to one another. There are two methods of establishing the communication between the Eresia FIP and the Eresia Visual Test Environment.

15.5.1 Establishing a Connection between the Eresia FIP and the Eresia Visual Test Environment

There exists two possible routes one may follow in order to establish the connection between the Eresia FIP and the Eresia Visual Test Environment, both are involve the same process but it is rather the order in which things are done which differs.

The first scenario, which is by far the most common is that the Eresia Visual Test Environment will need to be open. The user will then select the record button located on the task bar in the Eresia Visual Test Environment. This action will bring up the `Usecase Preamble` dialog box.

The `Usecase Preamble` dialog box contains the following fields which will need to be populated.

- `Usecase Identifier` - The internal name of the usecase or script that is to be recorded. When the usecase is saved it must be saved as this name.
- `Name` - The name of the end user recording the usecase.
- `Portal Type` - The `Eresia File Portal` option must be selected from this drop down list.
- `Description` - The description or intent of the recorded usecase.

The result of supplying values for the fore mentioned fields can be seen in the following extract of a recorded Thistle script.

```
01 usecase CML_DEMO_FILE_HEADER(
02   typespath := "C:\\CodeMagus\\Types\\cml_settlement_demo.objtypes",
03   openstring := "binary(C:\\FILE.bin,recfm=f,mode=rb,reclen=190)");
04
05 { preamble }
06
07   created by 'Developer Name';
08   description 'The Code Magus Limited file header.';
09   date 2009-04-02T18:04:34;
10   target 'Eresia File Portal';
11   interface Portal.recio : CodeMagus.RECIO;
12   interface Portal.Types : CodeMagus.Types;
13
14 begin
15 ...
```

In line 1 the `Usecase Identifier` value from the `Usecase Preamble` dialog is added with the `usecase` prefix. Line 2 shows the name entered in the `Name` value from the `Usecase Preamble`. Line 8 shows the `Description` value from the `Usecase Preamble` and line 10 uses the value from the `Portal Type` field.

It is worthwhile to note that the value from the `Open Spec` or `Default Open Spec` will be provided to an `openstring` parameter as seen in line 3, and the value from the `Object Type Definition File` node is given to the `typespath` parameter as in line 2.

Once the fields have been populated and the `OK` button is pressed, the Eresia Visual Test Environment and the Eresia FIP will be connected and recording can proceed.

In the second scenario there is only a slight difference in that the Eresia FIP is already open. All that will need to be done is that the Eresia Visual Test Environment must be started and the record button pressed, once the end user has completed populating the `Usecase Preamble`, the Eresia Visual Test Environment will automatically connect to the open Eresia FIP.

When the Eresia Visual Test Environment and the Eresia FIP have been connected, the window state in the bottom right hand corner of the Eresia FIP will change from `Connectionless` to `Portal Client -> Record`.

15.5.2 Copying Records into a Worksheet

In Section 15.4 the process of adding a `Worksheet Collection`, a `Default Open Spec` and a new `worksheet` was demonstrated. It is now possible to discuss what is required to populate a given `worksheet`.

In Figure 21 it can be seen that the `Worksheet Collection` has a `Default Open Spec` have been populated as previously discussed, in addition to that the `Candidate Data File` contains a collection of records read in from a file. It can be seen from the figure that the `worksheet` also contains a `Record Collection` node. The `Record Collection` node in the `worksheet` will be the target for the records copied from the `Record Collection` under the `Candidate Data File` node.

In order to copy a particular record from the `Record Collection` under the `Candidate Data File` node, the record must be selected by clicking on the record with the mouse pointer, the row in the grid will be highlighted in order for the end user to see which record has been selected. The user will then press CTRL-C[5], select the `Record Collection` node in the `worksheet` which will be the destination of the copied record and then press CTRL-V[6] to copy the record into the `Record Collection`.

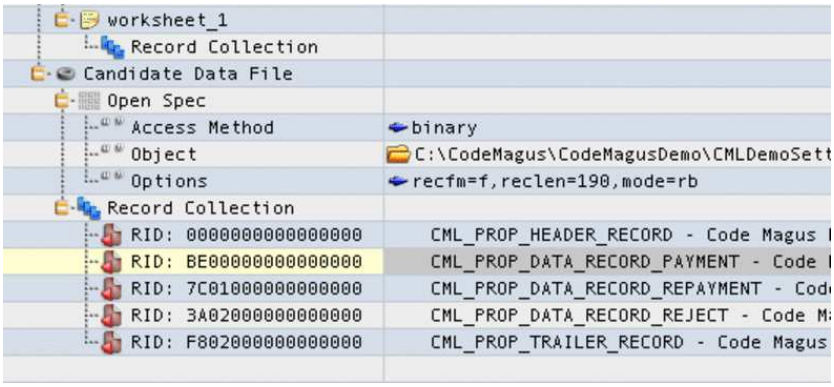


Figure 21: Preparing to copy a Record from a Candidate Data File to a Worksheet

The result is of copying the record to the `worksheet`'s `Record Collection` node is shown in Figure 22. The `RID`[7] of the record in the `worksheet`'s `Record Collection` node is prefixed with `Copy of`, the record is therefore an entirely new record, may be modified, in addition to having all the usual properties and attributes of records opened in the Eresia FIP.

At this point the Eresia FIP and the Eresia Visual Test Environment will be connected to one another and the `worksheet` will contain the record that is to be recorded. In order to now do the recording of the Thistle Script, the end user must right-click on the `worksheet` node, and the additional menu item `Record` will now be present. The is illustrated in Figure 23.

After the selection of the `Record` menu item, the end user will be able to switch focus to the Eresia Visual Test Environment and view and save the recorded artefact. When the user has finished recording the stop button in the Eresia Visual Test Environment may be selected and the portals will disconnect.

[5]The Control Key and the character C key pressed at the same time to perform a copy function.
[6]The Control Key and the character V key pressed at the same time to perform a paste function.
[7]Record Identifier

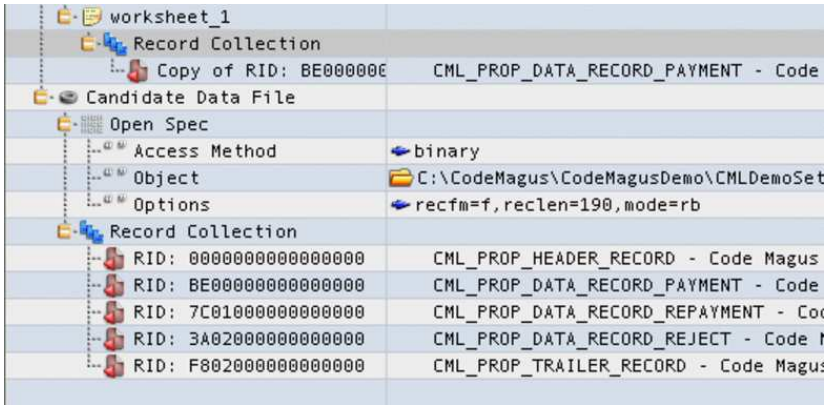


Figure 22: The Record Copied in the Worksheet.

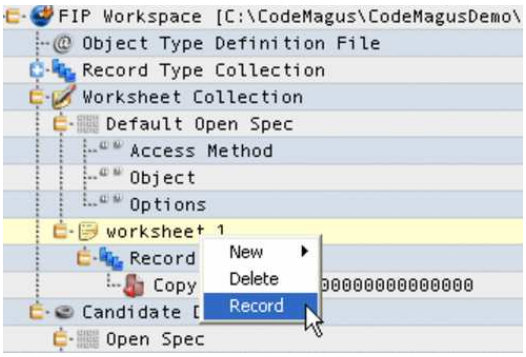


Figure 23: The Options on Worksheet with the Record Option Enabled

The recorded artefact will be written to the Eresia Visual Test Environment and will layed out similarly to the following recorded artefact.

```
01 usecase CMLS_DATA_RECORD_PAYMENT(typespath := "C:\\CodeMagus\\cmlds.objtypes",
02   openstring := "binary(C:\\CodeMagus\\data\\CMLS_SETTLEMENT_DEFAULT_VALUES.bin,
03                       recfm=f,mode=wb,reclen=190)");
04
05 { preamble }
06
07   created by 'Developer Name';
08   description 'Code Magus Limited: Proprietary Data Record Layout - Payment';
09   date 2009-03-03T07:01:13;
10   target 'Eresia File Portal';
11   interface Portal.recio : CodeMagus.RECIO;
12   interface Portal.Types : CodeMagus.Types;
13
14 begin
15
16   types := Portal.Types.Connect(typespath);
17   stream := Portal.recio.Connect();
18   OutputStream := stream.open(open_string:=openstring,
19                               mode:="SEQ_OUTPUT",flags:="/VERBOSE");
20   with rec do begin
21      with DATA_RECORD_PAYMENT do begin
22         with CMLDATAR do begin
23           TRANSACTION_CODE                := 1502;
24           TRANSACTION_CLASSIFIER          := 1;
25           PROCESS_TIME                    := 103001;
26           PROCESS_DATE                    := 090210;
27           CURRENCY_CODE                   := 710;
28           TRANSACTION_AMOUNT              := 000000000099999;
29           TRANSACTION_DECIMALIZATION      := 2;
30           MESSAGE_NUMBER                  := 000000000000100;
31           UNIQUE_ID                       := 123457;
32           ENCRYPTION_KEY                  := "A04522";
33           with TERMINAL_DESCRIPTION_DATA do begin
34             TERMINAL_ID                   := "P1234";
35             TERMINAL_TYPE                 := "POSD";
36           end
37           with MERCHANT_DESCRIPTION_DATA do begin
38             MERCHANT_NAME                 := "MARIOS PIZZARIA     ";
39             MERCHANT_CLASSIFIER           := "RESTRNT";
40             with MERCHANT_ADDRESS do begin
41               MERCH_STREET_ADDRESS        := "25 HUNDREDTH AVE    ";
42               MERCH_SUBURB                := "SUMMER VALLEY       ";
43               MERCH_TOWN                  := "CALIFORNIA          ";
44               MERCH_POSTAL_CODE           := "1955";
45             end
46           end
47            DEBIT_CREDIT_INDICATOR := "DR";
48            LOCAL_INTERNATIONAL_FLAG := "INTERNATIONAL";
49            ACCOUNT_SCORE := +21.32;
50            POINTS_SCORE := +21.32;
```

```
51            ACCOUNT_POINT_RATIO := +21.32;
52         end
53      end
54   end
55   Buf:=types.GetBuffer(rec.CML_DATA_RECORD_PAYMENT,"CML_DATA_RECORD_PAYMENT");
56   stream.write(stream:=OutputStream,buffer:=Buf);
57   delete record;
58 end.
```

The end user may not close either the Eresia Visual Test Environment or the Eresia FIP while they are connected, and if this is attempted a message will appear indicating that the portal is still in record mode.

16 Default Value Artefacts

```
usecase CMLS_HEADER_RECORD(typespath := "C:\\CodeMagus\\cmlds.objtypes",
   openstring := "binary(C:\\CodeMagus\\data\\CMLS_SETTLEMENT_DEFAULT_VALUES.bin,
                        recfm=f,mode=wb,reclen=190)");

{ preamble }

   created by 'Developer Name';
   description 'File Header based on the Code Magus Limited Proprietary \'
      \'Specification.';
   date 2009-03-03T07:01:13;
   target 'Eresia File Portal';
   interface Portal.recio : CodeMagus.RECIO;
   interface Portal.Types : CodeMagus.Types;

begin

   types := Portal.Types.Connect(typespath);
   stream := Portal.recio.Connect();
   OutputStream := stream.open(open_string:=openstring,
                               mode:="SEQ_OUTPUT",flags:="/VERBOSE");
   with record do begin
      with CML_PROP_HEADER_RECORD do begin
         with CMLHEADR do begin
            TRANSACTION_CODE := 1501;
            BATCH_NUMBER := 0001;
            PROCESS_TIME := 091406;
            PROCESS_DATE := 090227;
            MESSAGE_NUMBER := 000000000000001;
            PROCESSING_SYSTEM := "CODE MAGUS PROCESSING SYSTEM  ";
            FILE_SUBMITTED_BY := "DR SR DONALDSON 011 500 2323                    "
            SUBMISSION_TIME := 091406;
            SUBMISSION_DATE := 090227;
            SOFTWARE_VERSION := "Proprietary Financial Record Layout Version 1";
            SOFTWARE_VENDOR := "Code Magus Limited";
```

```
        end
      end
   end
   OutputBuffer := types.GetBuffer(record.CML_PROP_HEADER_RECORD,
                                                "CML_PROP_HEADER_RECORD");
   stream.write(stream:=OutputStream,buffer:=OutputBuffer);

   delete record;

end.
```

The recorded thistle scripts that have been looked at thus far, are a replayable representation of particular records or messages within a file or log, this representation is however static at this point and in order to make the recorded scripts dynamic artefacts that will be useful going forward, certain elements within the record will need to be modified in order to make them re-runnable units that are not only an accurate representation of the data but also valid simulations of real world scenarios.

By using the extensions provided by Thistle, providing overrides to the recorded artefacts can be done programmatically, which will then give the end users the ability to focus on creating meaningful test cases without having to concern themselves about the validity of the underlying data.

It is noteworthy to mention that the end users will too be able to provide there own set of data to the records or messages by means of an input spreadsheet which is discussed in detail in Section 18.

Identifying the fields that are candidates for overrides depends largely on the specification that defines the records or messages, and it will be up to the developer to select suitable candidates for this process.

The following guidelines can be used in identifying elements they may be considered for override candidates:

1. Elements that are constrained by certain values.
2. Elements that have functional dependencies on tasks to be performed.
3. Elements that have functional dependencies on other elements.
4. Elements whose default values have functional dependencies on other elements, ie. conditional functional dependencies.

Consider the following recorded artefact.

```
01 usecase CMLS_DATA_RECORD_PAYMENT(typespath := "C:\\CodeMagus\\cmlds.objtypes",
02   openstring := "binary(C:\\CodeMagus\\data\\CMLS_SETTLEMENT_DEFAULT_VALUES.bin,
03                         recfm=f,mode=wb,reclen=190)");
04
05 { preamble }
06
07   created by 'Developer Name';
```

```
08   description 'Code Magus Limited: Proprietary Data Record Layout - Payment';
09   date 2009-03-03T07:01:13;
10   target 'Eresia File Portal';
11   interface Portal.recio : CodeMagus.RECIO;
12   interface Portal.Types : CodeMagus.Types;
13
14 begin
15
16   types := Portal.Types.Connect(typespath);
17   stream := Portal.recio.Connect();
18   OutputStream := stream.open(open_string:=openstring,
19                               mode:="SEQ_OUTPUT",flags:="/VERBOSE");
20   with rec do begin
21      with DATA_RECORD_PAYMENT do begin
22         with CMLDATAR do begin
23           TRANSACTION_CODE                := 1502;
24           TRANSACTION_CLASSIFIER          := 1;
25           PROCESS_TIME                    := 103001;
26           PROCESS_DATE                    := 090210;
27           CURRENCY_CODE                   := 710;
28           TRANSACTION_AMOUNT              := 000000000099999;
29           TRANSACTION_DECIMALIZATION      := 2;
30           MESSAGE_NUMBER                  := 000000000000100;
31           UNIQUE_ID                       := 123457;
32           ENCRYPTION_KEY                  := "A04522";
33           with TERMINAL_DESCRIPTION_DATA do begin
34             TERMINAL_ID                   := "P1234";
35             TERMINAL_TYPE                 := "POSD";
36           end
37           with MERCHANT_DESCRIPTION_DATA do begin
38             MERCHANT_NAME                 := "MARIOS PIZZARIA     ";
39             MERCHANT_CLASSIFIER           := "RESTRNT";
40             with MERCHANT_ADDRESS do begin
41               MERCH_STREET_ADDRESS        := "25 HUNDREDTH AVE    ";
42               MERCH_SUBURB                := "SUMMER VALLEY       ";
43               MERCH_TOWN                  := "CALIFORNIA          ";
44               MERCH_POSTAL_CODE           := "1955";
45             end
46           end
47            DEBIT_CREDIT_INDICATOR := "DR";
48            LOCAL_INTERNATIONAL_FLAG := "INTERNATIONAL";
49            ACCOUNT_SCORE := +21.32;
50            POINTS_SCORE := +21.32;
51            ACCOUNT_POINT_RATIO := +21.32;
52         end
53      end
54   end
55   Buf:=types.GetBuffer(rec.CML_DATA_RECORD_PAYMENT,"CML_DATA_RECORD_PAYMENT");
56   stream.write(stream:=OutputStream,buffer:=Buf);
57   delete record;
58 end.
```

Before we begin to override the necessary values, some modifications to the script are

required. The reason for this is that recorded artefacts give end users a ready to run, standalone script, for the purposes of the Eresia patterns it is desirable to leave processing such as interface definition, typing and file i/o or sending a message over the network to the caller, we will remove these processes from the recorded artefact leaving only a structure which constitutes the message which will be returned to the caller. The discussion on how the caller achieves this will be deferred to Section 21.

The modifications can be achieved by the following steps:

1. Remove the parameter declaration entirely, which is the bracketed text on lines 01 through 03, the brackets should be removed as well.
2. Remove any interfaces from the preamble, which would be lines 11 and 12 in the example above.
3. Remove the initialisation of the interfaces, since there are no longer any interfaces to initialise. In the example above this would be lines 16 and 17.
4. Remove any calls to the interfaces. In the example above this would lines 18, 19, 55, 56.
5. Remove the outermost scope created by objtypes when recording. This scope will get re-added at a later stage before the structure gets passed to objtypes.
6. Lastly remove the `delete record;` statement on line 57, and replace the statement with a return of the now top most scope's node, ie `DATA_RECORD_PAYMENT`, since it is required to pass the structure back to the caller.

The above mentioned steps will apply to any recorded Eresia NIP or FIP scripts. After we have trimmed the recorded artefact as described what will be left is the following;

We are now in a position to examine which elements are suitable candidates for overrides, let us assume that we are developing a package with the following requirements:

- The field TERMINAL_TYPE must have the value "POSD".
- The fields PROCESS_TIME and PROCESS_DATE must be equal to the time and date of the package run.
- The UNIQUE_ID must contain the same value as the PROCESS_TIME to ensure a unique value within a run.
- If the CURRENCY_CODE is 710 then the LOCAL_INTERNATIONAL_FLAG must be "DOMESTIC".

It is now possible to construct our default values based on these requirements, which can be translated into Thistle using the guidelines mentioned earlier in this section. The overrides section would be as follows:

```
00
01   {CodeMagus Overrides}
```

```
02   { In each case, the over-rides should be kept separate so that they are easy
03     to re-apply. }
04      with DATA_RECORD_PAYMENT do begin
05         with CMLDATAR do begin
06           with TERMINAL_DESCRIPTION_DATA do begin
07             TERMINAL_TYPE                  := "POSD";
08           end
09           PROCESS_TIME := System.TimeFormat(System.TimeCurrent(),"HHMMSS");
10           PROCESS_DATE := System.DateFormat(System.DateCurrent(),"YYMMDD");
11           UNIQUE_ID := PROCESS_TIME;
12           CURRENCY_CODE := 710;
13           LOCAL_INTERNATIONAL_FLAG := "DOMESTIC     ";
14         end
15      end
```

One may notice the similarities between the modified recorded artefact and the over-rides, In that the top two scopes are exactly the same, but only the overridden fields are present. The reason for this is that this section of code will be placed in the same usecase in which the recorded message was modified to conform to the Eresia Patterns framework. The artefact is now ready to be packaged along with all the other components, which is discussed in Section 21.

```
01 usecase CMLS_DATA_RECORD_PAYMENT;
02
03 { preamble }
04
05   created by 'Developer Name';
06   description 'Code Magus Limited: Proprietary Data Record Layout - Payment';
07   date 2007-03-16T12:20:10;
08   target 'Eresia File Portal';
09
10 begin
11      with DATA_RECORD_PAYMENT do begin
12         with CMLDATAR do begin
13           TRANSACTION_CODE                := 1502;
14           TRANSACTION_CLASSIFIER          := 1;
15           PROCESS_TIME                    := 103001;
16           PROCESS_DATE                    := 090210;
17           CURRENCY_CODE                   := 810;
18           TRANSACTION_AMOUNT              := 000000000099999;
19           TRANSACTION_DECIMALIZATION      := 2;
20           MESSAGE_NUMBER                  := 000000000000100;
21           UNIQUE_ID                       := 123457;
22           ENCRYPTION_KEY                  := "A04522";
23           with TERMINAL_DESCRIPTION_DATA do begin
24             TERMINAL_ID                   := "P1234";
25             TERMINAL_TYPE                 := "POSD";
26           end
27           with MERCHANT_DESCRIPTION_DATA do begin
28             MERCHANT_NAME                 := "MARIOS PIZZARIA     ";
29             MERCHANT_CLASSIFIER           := "RESTRNT";
30             with MERCHANT_ADDRESS do begin
```

```
31              MERCH_STREET_ADDRESS          := "25 HUNDREDTH AVE    ";
32              MERCH_SUBURB                  := "SUMMER VALLEY       ";
33              MERCH_TOWN                    := "CALIFORNIA          ";
34              MERCH_POSTAL_CODE             := "1955";
35            end
36          end
37           DEBIT_CREDIT_INDICATOR := "DR";
38           LOCAL_INTERNATIONAL_FLAG := "INTERNATIONAL";
39           ACCOUNT_SCORE := +21.32;
40           POINTS_SCORE := +21.32;
41           ACCOUNT_POINT_RATIO := +21.32;
42        end
43     end
44
45
46   {CodeMagus Overrides}
47   { In each case, the over-rides should be kept separate so that they are easy
48     to re-apply. }
49     with DATA_RECORD_PAYMENT do begin
50        with CMLDATAR do begin
51          with TERMINAL_DESCRIPTION_DATA do begin
52            TERMINAL_TYPE                 := "POSD";
53          end
54          PROCESS_TIME := System.TimeFormat(System.TimeCurrent(),"HHMMSS");
55          PROCESS_DATE := System.DateFormat(System.DateCurrent(),"YYMMDD");
56          UNIQUE_ID := PROCESS_TIME;
57          CURRENCY_CODE := 710;
58          LOCAL_INTERNATIONAL_FLAG := "DOMESTIC     ";
59        end
60     end
61
62   return DATA_RECORD_PAYMENT;
63 end.
64
```

It can be seen from the finished artefact that the overrides are not standalone but rather compliment the modified recorded artefact. At this point you may ask yourself why not just modify the values in the modified recorded artefact? The answer lies in the fact that the software development life cycle is never static, requirements may change and the specification may be modified from time to time.

By applying the overrides in this manner saves crucial time when upgrades are required.. Instead of starting at the very beginning and re-applying the work done thus far, all that will need to be done is that the artefact will need to be re-recorded and placed in the appropriate section, the artefact will then be inline with the new specification, requirement or update and all overrides collected up to that point will be preserved.

17 The Alias Library

When referring to elements in thistle, particularly when the qualification path is very long, a convenient mechanism is required that will allow users to reference only the element required and not the fully qualified string.

From a scripting point of view, fields must be qualified in order for the execution environment to be able to identify which field is currently being referring to, the use of the alias library creates a suitable mechanism in which the qualification of strings is abstracted away from the end user.

Consider the following extracts of two recorded scripts:

```
with MERCHANT do begin
  with ADDR_INFO do begin
    with POSTAL_ADDRESS do begin
      STREET_ADDRESS := "21 Smith Street";
      SUBURB := "Kensington";
      TOWN_OR_CITY := "Potchefstroom";
      POSTAL_CODE := "0254";
    end
  end
end
return MERCHANT_DATA;
```

and

```
with CUSTOMER do begin
  with ADDR_INFO do begin
    with POSTAL_ADDRESS do begin
      STREET_ADDRESS := "10 Milton Ave";
      SUBURB := "Douglasdale";
      TOWN_OR_CITY := "Sandton";
      POSTAL_CODE := "2488";
    end
  end
end
return CUSTOMER_DATA;
```

Suppose we created the following package in order to manipulate the customer and merchant address data for file processing, we currently would have to type out the fully qualified string to access the `STREET_ADDRESS` node, as illustrated below:

```
package CreateAddressFile;
{ preamble }

   created by 'Developer Name';
   description 'Manipulate data to create an Address File';
   date 2008-06-30T09:31:28;
   target 'Eresia File Portal';
   usecase   MerchData  : DataScripts.MERCHANT_DATA;
   usecase   CustData   : DataScripts.CUSTOMER_DATA;
```

```
begin
  MerchStruct := MerchData();
  CustStruct  := CustData();

  PostToMerchantStr := MerchStruct.ADDR_INFO.POSTAL_ADDRESS.STREET_ADDRESS;
  PostToCustStr     := CustStruct.ADDR_INFO.POSTAL_ADDRESS.STREET_ADDRESS;
end
```

This is clearly not the correct approach since the developer or tester would have to know the fully qualified string each time a variable somewhere in the tree needed to be accessed.

The correct approach would be to somehow be able to access just the bottom most element (ie. `STREET_ADDRESS`, but how would this be possible, since the Eresia Visual Test Environment would not be able to differentiate between whether we are dealing with a customer's or a merchant's `STREET_ADDRESS`.

The answer lies in the Alias Library. In the library we will create structures of the form:

We therefore would now have direct access to the fully qualified string and the only information required to make this possible is the entry point, ie `MERCHANT` or `CUSTOMER` which would typically be a record type, but need not be, as in the case below:

```
01  package GETADDRESS;
02  { preamble }
03
04    created by 'Developer Name';
05    description 'Manipulate data to create an Address File';
06    date 2008-06-30T09:31:28;
07    target 'Eresia File Portal';
08    library AddrLib    : CMLSFilesScripts.AddressLib;
09    usecase MerchData  : CMLSFilesScripts.MERCHANT;
10    usecase CustData   : CMLSFilesScripts.CUSTOMER;
11
12 begin
13   MerchStruct := MerchData();
14   CustStruct  := CustData();
15
16   MerchMap    := AddrLib.MerchantAddressMap();
17   CustMap     := AddrLib.CustomerAddressMap();
18
19   PostToMerchantStr := MerchStruct.[MerchMap['MERCHANT']['STREET_ADDRESS']];
20   PostToCustStr     := CustStruct.[CustMap['CUSTOMER']['STREET_ADDRESS']];
21 end.
```

18 The Input Spreadsheet

From an end users perspective the input spreadsheet is the unit that will be responsible for supplying user specific input to be used with the other pattern elements in order to satisfy the requirements set out by the module.

The input spreadsheet allows the end user to have complete control over the executing package. The end user has the ability to populate any field in any record or message that has been delivered as part of the solution.

The fields within the records or messages can be populated in any order, and also allows the records to be ordered arbitrarily. This flexibility makes for very versatile and robust test cases.

The input spreadsheets are reusable and in this way it is possible to build up regression test packs and reusable functional testing packs.

Each input spreadsheet can be divided into two sections, namely the control section and an input section.

18.1 The Control Section

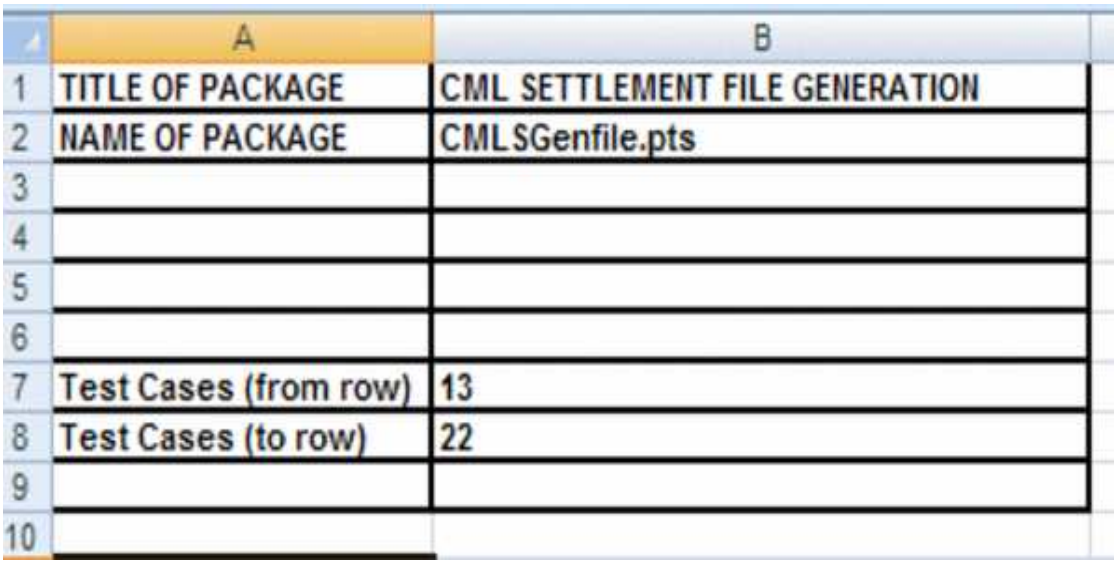
The control section of the input spreadsheet provides the range of scenarios that will be processed within a particular package execution where the input spreadsheet is used as the input source.

The control section of the spreadsheet should look identicle for most, if not all implementations of the pattern, since any other global information should be made available through the use of the applparms interface which is discussed in Section 20 and is described fully in the Application Parameters Library User Guide and Reference[**?**].

An example of the control section can be seen in Figure 24.

In the Figure 24, it can be seen that there are four populated entries in column B that map to corresponding descriptive text in column A. The entries in column B that map to the descriptions in column A are as follows:

1. `TITLE OF PACKAGE` - A short high level description of the package.
2. `NAME OF PACKAGE` - The Name of the thistle package that references the spreadsheet.
3. `Test Cases (From Row)` - The starting row in the spreadsheet that will be fed to the script suite.
4. `Test Cases (To Row)` - The ending row in the spreadsheet that will be fed to the script suite.



	A	B
1	TITLE OF PACKAGE	CML SETTLEMENT FILE GENERATION
2	NAME OF PACKAGE	CMLSGenfile.pts
3		
4		
5		
6		
7	Test Cases (from row)	13
8	Test Cases (to row)	22
9		
10		

Figure 24: The Control Section of the FIP Input Spreadsheet

Items 1 and 2 described above do not typically get modified by the user and exist mainly for information purposes. Items 3 and 4 are determined by the users processing requirements.

18.2 The Input Section

The input section of the input spreadsheet provides the user with direct control over the type of data and the contents thereof that will be processed in a given run. The Input Section can itself be broken down into the following:

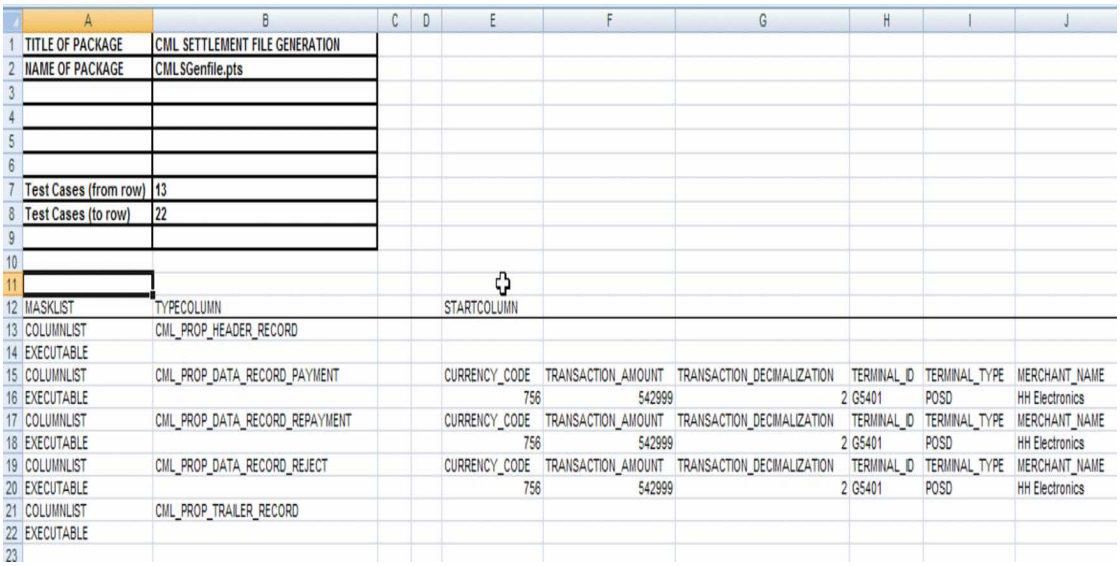
- The Standard Headings
- The User Space

18.3 The Standard Headings

As can be seen in Figure 25, row 13 in the figure contains the following 3 standard headings:

- `MASKLIST`
- `TYPECOLUMN`
- `STARTCOLUMN`

The above mentioned headings are processed by the GetHeaders usecase in the common scripts. At this point it is sufficient to note the GetHeaders usecase gathers the required



	A	B	C	D	E	F	G	H	I	J
1	TITLE OF PACKAGE	CML SETTLEMENT FILE GENERATION								
2	NAME OF PACKAGE	CMLSGenfile.pts								
3										
4										
5										
6										
7	Test Cases (from row)	13								
8	Test Cases (to row)	22								
9										
10										
11										
12	MASKLIST	TYPECOLUMN			STARTCOLUMN					
13	COLUMNLIST	CML_PROP_HEADER_RECORD								
14	EXECUTABLE									
15	COLUMNLIST	CML_PROP_DATA_RECORD_PAYMENT			CURRENCY_CODE	TRANSACTION_AMOUNT	TRANSACTION_DECIMALIZATION	TERMINAL_ID	TERMINAL_TYPE	MERCHANT_NAME
16	EXECUTABLE				756	542999	2	G5401	POSD	HH Electronics
17	COLUMNLIST	CML_PROP_DATA_RECORD_REPAYMENT			CURRENCY_CODE	TRANSACTION_AMOUNT	TRANSACTION_DECIMALIZATION	TERMINAL_ID	TERMINAL_TYPE	MERCHANT_NAME
18	EXECUTABLE				756	542999	2	G5401	POSD	HH Electronics
19	COLUMNLIST	CML_PROP_DATA_RECORD_REJECT			CURRENCY_CODE	TRANSACTION_AMOUNT	TRANSACTION_DECIMALIZATION	TERMINAL_ID	TERMINAL_TYPE	MERCHANT_NAME
20	EXECUTABLE				756	542999	2	G5401	POSD	HH Electronics
21	COLUMNLIST	CML_PROP_TRAILER_RECORD								
22	EXECUTABLE									
23										

Figure 25: The Input Section of the FIP Input Spreadsheet

heading information and returns it to the package when required. The GetHeaders use-case is discussed in detail in the section 19.

It is important that one is aware of the constraints on each of the available standard headings and the function signified by each. We shall first examine the constraints placed on each of the standard headings and then describe each in detail.

The `MASKLIST` heading has the constraint that it must appear in column A and must also appear after the control section of the spreadsheet. The `TYPECOLUMN` has the constraint that it must be placed in the same row as the `MASKLIST` heading. The `STARTCOLUMN` is constrained in the fact that it must appear in the same row as the `MASKLIST` and must appear after the `STARTCOLUMN`.

The `MASKLIST`, `TYPECOLUMN` and `STARTCOLUMN` describe what can be expected in the rows to follow under each of the respective headings.

In the rows that follow, the `MASKLIST` column will always contain one of three values, namely:

- `COLUMNLIST` - Indicates a row that will contain the type of record or message that will be created in subsequent `EXECUTABLE` rows and the fields that are to be populated in the record or message.
- `EXECUTABLE` - Indicates a row that will be processed according to the definitions set out in the `COLUMNLIST`.
- `INCOMPLETE` - Indicates a row that will not be processed.

The `TYPECOLUMN` defines the column in which desired type of record or message must be specified, a type can however only be specified when a row has been defined as a

COLUMNLIST.

The STARTCOLUMN anchors the column in which we will begin listing our field names to be overwritten in the case where the row has been defined as a COLUMNLIST or anchors the column where we will begin inputting data into the required cells where the row has been defined as EXECUTABLE.

18.4 The User Space

As the name implies, the user space is where the end users will configure the data to be sent or created as per their specific requirements.

Since the user space is not static in any way, ie. it will be different from package to package and from case to case, it is therefore important to understand how end users would typically configure the user space to meet their needs.

As described in the previous section the MASKLIST column will always contain either COLUMNLIST, EXECUTABLE or INCOMPLETE. These elements describe the function of the row, ie whether a row is a declaration, an executable unit or to be ignored.

In Figure 25 a typical example is provided, where the rows 13, 15, 17, 19, 21, 23 are COLUMNLIST or declaration type rows, rows 14, 16, 20, 22, 24 are EXECUTABLE type rows or executable units and row 18 is an INCOMPLETE or ignored row.

It can be seen that executable units always follow declaration type rows since they work together in order to map the correct data to the corresponding fields of a particular type. It is important to not at this point that executable units are dependant on there being a declaration type row present and that there is a many to one mapping from executable units to declaration type rows. In other words, there may be more than one executable unit to any one declaration type row.

We can now go a step further and analyse what comprises each of the various types of rows that make up the user space, each of them namely, Declaration Type Rows, Executable Units and Ignored Rows are examined in the remainder of this section.

18.5 Declaration Type Rows

Starting with the COLUMNLIST or declaration type row, we can see from row 14 that only the COLUMNLIST is present under the MASKLIST column in addition to CML_PROP_HEADER_RECORD under the TYPECOLUMN column, yet there are no fields declared starting under the STARTCOLUMN. This is perfectly valid since the default values from the recorded artefacts will be used, as discussed in section **??**.

For illustrative purposes we will however focus on when there are fields declared in the declaration type row. If we look at row 15 in Figure 25, it can be seen that there are six

fields declared in the declaration type row, starting at the STARTCOLUMN ie. column E to column J. Consequently there are no restrictions on the amount of columns used, other than the limits imposed by the spreadsheet itself.

A question which may arise at this point is, how are the field names that are declared in the columns E through J determined and mapped? The answer is of course, the Alias Library, which has been discussed in section . This, together with the helper artefacts discussed in section , create a structure, that will in map the fields to actual variables that exist in the type declared in the TYPECOLUMN. This structure houses the data from a given executable unit or EXECUTABLE type row (discussed in the next section) to be mapped correctly and in turn be passed to objtypes to create a buffer to be delivered through a message or written to file.

18.6 Executable Units

An executable unit, as can be seen in row 14 contains no data, this is line with the case in the previous section where a COLUMNLIST or declaration type row has no fields declared, the executable unit then triggers the previous declaration to create a structure with all defaults to be passed to objtypes to create a buffer to be delivered through a message or written to file. This case is however the most trivial, it is in most cases more common to populate the fields with data more suited to the task at hand.

In row 16 we have such a case whereby in the columns E through J there is customized data that maps to the fields set in the corresponding declaration type row. This data will in turn be passed to the variables created in the structure defined by the previous declaration row, which will then be used with the default values and get passed to objtypes to create a buffer to be delivered through a message or written to file.

18.6.1 An Ignored Row

An ignored row will always start with INCOMPLETE in the MASKLIST column as can be seen in row 18 in Figure 25. This row will not be processed and may contain any data, fields or comments the end user desires, and when the need arises could be converted to an executable unit or declaration type row as required.

19 Helper Artefacts

Helper Artefacts are typically thistle scripts, thistle library scripts or Type A interfaces that are aimed at minimizing the effort required in performing tasks that are either very common, for example, retrieving a currency code, or tasks that are essentially house keeping tasks, for example, spreadsheet navigation,

Helper Artefacts have no specific format, since this would limit flexibility in terms of what can be achieved, they must however address the problem space in an efficient and effective manner and should be general enough to be used across multiple solutions.

19.1 Code Magus Limited Helper Artefacts

Code Magus Limited provides amongst others, the following helper artefacts:

- GetNextColumn
- GetHeaders

The Design pattern for flexible script solutions requires any input from the input spreadsheet to be dynamic. The reason for this is that for any given run the fields contained in a column can and usually will change, in fact they will usually change several times within a single run.

In order to deal with this requirement, the two helper scripts mentioned above have been provided to aid developers in their implementations.

19.1.1 GetNextColumn

The GetNextColumn artefact returns successive column identifiers each time it is called, starting at some user defined initialisation point. The initialisation is done by providing GetNextColumn with an integer corresponding to column identifier, where column A is equal to zero, column B equal to one and so on.

Once GetNextColumn has been initialised, successive calls to GetNextColumn will return successive column identifiers.

```
01  package GETADDRESS;
02  { preamble }
03
04    created by 'Developer Name';
05    description 'Manipulate data to create an Address File';
06    date 2008-06-30T09:31:28;
07    target 'Eresia File Portal';
08    library AddrLib    : CMLSFilesScripts.AddressLib;
09    usecase MerchData  : CMLSFilesScripts.MERCHANT;
10    usecase CustData   : CMLSFilesScripts.CUSTOMER;
11
12 begin
13   MerchStruct := MerchData();
14   CustStruct  := CustData();
15
16   MerchMap    := AddrLib.MerchantAddressMap();
17   CustMap     := AddrLib.CustomerAddressMap();
18
```

```
19   PostToMerchantStr := MerchStruct.[MerchMap['MERCHANT']['STREET_ADDRESS']];
20   PostToCustStr     := CustStruct.[CustMap['CUSTOMER']['STREET_ADDRESS']];
21 end.
```

The example above shows GetNextColumn being initialised to zero or to the beginning of column identifier list. The output from the three successive calls to GetNextColumn will produce the output:

```
A B C
```

If GetNextColumn had been initialised with one, the output would have been:

```
B C D
```

Section 21 will provide details on how GetNextColumn will be used for dynamic spreadsheet traversal.

19.1.2 GetHeaders

Column Traversal is a useful mechanism but it does not provide details on which columns will need to be navigated. A controlling mechanism needs to know which columns in a spreadsheet have significance. The GetHeaders artefact allows developers to create anchors within a spreadsheet which determine the boundaries of what is required.

Recall from Section 18.3 that the design pattern makes use of three standard headings, namely:

- `MASKLIST`
- `TYPECOLUMN`
- `STARTCOLUMN`

Each of these headings defines their own boundary, which in turn defines how the data in the input spreadsheet should be formatted. What is required is a mechanism to determine where in the spreadsheet this boundaries exist, which is exactly the purpose of the GetHeaders artefact.

Consider the following thistle artefact to retrieve heading and boundary information from a given spreadsheet.

```
01 package SeekHeadings;
02
03     created by  'Developer Name';
04     description 'Retrieve header information from a spreadsheet.';
05     date         2005-04-13T13:13:13;
06     target      'Retrieve headers';
07     interface Portal.Excel : CodeMagus.Excel;
08     usecase GetHeaders : CommonScripts.GetHeaders;
09
10  begin
11    {Connect to Excel}
12    [thisInstance].SDATA := Portal.Excel.Connect(System.Root_Directory
13                             # System.Defined_Names.AMEXFilesSpreadsheets #
14                             "Generic_Scenarios.xls");
15
16    {Get Header Information}
17     HdrInf :=
18    GetHeaders(SDATA.WorkSheet.File_Sheet,"MASKLIST","TYPECOLUMN","STARTCOLUMN");
19
20     System.DumpScope(HdrInf);
21
22   end.
```

Line 17 and 18 show the usage of the GetHeaders artefact, the first parameter supplied to the artefact is the worksheet which is being dealt with, the following three parameters are the standard headings, in which the order is of importance since the GetHeaders artefact will search for the headings in that particular order.

The constraints on the headings as mentioned in Section 18.3 is that the `MASKLIST` must reside in column A and that the subsequent headings must appear in the same row as the `MASKLIST`.

The GetHeaders artefact returns a structure which contains all the information necessary to construct a controlling mechanism capable of dynamically traversing the spreadsheet in order to implement the design pattern for flexible script solutions.

The output of the dumpscope on line 19 is as seen below.

```
Target1Row <property: string len 2> = '12'
Target2Col <property: string len 1> = 'B'
Target2ColAsNumeric <property: string len 1> = '2'
Target3Col <property: string len 1> = 'E'
Target3ColAsNumeric <property: string len 1> = '5'
Target4Col <property: string len 2> = 'IV'
Target4ColAsNumeric <property: string len 3> = '255'
Target5Col <property: string len 2> = 'IV'
Target5ColAsNumeric <property: string len 3> = '255'
```

The dumpscope reveals the structure that has been returned after the call to GetHeaders.

the Targets ie. `TargetXRow,TargetXCol` corresponds to the headings supplied to the GetHeaders artefact. For instance, `TYPECOLUMN` corresponds to the values returned by `Target2Col` and `Target2ColAsNumeric`. In other words, the `TYPECOLUMN` appears or is bounded by column B, the value returned by `Target2ColAsNumeric` is a numeric representation of the column, where `1` represents column A and so on. This is useful since the GetNextColumn must be initialised with an integer.

It can be seen from the output of the dumpscope that no elements exist for `Target1Col` and `Target1ColAsNumeric` but only an element for `Target1Row`. The reason for this is that it is by design that the `MASKLIST` will always appear in column A, what was not known is which row it would appear in, which has thus been returned.

Another point to note is that the structure returns entries for a fourth and fifth target, this is to provide flexibility in order for additional headings to be defined. This functionality however is not required for the design pattern for flexible script solutions.

Section 21 will provide details on how GetHeaders and GetNextColumn will be used for dynamic spreadsheet traversal.

19.2 User Defined Helper Artefacts

Users can as necessary create their own helper artefacts in order to solve frequently occurring problems. This approach will save on time and resources in the future when these problems are encountered in the future.

20 The Applparms Interface

20.1 Introduction

In order to provide a common user interface to any of the multitude of possible implementations of the Code Magus design pattern, the applparms library was developed, creating the platform for the development of several GUI styles/templates that developers may use in providing graphical user interfaces to their implementations.

It was found that even though the input spreadsheet provided a useful and powerful interface for users, the fact that a multitude of input spreadsheets may exist for a given pattern implementation was reason enough to provide an interface layer for users beyond that so that absolutely no modification of the scripts are ever required by end users.

Developers now have the ability to define an arbitrary amount of parameters as well as the required constraints on those parameters in the applparms configuration file. The GUI will then open the respective applparms configuration file which can then be populated by the end user running the respective package.

Once the user has provided values for all the parameters, the values will be returned to the thistle script via the applparms type A interface and will be used in the running script to whatever end intended by the developer.

There are currently two GUI's available for use with Eresia, namely a tabbed version as well as a list version. All the GUI's as well as the command line user interface are described in detail in the applparms reference. Figure 26 and Figure 27 illustrate the tabbed GUI and the list GUI respectively.

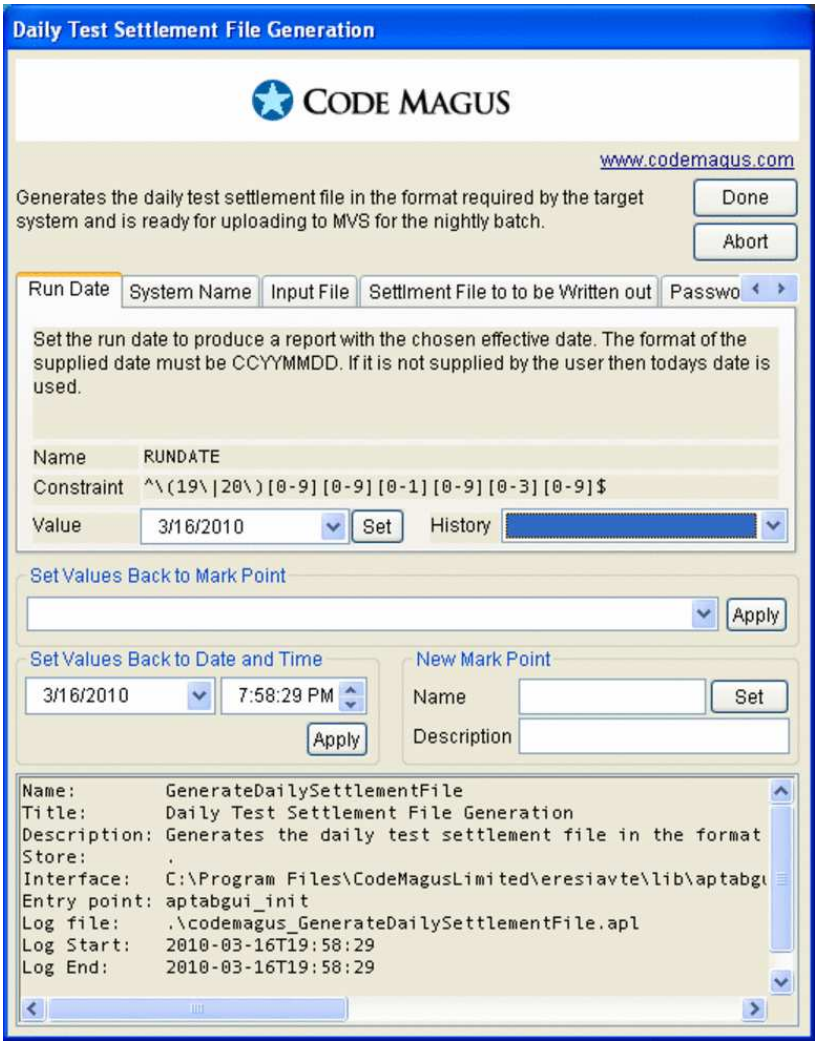


Figure 26: The applparms tabbed GUI

For the purposes of the discussion we will use the tabbed GUI as a running example, however, all aspects relating to the applparms configuration file as well as the applparms type a interface in Eresia apply equally to both. It must also be noted that the functionality described in this document will be kept brief and should more information be required, the Application Parameters Library User Guide and Reference[7] should be consulted.

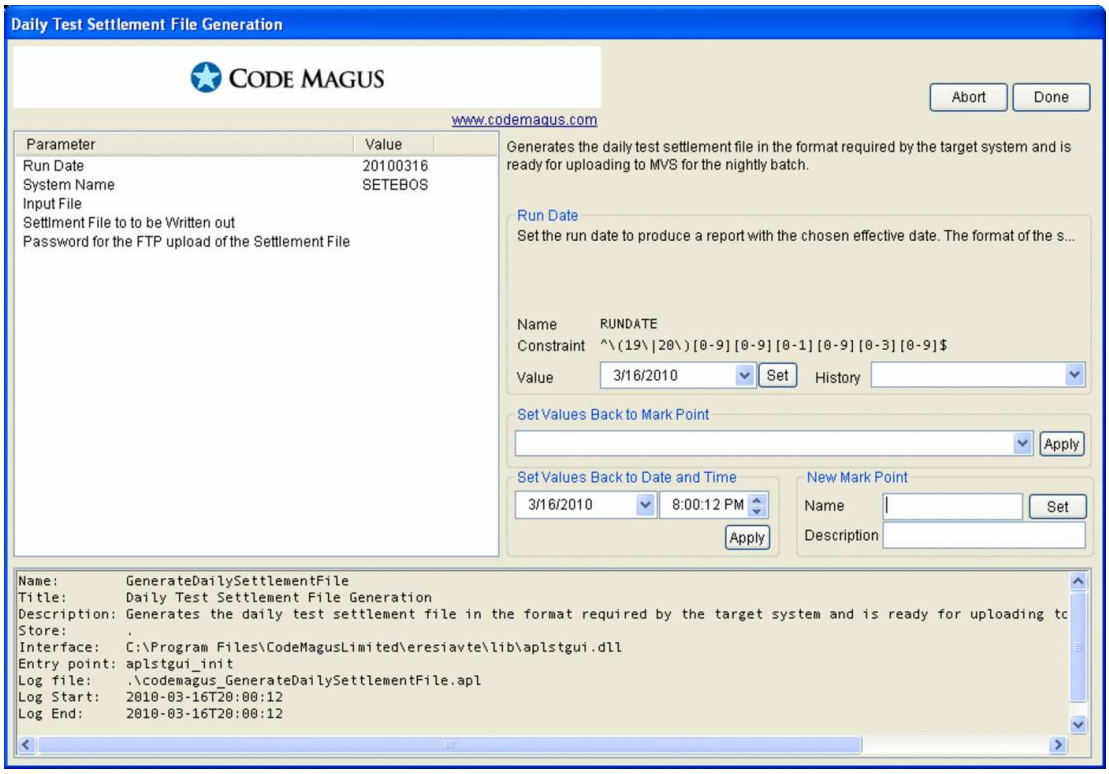


Figure 27: The applparms list GUI

20.2 The Applparms Configuration File

The applparms configuration file will be used by the GUI to provide the parameter members to be used by the pattern. In most cases related to the pattern these parameters represent items at a global level, for example, the name or IP address of a host that is to be transacted against or the name of the input spreadsheet required for the given run. However, developers are free to define as many parameters as necessary that help achieve their specific goals.

In the example below, two parameters have been defined, namely `InputSpreadsheet` and `SettlementFileName`. Users will supply values to these two parameter which will then be used by the scripts for subsequent processing. In Section 20.3 the detail around opening and accessing an applparms configuration file in thistle is provided.

```
application CMLSettlement;
-- CMLSettlement module APD file used by the CMLDemoSettlement suite of
-- settlement file generation scripts.
--
-- $Author: justin $
-- $Date: 2010/03/10 13:23:17 $
-- $Id: CMLSettlement.apd,v 1.2 2010/03/10 13:23:17 justin Exp $
-- $Name:  $
-- $Revision: 1.2 $
```

```
-- $State: Exp $
--
--
--

   title "Code Magus Limited Settlement File Creation Module.";
   description "The Code Magus Limited settlement script suite, "
               "facilitating the creation of settlement files based "
               " on the American Express Programmer Specifications."
     ;

-- Set the logs directory to 'C:\Documents and Settings\{username}'
-- using USERPROFILE which is a standard Windows XP environment
-- variable.

   set LOGHOME = ${USERPROFILE};
   set TODAY = ${DATE_YYYYMMDD};

-- Set an path evironment variable for parameters that contain a path
-- so as to point to the default open path for that parameter.

   set CODEMAGUS_APPLPARMS_SettlementFileName_PATH=
           "C:\CodeMagus\CodeMagusDemo\CMLDemoSettlement\testdata\data\";

   set CODEMAGUS_APPLPARMS_InputSpreadsheet_PATH=
           "C:\CodeMagus\CodeMagusDemo\CMLDemoSettlement\testdata\spreadsheets\";

   store ${LOGHOME};

-- The interface defines the shared object or DLL program that will interact
-- with the user to ensure that all parameters have a value. The value
-- for interface is string in quotes naming the DLL program or the word default
-- in which case the UI is the command line UI.

   interface "C:\Program Files\CodeMagusLimited\eresiavte\lib\aptabgui.dll";
   entry aptabgui_init;

   parameter InputSpreadsheet
      title "Input Spreadsheet";
      default NULL; -- forces the user to insert a value if one is not found
      options filename;
                  -- allows system to file system navigate to choose name
      description
         "This is the name of the Excel spreadsheet that will be "
         "responsible for supplying input data to the current run "
         "of the script.  The spreadsheet consists of executable "
         "test definitions and scenarios that are the driving force "
         "in the creation of the settlement file. ";
      constraint "^[^ ]\+$"; --- at least one character, no spaces.
   end

   parameter SettlementFileName
```

```
      title "Settlement File Name";
      default NULL; -- forces the user to insert a value if one is not found
      options filename;
                  -- allows system to file system navigate to choose name
      description
         "This is the name of the settlement test file that will be "
         "created as a result of a success execution of this script. "
         "The file will be in a format suitable for directly copying "
         "into the concantenation of your settlement process. ";
      constraint "^[^ ]\+$"; --- at least one character, no spaces.
   end

end.
```

20.3 Using the Applparms interface in thistle

The script extract below illustrates the interaction between thistle, the applparms interface and the applparms configuration file. The applparms interface is defined in the preamble section on line 12, making it available for use within the script.

On line 16, we connect to the applparms interface, this creates an instance that will be used to open and access the applparms configuration file via the mechanisms provided by the interface.

```
00  usecase CMLSCreateFile(
01            ACCESS_METHOD := "binary",
02            AM_OPTIONS    := "recfm=f,mode=wb,reclen=190");
03
04
05  { preamble }
06
07     created by 'Justin Albertyn';
08     description 'Create a Code Magus Limited Settlement File';
09     date 2005-08-04T10:51:18;
10     target 'Eresia File Portal';
11     interface Portal.Excel       : CodeMagus.Excel;
12     interface ApplParms          : CodeMagus.ApplParms;
13
14  begin
15    {Set up optional initial parameters}
16     ParameterData := ApplParms.Connect();
17
18     ParameterData.open("C:\YourPath\CMLSettlement.apd","/VERBOSE");
19
20
21     FILENAME    := ParameterData.SettlementFileName;
22     DATABOOK    := ParameterData.InputSpreadsheet;
23
24     {Connect to Excel}
```

```
25      [thisInstance].TESTDATA := Portal.Excel.Connect(DATABOOK);
26
27      open_spec_str  := ACCESS_METHOD # "(" # FILENAME # "," # AM_OPTIONS # ")";
28
29  ...
30  end
```

Following that, the instance opens the applparms configuration file, producing the GUI as shown in Figure 28.

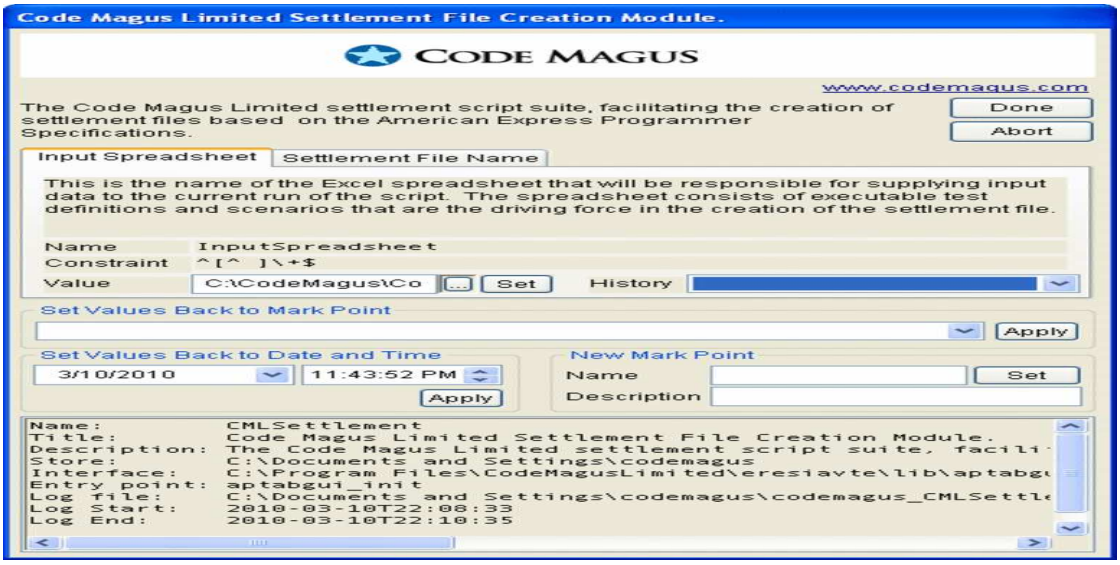


Figure 28: The applparms tab GUI opened from thistle

Using the GUI, the user will need to provide the values required by the parameters which must match the constraints set by the developer. Once the user has provided the detail and pressed the `DONE` button, control will be returned to thistle and the values provided by the user will populate the variables on lines 21 and 22.

This mechanism of providing input via the applparms GUI's is the standard required by the pattern as it simplifies interaction for end users as well as significantly limits input related errors previously encountered.

21 Control Artefacts

Thus far, a number of pattern components have been discussed in relative isolation from one another. The components themselves may be useful as standalone entities or may have limited value, but once placed together they form the core of the design pattern for flexible script solutions. It is the job of the control artefacts to bind all the components together.

The two artefacts that provide the required control functionality and bindings are the Control Usecase and the Control Package.

21.1 The Control Package

A Package is a Thistle artefact which is runnable by the Eresia Visual Test Environment, and the control package is thus the runnable unit in the solution. The package artefact itself should contain only enough detail in order to ensure that users are not required to edit the control usecase.

The sample package below specifies only detail that will be required from the control usecase. Developers will be able to use this package almost as is for any implementation of the design pattern for flexible script solutions. There are however values that will have to change depending on what is being implemented.

```
01 package YourControlPackName;
02 { preamble }
03
04    created by 'Developer Name';
05    description 'Control Package';
06    date 2005-08-02T09:31:28;
07    target 'Package intended target';
08    usecase YourControlUsecase  : FileScripts.YourControlUsecase;
09
10 begin
11  {Set up initial parameters}
12  typespath     := System.Root_Directory # System.Defined_Names.CMLTYPES
13                         # "YourTypesDefFile.objtypes";{supply req. objtypes file}
14
15  FILENAME      := System.Root_Directory # System.Defined_Names.CMLDATA;
16  ACCESS_METHOD := "binary";                    {supply req. access method}
17  AM_OPTIONS    := "recfm=f,mode=wb,reclen=190";{supply req. access method opts}
18  APDFILE       := 'YourApplparmConfig.apd';    {applparms configuration file}
19
20
21  {Call the control usecase with initial parameters.}
22
23  YourControlUsecase(typespath    :=typespath,
24                     FILENAME     :=FILENAME,
25                     ACCESS_METHOD:=ACCESS_METHOD,
26                     AM_OPTIONS   :=AM_OPTIONS,
27                     APDFILE      :=APDFILE);
28 end.
```

It is useful to supply a meaningful name to the control package, and to the control usecase, accordingly the package name on line 1 and usecase name on line 8 would need to be changed at the users discretion.

It is good practice to update the usecase preamble on lines 4 to 7, since it provides useful

information to someone using the artefact the first time.

The initial parameters on lines 12 through 18 require any appropriate paths to be set up in the thistle configuration file. Thus any path changes will be limited to this file and users can format their path strings through the `System` structure.

There may be situations where developers will need to add fields or constructs to the control package but for the general case the above example is well suited and should not need any modification whatsoever.

21.2 The Control Usecase

The Control usecase is the artefact that binds all the pattern components together into one logical unit. The layout presented below will provide developers with the necessary structure to successfully implement the design pattern for flexible script solutions, and will be seen that the layout will not change from implementation to implementation, but will can be built upon to create the intended solutions.

In order to make clarify the use and structure of the control usecase, it will be broken into sections which will in turn each be investigated individually. The sections will be defined as follows:

1. The Preamble Section - Lines 1 through 21.
2. The Initialisation Section - Lines 24 through 55.
3. The Main Control Loop
   (a) The Columnlist Collector - Lines 60 through 80.
   (b) The Execution Manager - Lines 82 through 113.
   (c) The Re-initialisation Section - Lines 114 through 116.

```
001 usecase ControlUsecaseName(
002          typespath     := "C:\\SomePath\\Default.objtypes",
003          FILENAME      := "C:\\SomePath\\Default.bin",
004          ACCESS_METHOD := "binary",
005          AM_OPTIONS    := "recfm=f,mode=wb,reclen=190",
006          APDFILE      := nil);
007
008{ preamble }
009
010    created by 'Developer Name';
011    description 'Control Usecase';
012    date 2005-08-04T10:51:18;
013    target 'Eresia File Portal';
014    interface Portal.recio       : CodeMagus.RECIO;
015    interface Portal.Excel       : CodeMagus.Excel;
016    interface Portal.Types       : CodeMagus.Types;
017    interface ApplParms          : CodeMagus.ApplParms;
018    Library AliasLibrary         : LibraryScripts.AliasLibrary;
019    usecase GetColumn            : CommonScripts.GetNextColumn;
020    usecase GetHeaders           : CommonScripts.GetHeaders;
021    usecase RecordedScript       : FilesScripts.YOUR_RECORDED_SCRIPT;
022
023 begin
024    stream        := Portal.recio.Connect();
025    types         := Portal.Types.Connect(typespath);
026    ParameterData := ApplParms.Connect();
027    ParameterData.open("C:\YourPath\CMLSettlement.apd","/VERBOSE");
028
029    FILENAME      := ParameterData.SettlementFileName;
030    DATABOOK      := ParameterData.InputSpreadsheet;
031
032    [thisInstance].TD := Portal.Excel.Connect(DATABOOK);
033
034    {Get All General Data From the Spreadsheet}
035    IndexFrom   := TD.WorkSheet.File_Sheet.B[7];
036    IndexTo     := TD.WorkSheet.File_Sheet.B[8];
037
038    open_spec_str  := ACCESS_METHOD # "(" # FILENAME # "," # AM_OPTIONS # ")";
039
040    {Get Header Information}
041    HdrInf
042   := GetHeaders(TD.WorkSheet.File_Sheet,"MASKLIST","TYPECOLUMN","STARTCOLUMN");
043
044    {Open the output stream to write the file}
045    outstrm :=
046    stream.open(open_string:=open_spec_str,mode:="SEQ_OUTPUT",flags:="/VERBOSE");
047    {Types Available for recording}
048    SampleRecord      := RecordedScript();
049
050    {Load Aliases From Library Artifact}
051    SRFldMapping   := AliasLibrary.RecordedScriptMap();
052
```

```
053    {GLOBALS}
054    msg_no := 0;
055
056    {Get necessary data from the spreadsheet as many times as necessary}
057    for Index := IndexFrom to IndexTo do
058    begin
059
060        status := TD.WorkSheet.File_Sheet.A[Index];
061
062        {If the row is definition collect the data and     }
063        {allow it to persist until a new definition occurs }
064        if status = "COLUMNLIST" then
065        begin
066          CurrDef := "";{initialize Structure;}
067          CurrDef.TypeName
068             := TD.WorkSheet.File_Sheet.[HdrInf.Target2Col].[Index];
069          CurrDef.Column   := "";
070
071          GetColumn(HdrInf.Target3ColAsNumeric-1);
072          iter := GetColumn();
073
074          while (TD.WorkSheet.File_Sheet.[iter].[Index] <> "") do
075          begin
076            CurrDef.Column.[iter]
077               := TD.WorkSheet.File_Sheet.[iter].[Index];
078            iter := GetColumn();
079          end
080        end
081
082        if status = "EXECUTABLE" then
083       begin
084
085           {Code Magus Limited: Sample Record Layout}
086        if CurrDef.TypeName = "SAMPLE_RECORD_TYPE_NAME" then
087           begin
088
089            {Fetch Data From Sheet As Defined by }
090             {the current definition              }
091             if CurrDef.Column <> "" then
092             begin
093               for col in CurrDef.Column do
094               begin
095                  {Add the values populated in the spreadsheet to the }
096                  {record structure.}
097       SampleRecord.[SRFldMapping[CurrDef.TypeName][CurrDef.Column.[col]]]
098                  := TD.WorkSheet.File_Sheet.[col][Index];
099               end
100            end
101
102            {Automatically calculated variables.}
103            msg_no := msg_no + 1;
104
```

```
105             {Stuff Auto-calculated variables.}
106             SampleRecord.[SRFldMapping[CurrDef.TypeName]["MESSAGE_NUMBER"]]
107                                                                     := msg_no;
108             {Write out the buffer to file}
109             Data.Buffer   := types.GetBuffer(SampleRecord,CurrDef.TypeName);
110             stream.write(outstrm,Data.Buffer);
111
112           end {Code Magus Limited: Sample Record Layout}
113       end{End if "EXECUTABLE"}
114
115       {Reload Default Values}
116       SampleRecord      := RecordedScript();
117    end{For}
118 end.
```

21.2.1 The Preamble Section

The Preamble Section begins the mandatory `usecase` keyword, followed by the name assigned to the control usecase, which as mentioned previously is useful to represent something meaningful.

The control usecase name is then followed by parenthesis containing the parameter list that will be populated when being called from the control package as was demonstrated in Section 21.1. The values assigned to the parameter in the control usecase provide default values should values not be given when called from the control package.

In the example lines 10 through 13 represent the development and information details that are to be populated as the developers sees fit. The interfaces are defined over the next four lines and the listed interfaces, namely `recio`[1] , `types`,`excel` and `applparms`[7] are in most cases sufficient for the design pattern implementation.

Line 18 shows the `AliasLibrary` as discussed in Section 18.5 being initialised for use in the control usecase.

The three usecases initialised on line 19, 20 and 21 represent the `GetNextColumn` and `GetHeaders` artefacts discussed in Section19 and the `RecordedScript` on line 21 is the recorded artefact with overrides applied for a particular message or record. This artefact will contain all the required default values for the message or record. There will usually be several different default value artefacts in a given implementation of the design pattern.

21.2.2 The Initialisation Section

The Initialisation Section in the control usecase is point in the artefact at which all global variables will be defined, connections to all portals will be established and all structures will be populated.

Lines 24 to 32 are examples of the interface variables being connected to the various portals, as well as the opening and interfacing to the applparms GUI and configuration file. Lines 29 and 30 are more specifically variables being populated through the use of the applparms interface.

Lines 35 and 36 show the data being extracted from the control section of the spreadsheet. The `FILENAME` variable populated will be used to form the open specification string for the relevant access method which is shown on line 38.

The `FILENAME` forms the object part of the open specification string and parameter supplied from the control package will make up the rest of the string by supplying the access method name and options.

This method of creating the open specification allows developers to change the access method in the control package without having to worry about further implementation specific details. The `IndexFrom` and `IndexTo` variables will be used in the main control loop to control program flow.

Lines 41 and 42 show the creation of the structure representing all the relevant heading information in the input spreadsheet. This structure was discussed in Section 19 and will provide the boundaries within the input spreadsheet.

Lines 45 and 46 creates the output stream to which data will be written and will be used in the execution manager for messages or records destined for a file or the network.

An instance of a default value artefact being assigned to a structure to used within the execution manager is shown on line 48. The data from the input spreadsheet will be applied as necessary to this structure which creates the situation where only relevant information is required in the spreadsheet and all other values will be defaulted to reasonable values. In a given implementation there would normally be several structures associated with default value artefacts.

Line 51 shows the manner in which the mappings are retrieved from the alias library whereby an exported method containing the alias' for a particular record or message are assigned. On line 54 there is one example of a global variable being defined. In the example it is a message counter that will be inserted into the message or record at some point. Developers should use their own discretion when it comes to global variables, which are useful constructs for keeping track of running totals or state information.

21.2.3 The Main Control Loop

The Main Control Loop manages the flow and collection of data from the spreadsheet into the defined structures as well as the output to the designated streams. The Main Control Loop comprises a `for` loop since the number of iterations has been predetermined by the values in the control section of the input spreadsheet.

The remainder of the code in the example will execute within the main control loop. The design pattern for flexible script solutions defines three standard headings under which certain types of data will be permitted. The values populated under the `MASKLIST` standard heading determine which action will be selected within main control loop. The values permitted under the `MASKLIST` standard heading column are `COLUMNLIST`,`EXECUTABLE` or `INCOMPLETE`.

Therefore we create a variable `status` as seen on line 60 which will be populated from the currently executing row within the input spreadsheet whose value has been collected from the `MASKLIST` column. If this value is equal to `COLUMNLIST` or `EXECUTABLE` then either the columnlist collector or execution manager will be started. No conditional statements will be executed if the `INCOMPLETE` keyword is encountered and hence the row is completely ignored.

Consider the spreadsheet in Figure 29 extract along with the control usecase example in order to investigate the remainder of the main control loop functionality.

	A	B	C	D	E	F	
1	TITLE OF PACKAGE	SAMPLE PACKAGE					
2	NAME OF PACKAGE	ControlPackage.pts					
3							
4							
5							
6							
7	Test Cases (from row)	13					
8	Test Cases (to row)	16					
9							
10							
11							
12	MASKLIST	TYPECOLUMN			STARTCOLUMN		
13	COLUMNLIST	SAMPLE_RECORD_TYPE_NAME			CUST_NAME	ACC_BAL	
14	EXECUTABLE				J.Smith		20000
15	COLUMNLIST	SAMPLE_RECORD_TYPE_NAME			CUST_NAME	ACC_BAL	
16	INCOMPLETE				H.Jones		23000
17							
18							

Figure 29: Input Spreadsheet Example

21.2.4 The Columnlist Collector

The Columnlist Collector can be seen on lines 64 to 80 of the control usecase example and will only process a row in the input spreadsheet when the `COLUMNLIST` keyword is found in the `MASKLIST` column. The `if` statement on line 64 ensures that this condition is met.

Line 66 shows the initialisation of the variable `CurrDef`, which is to become a structure that will contain the `COLUMNLIST` definitions. This structure will persist until the next `COLUMNLIST` keyword appears in the input spreadsheet in which case it will be reinitialised and populated.

The code on line 67 and 68 retrieves the currently defined type name from the `TYPECOLUMN` in the input spreadsheet and assigns it `CurrDef` structure, this is based on the current row being equivalent to the loop control variable `Index` and the appropriate target column as returned to the structure when GetHeaders was called on line 39 and 40. In Figure 29, assuming the variable `Index` was equal to 13 would assign the value `SAMPLE_RECORD_TYPE_NAME` to `CurrDef.TypeName`.

The lines 71 and 72 show the first usage of the GetNextColumn artefact as discussed in 19. The first call initialises GetNextColumn to one column before the `STARTCOLUMN`, that way when GetNextColumn is called for the first time the artefact will return the column identifier of the `STARTCOLUMN` which based on the layout of Figure 29 would return `E`. The value `E` is assigned to the variable `iter` and the starting point of all field definitions for the `COLUMNLIST` is available through the use of `iter`.

The `while` loop defined on lines 74 to 78 gathers all field definitions defined in the `COLUMNLIST` until an empty cell is encountered, which is the loop termination condition. For each iteration of the loop the `CurrDef` structure is updated with any field definitions that are encountered. At the end of each iteration the variable `iter` is moved along to the next column identifier and that column will then either update the structure once more or terminate the loop.

Once the loop termination condition is met, the `CurrDef` structure will populated as shown below and now contains sufficient information to deal with the data encountered in the execution manager.

```
TypeName <property: string len 23> = 'SAMPLE_RECORD_TYPE_NAME'
Column
```

```
E <property: string len 9> = 'CUST_NAME'
F <property: string len 7> = 'ACC_BAL'
```

21.2.5 The Execution Manager

The Execution Manager begins with an evaluation of the `status` variable that was assigned at the beginning of the main control loop to determine whether or not the code within the `if` statement is to be executed or skipped on the current iteration. The `if` statement ensures that the code will only be executed if the current row is in fact marked with the `EXECUTABLE` keyword.

The execution manager now has to determine which type of record or message was defined when the columnlist collector gathered the definitions for the preceding `COLUMNLIST` row. This is achieved by comparing the current `TypeName` in the `CurrDef` structure to some known type to be processed. The typical case is that there will be several comparisons to known types in a given execution manager each with a body of code particular to the processing of that type. The above example however only contains one comparison to a known type `SAMPLE_RECORD_TYPE_NAME`, and the shown code is the bare minimum required for the design pattern for flexible script solutions. Developers must add additional code as required for the needs of particular solution.

When the execution manager determines that type in the `CurrDef` structure matches the known type the processing and gathering of data will proceed. The `if` statement on line 91 determines whether or not there is in fact column data in the `CurrDef` structure. In a situation where no field definitions where defined, the column data in the `CurrDef` structure would not be present. The execution manager would then proceed past the if statement and execute the subsequent statements, which is in fact required in some situations.

It is however required to examine the operation of the code in a situation where field definitions were supplied in the `COLUMNLIST` row. The `for` statement on line 93 make use of the `in` construct to iterate through the child nodes of the `Column` element in the `CurrDef` structure. The `in` construct assigns the values of the child nodes to the variable `col`, for the example shown in Figure 29 the variable `col` will take on the values `E` and `F` respectively.

Before examining the assignment statement on line 97 and 98, it is worthwhile to point out that in the event of a variable being enclosed in square brackets, The Thistle runtime will interpret the variable as an expression or an evaluated node.

Consider the following statements:

```
Col   := "E";
[Col] := "ABC";
```

`Col` is assigned the value `"E"` whereas `[Col] := "ABC";` evaluates to the expression `E := "ABC";` which in turn assigns `"ABC"` to the variable `E`.

This construct will be repeatedly used in the design pattern for flexible script solutions since the data from the input spreadsheet cannot be known while the solution is being developed. Consider the code on lines 97 and 98 in the execution manager.

```
SampleRecord.[SRFldMapping[CurrDef.TypeName][CurrDef.Column.[col]]]
   := TESTDATA.WorkSheet.File_Sheet.[col][Index];
```

Assuming the execution manager was currently executing row 14 column E, the Left hand side of the assignment statement would evaluate to the following expression:

```
TESTDATA.WorkSheet.File_Sheet.E.14;
```

The data contained in column E, row 14 will then be the value that is assigned, ie `J.Smith`. The code enclosed in square brackets on the right hand side of the assignment statement would evaluate in the following manner.

```
SampleRecord.[SRFldMapping[CurrDef.TypeName][CurrDef.Column.[col]]]
```

evaluates to

```
SampleRecord.[SRFldMapping.SAMPLE_RECORD_TYPE_NAME.CUST_NAME]
```

The alias library will then be called to return the fully qualified variable name. This variable name is subsequently used to replace the particular value in the structure assigned from the default value artefact. The fully evaluated expression is shown below.

```
SampleRecord.CP01LVL.CUSTINFO.CUST_NAME := TESTDATA.WorkSheet.File_Sheet.E.14;
```

This process is repeated until there are no more columns to process in the `CurrDef` structure and all values from the input spreadsheet have been used to replace corresponding values in the structure assigned from the default value artefact.

Once all data from the input spreadsheet has been successfully placed in the structure, any global variables that are intended for the structure are updated as can be seen in the example on line 103. The global variables are then added to the structure as shown on line 106.

The remaining code in the execution manager creates a buffer based on the structure, and is written to a file or the network.

To summarize, the execution manager's aim is to successfully apply all the data from the default value artefact, the input spreadsheet and any defined global variables in the order in which they are expected to create a record or message destined for a file or network.

21.2.6 The Re-initialisation Section

It is the responsibility of the re-initialization section to ensure that all structures initialised from default value artefacts as well as any other variables or structures the require re-initialisation are appropriately initialised. The reason for this is that the execution manager will in most cases modify the contents of variables within structures initialised from default value artefacts. These variables must be restored to reflect the relevant structures before any modification by the execution manager. The re-initialisation section can be seen on line 116 of the control usecase example.

References

[1] **recio**: Record Stream I/O Library Version 1.
CML Document CML00001-01, Code Magus Limited, July 2008.
http://www.codemagus.com/documents/recio.pdf.

[2] **binary**: Fixed and Variable Length Record Stream Access Method Version 1.
CML Document CML00005-01, Code Magus Limited, July 2008.
http://www.codemagus.com/documents/binaryam.pdf.

[3] **objtypes**: Configuring for Object Recognition, Generation and Manipulation.
CML Document CML00018-01, Code Magus Limited, July 2008.
http://www.codemagus.com/documents/objtpuref.pdf.

[4] Eresia File Injection Portal (FIP) Version 2.
CML Document CML00037-02, Code Magus Limited, January 2009.
http://www.codemagus.com/documents/fipusrgd.pdf .

[5] Eresia Network Injection Portal (NIP) Version 2.
CML Document CML00038-02, Code Magus Limited, January 2009.
http://www.codemagus.com/documents/nipusrgd.pdf .

[6] Code Magus Eresia User Guide.
CML Document CML00040-01, Code Magus Limited, January 2009.
http://www.codemagus.com/documents/eresiaug.pdf .

[7] **applparms**: Application Parameters Library User Guide and Reference Version 1.
CML Document CML00054-01, Code Magus Limited, January 2010.
http://www.codemagus.com/documents/applparms.pdf .

[8] Code Magus Limited. Function and Operation of the Eresia File Injection Portal. (CML09000-01), March 2009.
http://www.codemagus.com/documents/eresia_fip_training.pdf .

[9] Code Magus Limited. Function and Operation of the Eresia Network Injection Portal. (CML09002-01), March 2009.
http://www.codemagus.com/documents/eresia_nip_training.pdf .

[10] Code Magus Limited. Function and Operation of the Eresia XML Injection Portal. (CML09003-01), March 2009.
http://www.codemagus.com/documents/eresia_xip_training.pdf .

[11] Code Magus Limited. Proprietary Financial Record Layout for Training and Demonstration Purposes. (CML09001-01), March 2009.
http://www.codemagus.com/documents/proprietary_training_specification.pdf .