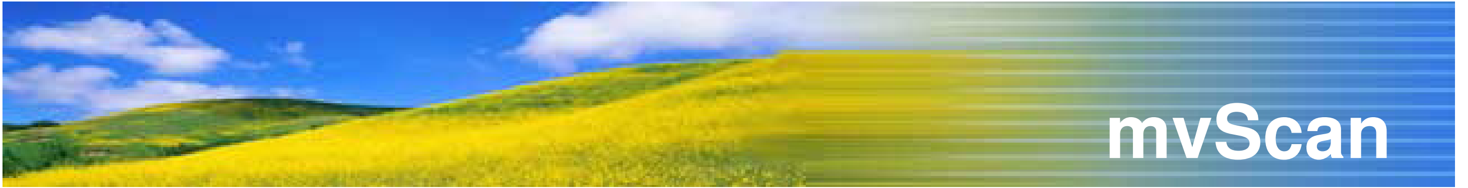


White Paper

Brian Leach Consulting Limited

http://www.brianleach.co.uk

Brian Leach Consulting Limited makes no warranty of any kind with regard to the material contained in this manual, including but not limited to the implied warranties or merchantability and fitness for a particular purpose.

The information contained in this document is subject to change without notice.

This document contains proprietary information that is protected by copyright. All rights are reserved. It may not be photocopied, reproduced, or translated, in whole or in part, without the prior express written consent of Brian Leach Consulting Limited.

Copyright 2009 Brian Leach Consulting Limited. All rights reserved.

Acknowledgments

UniVerse, UniData and UniObjects are registered trademarks of IBM U2.

Adobe and Acrobat are trademarks of Adobe Systems Incorporated.

All other company or product names mentioned are trademarks or registered trademarks of their respective trademark holders.

Contents

Contents...3
Making Sense of your Systems...4
  Understanding the Risks 5
  IBM UniVerse in the Dock 6
  Software Auditing Tools...7
  Adaptive Scanning 8
The mvScan Approach...9
  Customizing mvScan..10
  The Standard Map...11
  HTML Documentation...14
  Product Evaluations..16
  Further Information...16

Making Sense of your Systems

How well do you know your system?

Successful applications rarely stand still: they constantly adapt themselves to new business challenges, legislative requirements, improved practices, the advice of consultants and the whims of senior executives. Each change in personnel, each new organizational direction, mergers, splits, changes in the market: all of these will leave their marks on a mature system. A system tells the story of the organization in which it is situated, and such stories are rarely straight forward.

With so many changes, the knowledge of an application becomes a precious resource. Understanding your systems is not just a question of appreciating what they do, but what they have done and why.

Without that knowledge, training new staff and undertaking new projects becomes ever more costly: in terms both of time and of the risks involved. Things that may have made sense at one time have become engrained in the structure of an application: as the business has moved on, these may no longer be relevant but their ghosts remain – in the logic, in the data store, in the procedures that are followed. With the passage of time, there may be no hint that they are waiting there to trip up the unwary developer.

There are two dangers presented by this scenario. The first is dynamic: as subsequent changes are made, with each round of amendments so increase the associated risks of overlooking some vital but forgotten part of the system: a dusty year end routine brought out of retirement for its annual process and overlooked in the daily grind, causing failure or corruption when run because the system has moved on. Equally dangerous is the threat of inertia, with the risks of planned changes being judged too high to justify redevelopment when the system has outlived its original design.

In a perfect world, all changes will be have been managed, documented and their impacts noted Changing a file layout, a code block included in other programs, common shared areas of memory: all of these impacts would be logged and performed safely. In the real world, systems more often than not respond to immediate needs, with piecemeal development undertaken to solve individual problems and with that, the scope is lost.

Attempting to retrieve the shape of such a system is no trivial matter. But the costs and risks associated with not doing so, can be far harder to bear.

Understanding the Risks

Identifying and quantifying risk is a key component in the work of any IT professional, be they a director, project manager or analyst/programmer. The concept of risk is not a simple one, and sources of risk are wide and varied. For the system architect or developer, an understanding of risk is intimately combined with an appreciation of the impact of any changes.

What sort of risks? Imagine that in the heart of your IBM UniVerse application there sits a common block. A common block is a persistent and reserved area of memory used to store information shared across a number of routines: typically this may include user credentials, configuration options and references to your key database files. Under normal practice, a developer will have defined the structure of that common block in a separate code file, and included that definition inside every subroutine that makes use of this area of memory.

Common blocks are famously useful, but there is a significant drawback in their use: once a common block has been created it cannot be resized. Should the definition of that block alter between routines the application will terminate. Changing the calling arguments on a subroutine will have the same effect. The failure to identify even one single, rarely-used subroutine creates the risk of an application crash.

Not all risks are directly associated with new code. A new field may be added to a file that increases the record size: if a program reads this into a fixed sized array it too will fail. Other changes may lead to more subtle risks. Add a discount field to the order lines on a sales order: if any program manipulating these is not made aware of its existence, inserting or deleting entries from the existing fields in the series – product codes, order quantities, prices – will result in the discount being assigned to the wrong order line. Such changes offer embarrassing and potentially costly consequences for the organization.

IBM UniVerse in the Dock

IBM UniVerse is more than a simple database. Unlike its relational cousins, IBM UniVerse is a solid business platform with the language tools to deliver scalable and complex applications.

With this additional functionality comes greater scope for danger. IBM UniVerse applications are rarely designed using just one language: an application may be composed of UniVerse Basic code in the form of programs, subroutines and functions; commands executed through the PROCedural language or stored into batches as Paragraphs; enquiry statements written in RetrieVe or SQL; metadata held in file dictionaries and definitions powering fourth generation languages, menus and screen runners. The key to building an IBM UniVerse application is to choose the right tool for that part of the job: thus all of these interact to deliver a single solution.

Relational databases are highly structured and offer little beyond pure storage and recall. IBM UniVerse, in common with its sister MultiValue databases, offers tremendous flexibility and a wealth of runtime features. The information held in a UniVerse database is defined only when it is needed; relationships between files may be hidden behind individual fields; and much of the structure is discoverable only at runtime.

UniVerse applications are highly modular. Code is written in separate, isolated blocks and loaded at run time and on demand as the application dictates. There are huge advantages to this approach: applications can be extended without rebuilding the entire code base, new features can be added on demand and customized applications may make choices between loading alternate routines. All this means that there is no stage at which the whole application is automatically checked for consistency.

For UniVerse and other MultiValue platforms, it is always the responsibility of the developer to ensure that application integrity is maintained. And yet UniVerse offers the developer little assistance with this task – often at best they can search the known code base for a file name or subroutine call, but even this is limited to simple text searching tools that are not aware of context. A file may be opened using a synonym. A command may be built up using a paragraph and executed from a program. A colleague may have taken a working copy of a program to amend and placed it outside the normal source locations. The wrong version of a program may be resident into the global catalog directory from which it is loaded.

Finding the source of an error, or better still, understanding the context of changes proposed to the system, is reduced to little more than trial and error; time consuming and expensive.

Software Auditing Tools

Even in the best regulated sites, the open and flexible nature of the IBM UniVerse platform code presents barriers to discovery. For legacy systems, passing through many hands and subject to many coding styles and variations in conventions, this becomes a thankless and all but impossible task. With this fog of understanding, applications may become stale and stagnate because managers fear the risks involved in changing the legacy code until they no longer serve the business.

Software auditing tools such as **mvScan** can assist in clearing that confusion. These tools map out the impacts ahead of time, removing the need for analysts to spend hours researching through code listings. The more sophisticated software auditing tools can examine every part of an IBM UniVerse system - account by account, file by file, program by program, command by command - using parsing and recognition techniques to help build an impact map.

Impact maps chart the relationships and impacts hidden inside your application. These tools go beyond the simple relational charts to help to build up an understanding of how the component pats of an application fit together. A map allows developers to navigate the system in any direction, isolating risk areas and scoping changes.

By examining the map, a developer may for example quickly identify all the routines that call a particular subroutine before changing those calls. Looking further, she may locate all the routines that open a particular file – whether by file name or synonym – and all of the accounts from which that file is referenced, any triggers that are run when the file is updated, and any subroutines called from the dictionary of that file. If the file definition is held in a shared code block, she can quickly get a view of all the code in which the block has been included, and check which of those are working versions and which are just temporary copies. Perhaps someone has begun a change to the source of a routine, but has not carried it out to completion: before adding her own changes, she can check that the object code that is running matches the source.

A map can assist in support contexts also. An error message flashes on the screen: by turning to the map she may find the program that issues that specific text. An unassigned variable is reported: she can look for all programs that include that variable name. A command fails to select the expected number of records: she can trace the origin of that command. The more complete and detailed the map, the greater the payback.

Adaptive Scanning

Not all software auditing tools are alike, just as no two UniVerse business applications are the same. When selecting an auditing tool to map your applications, two factors should inform your choice: completeness and adaptability.

Completeness resides in the depth of the coverage. Consider the need to identify where a file is opened: most sites will make use of UniVerse Basic for their applications, but if you make use of PQN PROCs to perform file operations you need to ensure that the tool will parse the PROC constructs as well as those in BASIC. If your dictionary items call Basic subroutines, the auditor must be capable of showing those calls. If you have multiple copies of the source code to a routine, only with an auditor that checks the object code can you be sure which one is loaded into the global catalog.

Even so, this may only provide you with part of the story.

For real world auditing, adaptability is the key factor. You need to have confidence that the auditor will handle the conventions and structures that appear in your code. The simple token checking performed by many parsers is not enough: the auditor must be capable of adapting to specific technologies that form part of your code base. Perhaps your organization makes use of a generic routine to perform file opening – a parser that searches your code for OPEN statements will not find it. Or you may use a definition-based 4GL like SB+ or uvCase, or even your own screen runners and tools: how easily will your choice of auditor adapt to adding these definitions to your map?

Your code may already contain a lot of useful information in the form of comments, version history and standardized program headers. These can be a vital resource in building the system documentation. Even the best code parsers cannot compete with your programmers for knowledge, but sensitively extracting their comments may help to unlock that information in ways that can make it truly accessible.

The mvScan Approach

mvScan was developed to respond to the needs of real applications, many of which had been developed over long periods of time. To journey through these applications, **mvScan** offers an adaptive approach that can be tailored to suit your application. This is an iterative approach: **mvScan** is a tool to help you to build up the knowledge and map of your application.

mvScan first undertakes a voyage of discovery, mapping each of the significant elements in your application: every account, file, pointer, program, object code, PROC, paragraph, menu item, and tool. Each element is classified, named and placed into the map. At this point you can exercise your first level of control, deciding which areas should be scanned for impacts and which are known to be obsolete.

mvScan then begins the impact assessment. This takes two forms: first performing a structural analysis and armed with that knowledge it will progress to code parsing.

The structural analysis maps the physical data store: along the way it processes each account, file and dictionary. Accounts are examined for file pointers, for missing files, synonyms, Q pointers, bad or inaccessible file pointers, and to identify any files that include programs, object code or other command elements. Files are examined for sizing information, locating triggers, determining the file ownership, secondary indices and part file locations. Dictionaries are scanned to build a map of the fields, any missing (undefined) fields are reported, and file translations in the forms of T-correlatives, TRANS and XLATE expressions are parsed to build a two map of the relations between the files. Calls to BASIC subroutines from dictionary items are resolved.

Next the BASIC source code is parsed, identifying specific structures such as file OPEN, READ, WRITE and DELETE operations, calls made to subroutines or to external functions, PROCREAD and PROCWRITE statements, commands run through EXECUTE and PERFORM, links through CHAIN and ENTER, and code blocks held in INCLUDEd files. The VOC file and catalog directory are examined, multiple source locations identified, and the corresponding object code located. BASIC object code is deconstructed to give the string and variable table entries, common block usage and other statistics.

Commands are parsed and known commands – for example, SELECT, LIST and SORT statements – added to the file impacts. The auditor then turns its attention to PROCs searching out file operations. Menu and paragraphs are scanned and calls made to run or execute catalogued BASIC programs are added to the program map.

At all levels the impacts are built in both directions, allowing the developer to see from where items are called or linked as well as the routines that are called in turn: something that is difficult or impossible when tracing through source code.

Finally **mvScan** completes the initial map by parsing program headers, documentation, version stamps and other entities. These will form part of the completed documentation, and are searchable within the map: indeed, **mvScan** even promotes its own 'autodoc' style of comments as the basis for the HTML documentation it assembles on your behalf.

The map is built as a series of data files, allowing developers to use standard retrieval commands or reporting tools to locate the information within it. If you require technical documentation, **mvScan** can build the details into template-driven HTML pages that also provide a convenient means for analysts to navigate the map by following the hypertext links.

Customizing mvScan

mvScan provides a comprehensive set of auditing functionality straight out of the box. For many sites, this may be all that is required. But **mvScan** is built on the understanding that every system is different, and that a one-size-fits-all parser will never give you the depth of coverage needed for real world auditing. The key to **mvScan** is adaptability, allowing you to tailor it to your applications and to your site conventions.

mvScan was designed from the ground up to adapt itself to many differing environments, by presenting a flexible and extensible auditing framework through which you can incrementally discover your application.

mvScan provides a route through which the system is discovered and navigated. As the auditor iterates through your system each element is processed in a controlled and predictable manner, ensuring that the impact maps can be built in the correct sequence and with the necessary integrity.

mvScan invokes a series of plug-in subroutines to execute the parsing for each item processed. Standard plug-ins are provided to parse Files, Dictionaries, Basic code, PROCs and the other entries that make up the regular impact map. But the story does not end here: the plug-ins model is designed with a calling convention and template library that makes it easy to add your own custom parsing routines.

By adapting the templates provided you can easily generate your own plug-ins to identify constructs that are specific to your site or to the technologies you use. In this way, you can build up the detail in your map over time: adding support for custom tools and in-house conventions, standard libraries and routines, definitions and documentation sources. Custom plug-ins are already available for some 4GLs, standard code version stamps created using the free **mvStamp** and for specially formatted autodoc comments**.**

mvScan is a tool that will grow with your application.

The Standard Map

mvScan is both adaptive and extensible, and so the final shape of the impact maps will include details drawn from the tools and technologies employed by your organization.

As an indication, the following section details the information provided by the standard tools supplied with **mvScan**, and that forms the initial impact map assembled by the product without customizing its features.

Accounts

Synonyms
Full operating system pathname
All names for the account in the UV.ACCOUNT file
Impact keys for all files in the account.
Impact keys for program files in the account
Impact keys for directory files in the account
Impact keys for files referenced from the account
Impact keys for files containing paragraphs
Impact keys for files containing PROCs
Impact keys for files containing Menus.
File pointers that cannot be resolved.
Q pointers that cannot be resolved.

Files

Full operating system pathname
Impact keys for all accounts that reference this file
File contains programs
File contains PROCs
File contains Paragraphs
Impact keys for programs opening this file.
Impact keys for programs reading from this file.
Impact keys for programs writing to this file.
Related files.
Impact keys for files related from this file.
Impact keys for files relating to this file.
Subroutines calls from the file dictionary.
Impact keys for subroutines called from the file dictionary.
Last modification date.
Last access date.
Triggers installed on the file.
Impact keys to trigger routines on this file.
Impact keys for uvCase tools accessing this file.
Field numbers
Field name mved with FNO
Field type mved with FNO
Field description mved with FNO
Field multivalued mved with FNO
Field header mved with FNO

Programs

Full operating system pathname
Impact key to source file
Program item name
Global catalog entries
Impact keys to accounts in which it is cataloged.
Catalog names mv'ed with above.
Missing object flag
Path to object code.
Subroutines called by name.
Impact keys to subroutines called.
Indirect (unresolvable) calls.
Include files by name
Impact keys to included files.
Impact keys to files including this item.
Lines performing EXECUTE or PERFORM.
Text of command executed if resolvable.
Verb executed if resolvable.
Impact keys to dictionaries from which called (SUBR)
Impact keys to menus from which called.
Impact keys to paragraphs from which called.
Impact keys to PROCs from which called.
Other calls.
Missing or unresolved include files.
Files opened.
Impact keys to files opened.
Files read.
Impact keys to files read.
Files written.
Impact keys to files written.
Executed programs.
Impact keys to executed programs.
Impact keys to executed paragraphs.
Impact keys to programs executing this item.
Open statement lines.
uvCase tools calling the program.
File triggers from which the program is called.
Object scan date
Object code symbol names
Object code symbol types
Object code symbol flags
Object code named common blocks
Object code unnamed common size
Object code number of args

Paragraphs

Impact key of parent file.
Paragraph name
Programs called.
Impact keys to programs called.
Impact keys to programs from which executed.
Impact keys to paragraphs called.
Impact keys to paragraphs from which this is called.
Impact keys to menus from which this is called.
Unresolvable calls.

PROCs

Impact key of parent file.
PROC name
Files opened
Impact keys to files opened.
File reads
Impact keys to file reads.
File writes
Impact keys to file writes

Menus

Impact key of parent file
Menu name
Programs called
Impact keys to programs called
Impact keys to paragraphs called
Impact keys to menus called
Unresolvable calls.

Documentation

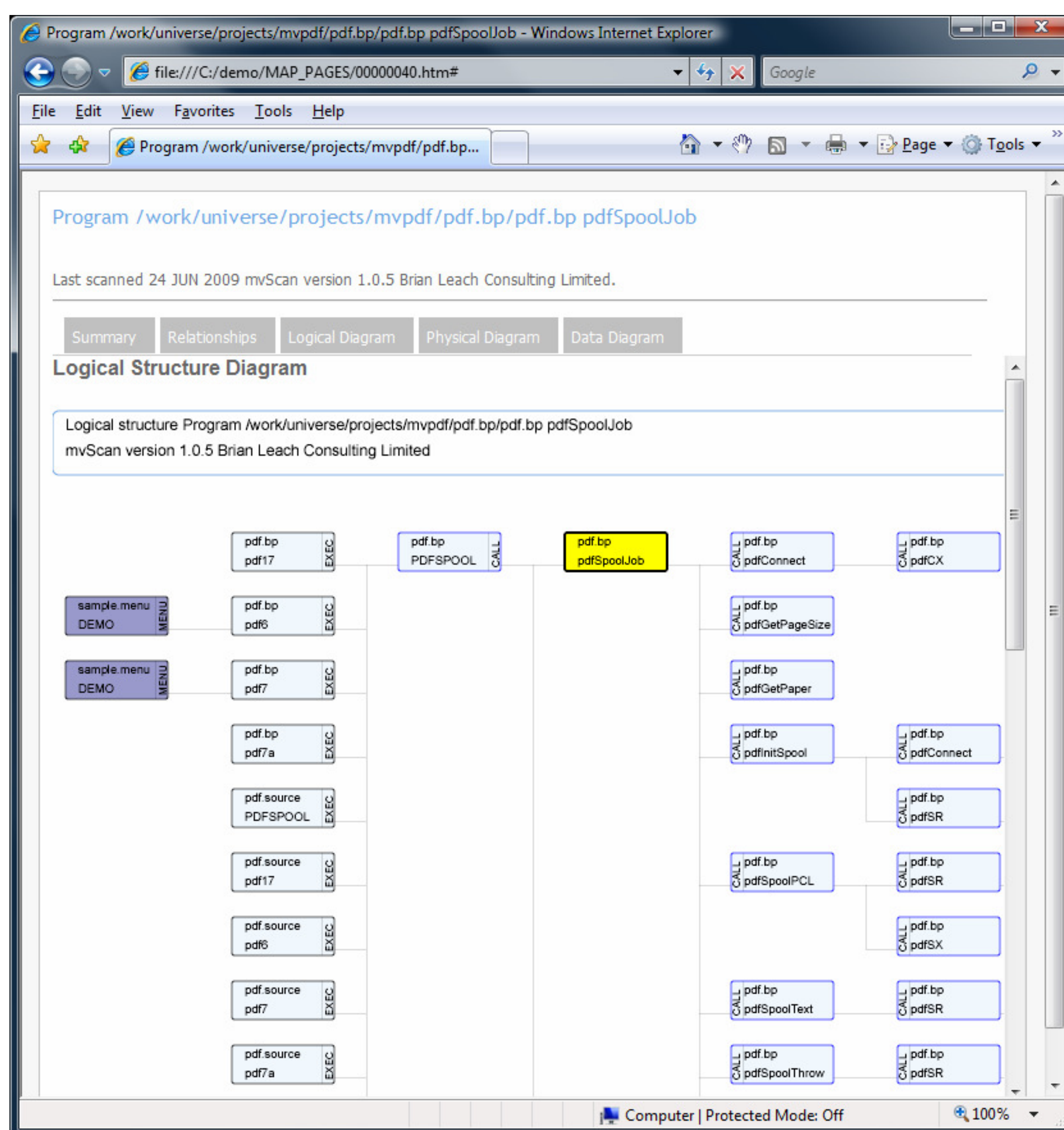
Impact key of parent file
Item name
Item type (program, paragraph, proc)
Item title
Item author
Modification history
Version number
Warnings section
Information section
Autodoc source sections
To do list section
Keywords section
Files updated section

HTML Documentation

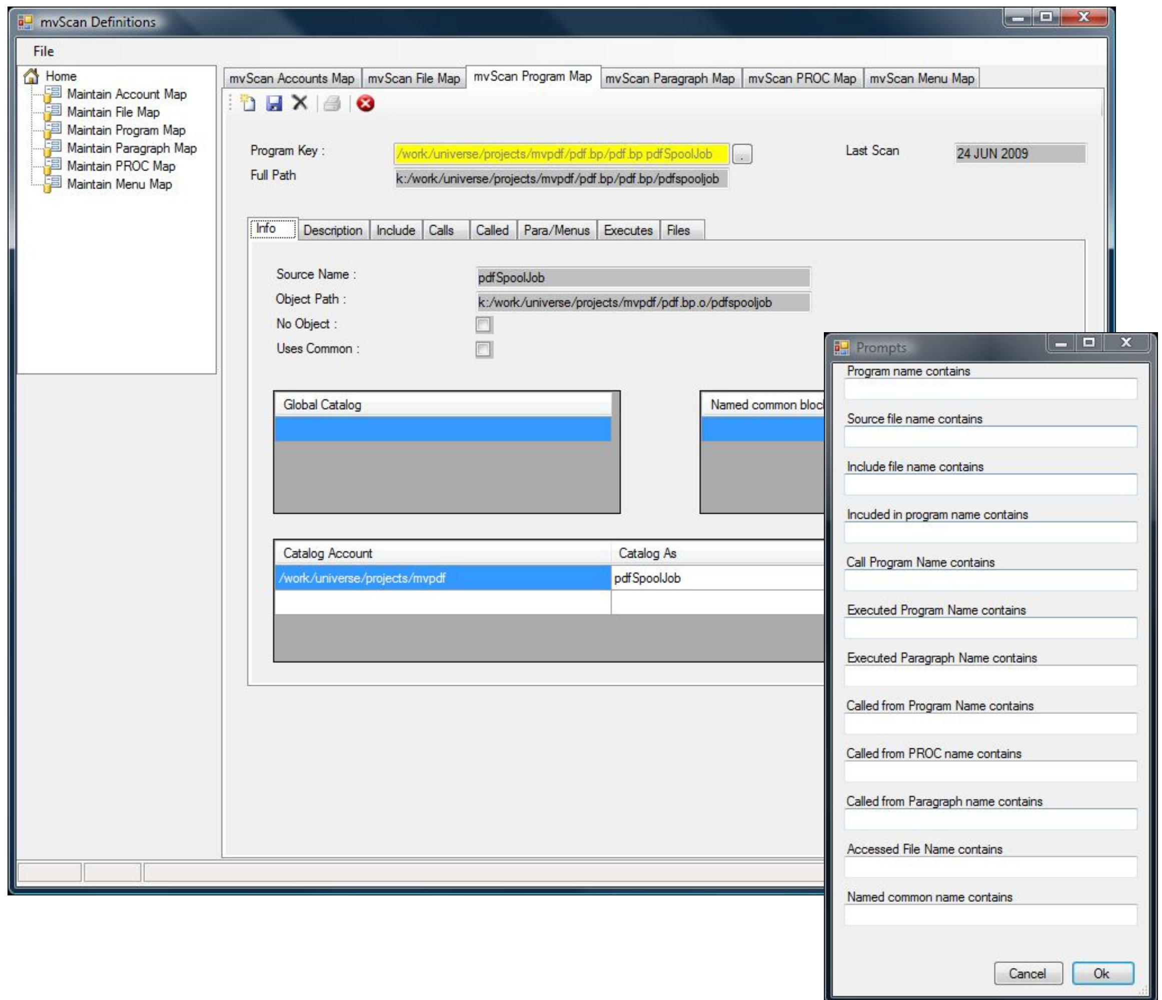
The map is designed to allow developers to run enquiries on the structure of their system. These can be more targeted and flexible than a purely graphical representation.

However, for training and introductory purposes, or to prevent consuming a database licence, mvScan will also generate system documentation in the form of HTML pages. These are linked together to provide a visual means of navigating quickly through the system by drilling up and down through the links and relationships identified by the parser.

For visual navigation, mvScan generates Scalable Vector Graphics charts with the same drill down features:



A Windows based client is also provided to offer a more direct view of the data held in the impact map:



Product Evaluations

Brian Leach Consulting Limited makes available limited period evaluation copies of its commercial products. Regrettably this is not possible for mvScan.

However, clients who are genuinely interested in discovering more about the product should contact Brian Leach Consulting Limited using the contact details below, to discuss opportunities for undertaking a trial standard scan of their application.

Further Information

For further information and assistance, or to request new features, please contact:

Brian Leach Consulting Limited
Web: http://www.brianleach.co.uk
Email: info@brianleach.co.uk