

Extracting Business Logic from existing COBOL programs as a basis for Redevelopment

Harry M. Sneed

Case Consult GmbH

Flachstraße 13
65197 Wiesbaden
Germany

harry.sneed@caseconsult.com

Abstract

The following paper describes an industrial pilot study carried out to regain the business knowledge embedded in a legacy COBOL application. The goal of the project was to ween out the information required to re-implement the ancient host programs in a new client/server environment. The solution followed a four step progress. First, the programs were restructured, secondly the programs were sliced into business logic modules, third the business modules were subjected to a multi view analysis and finally the views were integrated into a unified documentation describing the data, decision and procedural flow of each program slice.

Keywords:

Reengineering, Reverse Engineering, Knowledge Extraction, Business Rules, Program Comprehension, Slicing

1. Background of the Project

The work described here has taken place within the scope of a large scale redevelopment project at a German savings and loan bank. Originally, it was the goal of the project to reengineer and reuse the existing code, however, this approach was soon abandoned due to the poor quality of the code.[1] Instead it was decided to redevelop the application using modern object-oriented techniques. This decision called for creating a new functional specification and a new architectural design.

Developing a new functional specification for an existing system turns out to be a challenging task, especially if the domain knowledge of the old system has been lost, as is often the case after many years. It is interesting to observe how users become dependent on information processing systems without knowing how they really function. In this case, as in many other

similar cases, there is no detailed documentation, only a very high level description of the system architecture and this too is neither complete nor up to date. As noted by others before, the only real description of what a program is doing, is the program itself. [2]

Of course, there are still a few old programmers around who may know what the one or the other program was intended for, but their knowledge is fragmented and incomplete. There remain many black holes. If a description of the detailed business logic is to be recovered, then it must be recovered from the programs as the only reliable source of information.

The extraction of business logic from existing legacy where many different approaches ranging from procedural to data driven to object-oriented have been described before.[3] However, all of these approaches assume that the programs are in some understandable form. In the case at hand the programs are written in COBOL, but the programming style is at an Assembler level. The modules are large and complex, the control logic is driven by GO TO branches, the names are 8 character Assembler like codes, there are hardly any comments, there is no locality of reference and the processing logic is mixed with the transaction control logic for CICS and a hierarchical database logic based on IMS. As such, it is extremely difficult to identify and isolate the business logic.

In such a case, there is little point in re documenting the programs as they are, because any documentation extracted from the programs can be no better than the programs themselves. An unstructured, monolithic and unreadable program will provide an unstructured, monolithic and unreadable documentation. Besides, the business logic will still be hidden behind the transaction processing and data access logic. There can often be no reverse engineering without first reengineering. [4]

2. State of the Software

The state of the software in question is typical for many legacy application systems in industry. The programs are not only unstructured, but monolithic, unstructured, uncommented and incomprehensible. Besides the business logic is highly intertwined with the teleprocessing, map manipulation and database access logic. CICS/COBOL programs are by nature event driven. They are in fact, subprograms of the teleprocessing monitor. Maps are received, the contents checked and depending on the content, specific business functions triggered. The business functions contain data base accesses both to a hierarchical database-IMS as well as to relational databases-DB2. In addition, flat files are accessed via CICS I-O macros.

The database accesses are enclosed in DLI or SQL EXEC macros. The map and file as well as other control operations are enclosed within CICS EXEC macros. Thus, the COBOL language is enhanced by three other languages – DLI, SQL and CICS. These macros are scattered throughout the code and obscure the business solution.

The prevailing technique for connecting portions of code is via the GO TO branch. Loops are implemented with backward GO TO branches to one or more starting points and with exits to one or more labels outside of the loop. Loops can only be identified by recognizing the backward GO TO branches. Overlapping GO TO branches, or knots, have created a very fragile situation, where the slightest change can destroy the control structure. Thus, removing GO TO's by traditional methods is highly dangerous. Most statements, including IF statements, are terminated by a period, making the code relatively flat. These features reflect the Assembler style of programming in the original programs where most of the decisions are made at the beginning of a code block before taking any actions.. Occasional structured statements such as EVALUATE and IF...END-IF indicate more recent patches.

As with most older COBOL programs there is no information hiding or abstract data types. Each program has an extensive global area with thousands of variables of which only a few are really referenced, as a rule less than 10 %. Often entire data structures are copied in only to address a single elementary item.

On the positive side, the procedure divisions are segmented into sections. GO TO's seldom branch outside of a section, but instead are collected at the end mode. This makes it possible to separate sections from another at least as far as the control logic is concerned. Therefore, the programs do lend themselves to procedural slicing. They can be split up into a main section and several subsections.

Another positive aspect is that most of the database accesses are separate subroutines at the end of the program which can be invoked from anywhere else within the program. This makes it possible to isolate the database access logic from the business and the presentation logic. This also has a positive effect on the program documentation since it is easier to identify the objects accessed.

3. The Logic Extraction Method

Once an organization has decided to redevelop it's applications, one of the first steps is to extract the business logic from the existing programs.[5] This was the subject of a pilot reverse engineering project conducted by the author at the customer site. The major portion of the project was devoted to the development of a set of tools to support the automated knowledge acquisition process, a process consisting of four sequential steps. The first step was to restructure the procedural code to facilitate slicing. The second step was to slice the code into partial programs each processing a discrete business rule. The third step was to generate a set of views on each partial program. The fourth and final step was to integrate these disjointed views into a single unified business rule documentation. (See Figure 1)

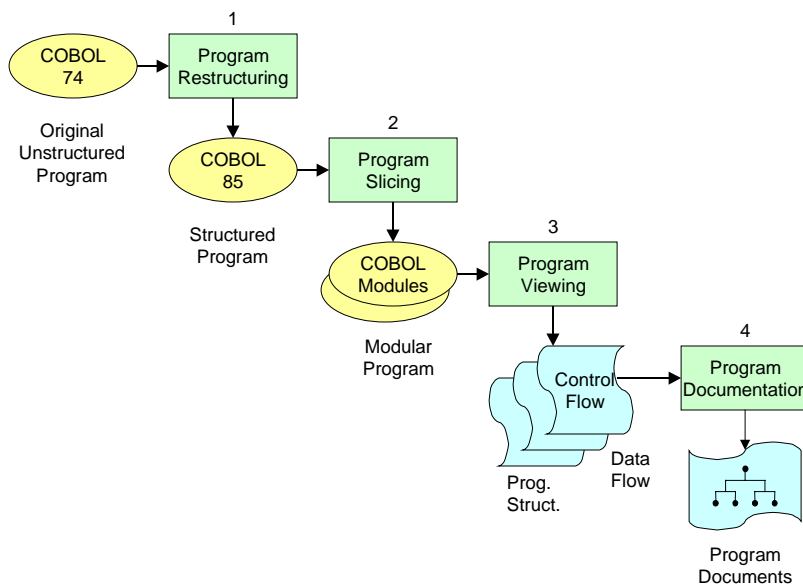


Fig. 1: Four Step Approach to Logic Extraction

3.1 Program Restructuring

The first step of the process - program restructuring - was fully automated. The source code was submitted to the tool „SoftRecon“ which has been described before in previous papers. [6] SoftRecon performs six vital restructuring functions:

- it reformats the procedural code splitting the lines with more instructions and indenting the nested code,
- it cuts the input/output operations and database accesses out of the mainline code and pastes them on the end of the program in a separate data access section,
- it removes obsolete and dangerous statement types such as the PERFORM THRU, ALTER and NEXT SENTENCE replacing them with standard statements,
- it removes the periods at the end of the IF statements, replacing them with END-IFs for each IF..ELSE pair,
- it removes all of the GO TO branches, replacing them with a label variable assignment to PERFORM the paragraph to which the GO TO is branching without returning,
- it recognizes backward branches and converts them to a PERFORM UNTIL loop construct.

Each of these functions is performed in a separate pass through the program. After each pass the source is left in another more structured state. The final state is a purely structured program with no GO TO branches, a nested decision logic and a three layer architecture consisting of

- a control layer,

- a processing layer and
 - a data access layer.[7]
- (see Figure 2)

3.2 Use Case driven Program Slicing

In the second step human intelligence was required to identify the logical entry points, i.e. those points where the processing of a particular use case begins. This can be a function, a label or a procedure entry. A use case was usually related to the main section with several sub sections. The tool user needs only to mark the source line where the slice begins. This could be the point where an input panel is received or it could be the beginning of a processing loop. The rest is taken care of automatically by means of a recursive invocation algorithm in the tool COBWrap. Each PERFORM from the original code slice is pursued to include that path in the slice. If the included slice contains additional references to other code slices these too will be pursued and the affected code included until all PERFORMs have been resolved.

After the source segment has been cut out of the procedure division, a data flow analysis is performed to recognize all data variables processed by that segment. These variables are then marked in the Data Division together with the structures they are included in to create a new reduced Data Division for the sliced code containing only those variables used by that code. The same process is repeated for the files in the Environment Division. The end result is a partial program consisting

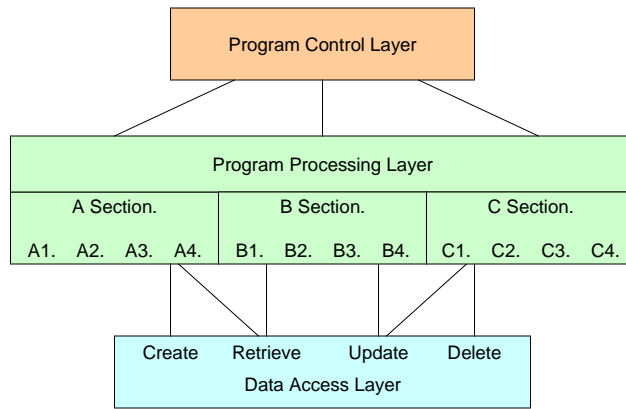


Fig. 2: Three Layered Restructured Program

of all procedures and data required to process a given use case. The reduced data structure is placed in the Linkage Section to be passed as a parameter from the calling program.

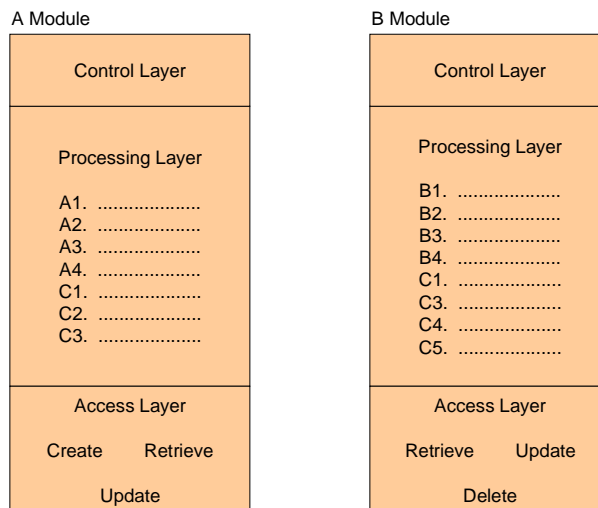
This second step is repeated for each unique use case, so that there are several partial programs created from the same original program. Code traversed by many rules is duplicated and included in each partial program. The underlying technique is that of procedural slicing introduced by Weiser [8] and extended by Cimitile and others.[9]

From the 14 sample programs in the pilot study 39 slices or partial programs were generated giving an average of 3 use cases per program. The final code volume was 3.6 times greater than the original size showing that much of the procedural code is common to several slices and that many data variables are used in different slices. This is a similar expansion to that obtained by class flattening in C++.[10] The purpose of the slicing here is, however,

not to optimize the code by removing redundancy, but to facilitate comprehension by collecting and ordering all code relevant to the processing of a given use case so that each case can be documented separately. Therefore, many of the same code sections are included in different modules as depicted in Figure 3.

3.3 Program Viewing

In the third step all of the partial programs extracted from the original source were submitted to an automated documentation process. This was done with the tool **COBAnal** which generates five views on each of the partial programs. A view corresponds to a document type highlighting certain features of the program. The first view is focused on the overall structure of the program. It provides an indented tree of all sections and paragraphs involved in the processing of a particular business rule plus their **PERFORM** and **GO TO** references to one another. This view amounts to a graphical table of contents.



C Functions performed by A & B are included in A & B

The second view is that of the external interfaces of each partial program. These include the copy references, the subprogram calls, the entries, the input/output operations, the interactions with the teleprocessing monitor and the database accesses. This view reveals the incoming and outgoing data flow at the program level.

The third view presents the partial program logic. It contains all of the decision modes – IFs, PERFORM loops, case selections, GO TO branches and labels in conjunction with their dependent statements or branches. The logic is represented in a structogram format with nested control blocks. This view corresponds to a decision tree, depicting the path to each branch. [11]

The fourth view is that of the internal data flow. For each paragraph and/or section, the input, output and conditional variables are listed to the left, to the right and below the paragraph affected. Inputs or arguments are listed to the left, outputs or results are listed to the right and predicates are listed in the middle. By means of a topological sort, these tables are converted to a data flow graph for tracing the flow of individual variables or data groups through the programs. This is equivalent to data slicing.[12]

The fifth and final view is the data structure of each partial program. This view includes the data declarations, the level, picture, usage and value clauses of only that data referenced in the slice at hand. It represents a subdomain of the total data domain, namely that subset of data required to process this particular use case. (see Figure 4)

3.4 Program Documentation

The union of these five views provides a multidimensional representation of the procedural structure, control flow, data flow and data structure of each partial program for each business rule. The fourth step is aimed at aggregating these views at the transaction, subsystem and system level. The views are analyzed and their contents stored in a relational database from which overall graphical documents are generated.

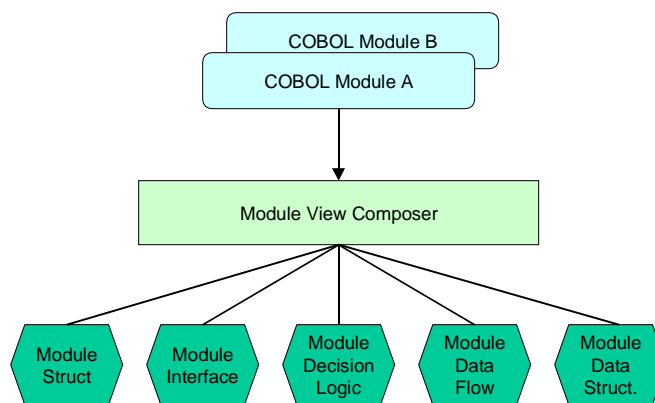
The database tables depict binary relationships between source and target entities including relation type and cardinality. For instance a control flow relation is that between a predecessor and a successor mode. The cardinality is conditional or non-conditional. The relation type is the governing condition, e.g.

Mode-B → Mode-A
if X > Y

Data flow relations are pairs of functions and variables with the types input, output and inout. Cardinality is given to note how many occurrences of the data there are.[13]

In the end the software repository contains a set of 16 relational tables

- System to Program (SYSPROG)
- Program to Module (PROGMOD)
- Program to Database (PROGDB)
- Module to Module (MODMOD)
- Module to Section/Paragraph (MODSECT)
- Module to Data Object (MODDATA)



Views of each Module from different Perspectives

Fig. 4: Module View Composition

- Module to Copy Member (MODCOPY)
- Module to Map (MODMAP)
- Module to File (MODFILE)
- Module to Database (MODBASE)
- Module to Data Items (MODATTR)
- Section to Data Items (SECTDATA)
- Section to Decision Modes (SECTCOND)
- Data Item to Data Object (DATAOBJ)
- Data Object to Data Object (OBJOBJ)
- Data Object to File/Database (OBJDB)

From these relations almost any kind of system document can be generated. In the project in question

- calling hierarchic trees, (see Appendix 4)
- data flow diagrams, (see Appendix 2)
- data usage diagrams, (see Appendix 3) and
- control flow diagrams (see Appendix 1)

were generated and supplemented by the comments taken from the source code. Once the relationships are stored in the database it is also possible to navigate through the repository and trace control and data flow across program boundaries, as well as to pose queries pertaining to impact analysis. [14] The general structure of the repository is depicted in Figure 5.

4. Project Results

The ultimate question is to what extent do such documents and repositories aide in program comprehension. The answer to that question depends on the person to whom that question is put. [15] In the project discussed here, the project was divided up into two groups. One group was made up of older programmers who had been maintaining the programs for years. Their immediate reaction was to claim the views and documents extracted from the source gave them no additional information they could not obtain

from the source directly.

The other group was made up of younger programmers whose task it was to rewrite the old programs in a new object oriented language. They were grateful for the additional information provided by the views and repositories. They were further grateful to have the programs sliced and the slices documented because this helped them to better isolate and understand the business logic.

It would seem from this observation that the question of documentation and information value is dependent on the receiver and his objectives. If his objective is to re implement the business logic in another context and he is not familiar with the old programs, then he will regard documents extracted from the code as a useful means of comprehending the business logic, especially if the documents represent a use case oriented view of the program.

If, however, the subject is familiar with the code and carries his knowledge of the business functionality around in his head, then the documents and repository will not tell him much that he does not already know.

The added effort of comprehending the documentation and navigating through a repository will seem to be an unnecessary burden with no added value. This tends to be the subjective perception of the knowledgeable programmers even though an objective view of the documentation provided might expose to them some aspects of their code of which they were not aware. They tend to believe they already know all they need to know about the system. This attitude often results in a rejection of the information provided by a process such as the one described here.

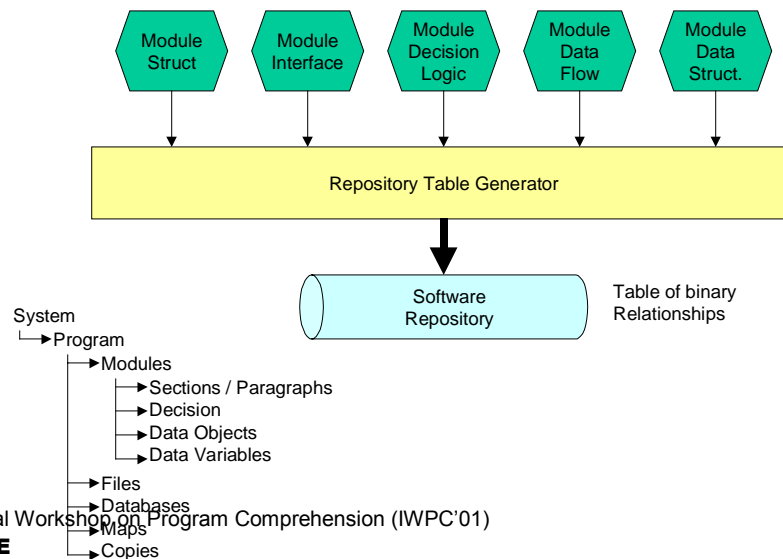


Fig. 5: Generation of Software Repository

In fact the attitude of the older programmer group led to the cancellation of this knowledge acquisition project in spite of the acceptance by the younger programming group. The older programmers claimed the younger programmers could come to them for any information about the old applications. This way they could retain their importance.

5. Conclusions

Knowledge of business functionality in software applications is seldom documented in industry. It exists either in the heads of the responsible programmers or in the code itself. A representation of that logic can be extracted from the code and presented to the persons responsible. If they are not familiar with the code they will accept the extracted information and will use it to re-implement the business logic in another context. If they are familiar with it they will tend to reject it and to re-implement the business logic out of their heads. This, in turn, causes them to neglect many important details which may be important for the re implementation.

The conclusion of this real life experiment in program comprehension is that those programmers which have been maintaining a system for many years are not the best candidates for re-implementing it in another environment for three reasons

- first**, they tend to overestimate their own knowledge of the business logic,
- secondly**, they confuse the current solution with the business problem to be solved,
- thirdly**, they believe that their existing solution is optimal whereas in reality it is usually dependent on the environment they have implemented it in.

For these reasons, it is recommended to re-implement legacy systems in a new environment with new programmers unfamiliar with the existing solutions, who use knowledge extraction tools to recapture the real business logic.

References:

- [1] Sneed,H.: “Generation of stateless Components from procedural Programs for Reuse in a distributed Environment” Proc. of 4th European Conference on Software Maintenance and Reengineering, IEEE Computer Society Press, Zürich, March 2000, p. 189
- [2] Corbi,T.: “Program Understanding – Challenge for the 1990’s”, IBM Systems Journal, Vol. 28, No. 2, 1989, p. 294
- [3] van Zuylen, H.J.: The REDO Compendium – reverse Engineering for Software Maintenance, John Wiley & Sons, New York, 1993, p. 225-248
- [4] Sneed, H./Jandrasics,G.: “Inverse Transformation of Software from Code to Specifications”, Proc. of ICSM-88, IEEE Computer Society Press, Phoenix, Oct., 1998, p. 102
- [5] Hanna, A. “Getting back to requirements proving to be a difficult Task”, IEEE Software Magazine, Oct. 1994, p. 49
- [6] Sneed, H.: “Architecture and Functions of a commercial Software Reengineering Workbench”, Proc. of 2nd European Conference on Software Maintenance and Reengineering, IEEE Computer Society Press, Florence, March, 1998, p. 2
- [7] Sellink, A./Sneed, H./Verhoef, C.: “Restructuring of COBOL/CICS legacy systems” Proc. of 3rd European Conference on Software Maintenance and reengineering, IEEE Computer Society Press, Amsterdam, March, 1999, p. 72
- [8] Weiser, M.: “Program Slicing”, IEEE Trans. on S.E., Vol. 10, No. 4, July, 1984, p. 352
- [9] Cimitile, A./ de Lucia, A./ Munro, M.: “Identifying Reusable Functions using Specification driven Program Slicing” Proc. of ICSM-95, IEEE Computer Society Press, Nice, Oct. 1995, p. 124
- [10] Binder, R.: Testing object-oriented Systems, Addison-Wesley Pub., Reading, 1999, p. 215-222
- [11] Sneed, H./Erdoes, K.: “Extracting Business Rules from Source Code”, Proc. of 2nd IWPC-96, IEEE Computer Society Press, Berlin, March, 1996, p. 240
- [12] Canfora, G./ Sansome, L./ Visaggio, G.: “Data Flow Diagrams – reverse Engineering Production and Animation”, Proc. of ICSM-92, IEEE Computer Society Press, Orlando, Nov. 1992, p. 336
- [13] Sellink, A./ Verhoef, C.: “An Architecture for automated Software Maintenance”, Proc. of 7th IWPC-99, IEEE Computer Society Press, Pittsburgh, p. 218
- [14] Sajaniemi, J.: “Program Comprehension through multiple simultaneous Views” Proc. of 8th IWPC, Computer Socitey Press, June, 2000, Limerick, p. 99
- [15] Balmas, F./ Wertz, H./ Singer, J.: “Understanding Program Understanding”, Proc. of 8th IWPC, IEEE Computer Society Press, June, 2000, Limerick, p. 256

Appendices:

Appendix 1: Business Logic Structure

1	02475	IF NEXT-METHOD-ID = 'E02300'
2	02476	@MOVE SPACES TO NEXT-METHOD-ID
2	02480	PERFORM UNTIL NEXT-METHOD-ID NOT = SPACES
3	02482	IF WMC01-MCNAME-EIN(WMC01-IND1) NOT = SPACE
4	02483	@*ADD 1 TO WMC01-IND1
4	02485	@*CONTINUE
3	02487	ELSE
4	02487	@*MOVE 'E02300' TO NEXT-METHOD-ID
3	02489	END-IF
2	02490	END-LOOP

Appendix 2 Data Flow Table

PROCEDURE/METHOD: E02100		
INPUTS/ARGUMENTS	PREDICATES	OUTPUTS/RESULTS
IAU02-QPLZ1	WMC01-MCPLZ-1	WMC01-MCPLZ-1
WMC01-MCPLZ-EIN (WMC01-	WMC01-MCPLZ-1	WMC01-MCPLZ-1
	WMC01-IND0	WMC01-MCPLZ-2
	WMC01-MCPLZ-EIN (WMC01	WMC01-IND0
	WMC01-MCPLZ-EIN (WMC01	WMC01-MCPLZ-NUM (WMC01-
		WMC01-IND0
		WAS00-QMELDNR
		MMC02-QMCPLZ1L
		INV01-QFEHLER

Appendix 3 Reduced Data Structure

INOUT	01	PZ-BEREICH.
OUTPUT	05 J01-QGRDNRI	PIC X(8).
COM /*		GRUNDNUMMER - EINGEBEFELD FUER
COM /*		DAS PRUEFZIFFERN-MODUL UQAS01K
INPUT	05 J01-QGRDNR	PIC 9(8).
COM /*		GRUNDNUMMER (INCL. PRUEFZIFFER
OUTPUT	05 J01-QPZACOD	USAGE DISPLAY
OUTPUT		PIC X.
COM /*		ANFORDERUNGS-CODE FUER DAS
COM /*		PRUEFZIFFERNMODUL UQAS01K
COM /*		1 = ANFUEGEN PRUEFZIFFER
COM /*		2 = PRUEFEN PRUEFZIFFER
COM /*		3 = ERMITTELN NAECHSTE GRUNDNR

Appendix 4: External Object References:

02213			MATCHCODE-VERARBEITUNG SECTION.
02371			E02 SECTION.
02435	INOUT	CALL	UDAS06K
02435	PARAM		JQAS06K
02545			AUSGABE-AUFBEREITUNG SECTION.
02668			DATENBANKSCHNITTSTELLE SECTION.
02712	INOUT	CALL	TSQUEUE
02712	PARAM		MQMC02K
02776	INPUT	SQL	FETCH CBS310T
02814	INOUT	CALL	PQTA09K
02814	PARAM		IQTA01K
02824	INPUT	SQL	SELECT BS310T