An Investigation: Reengineering Sequential Procedure-Driven Software into Object-Oriented Event-Driven Software through UML Diagrams

Richard Millham
Telus Communications Inc
Richard.Millham@shaw.ca

Abstract

Reengineering a COBOL legacy system is a difficult multi-step process, particularly when the COBOL legacy system is a sequential procedural-driven system which is being reengineered into an object oriented, event-driven system. In this scenario, it is necessary to analyse the legacy system in order to identify possible objects with their attributes and methods within the code and to determine how the legacy system's variables and procedures inter-relate in order to model pseudo-events from strictly sequential code. The focus of reengineering is too often based on theory rather than based on experience gained from real-world examples. This paper hopes to address this imbalance by providing a practical application of reengineering to an actual legacy telecommunications system.

Keywords: Wide Spectrum Language (WSL), Unified Modeling Language (UML), Reverse Engineering, COBOL Legacy Systems, Reengineering

1. Introduction

Throughout the world, there are many legacy systems, written in COBOL, which were designed many years ago but still function today. There are many reasons for their continued operation but these reasons include the high cost of developing a new system, the fact that these systems fulfill critical business functions for the business, and the fact that the business cannot afford the system downtime that switching to and testing of a new system would involve.

The example legacy system that is used in this paper constitutes one of the core business systems used by Telus, a Canadian communications company. These legacy systems were developed during the 1960's and 1970's as procedurally-driven, mainframe-based programs. Maintenance changes over the years obfuscated the original system's design which made it difficult to modify these systems to handle new user requirements, such as data services. Developing new systems to replace these existing systems was hampered by the high cost of new system development and by the real threat to billing deadlines posed by system downtime during the system switchover. The only feasible alternative to this dilemma is to re-engineer the legacy system into a more modern representation, such as an event-driven, object-oriented system. An object-oriented system, with any code changes limited to its encapsulating object, promises fewer side effects and lower maintenance costs than the global-scoped and globally impacting COBOL variables.

The first step in transforming a legacy procedural system into an object oriented system is to identify data clusters within the existing data source code. These data clusters will then be transformed into class objects; variables and procedures will be assigned as attributes and methods of a specific class object. Any sharing of variables or procedures among class objects will be represented as associations between class objects.

This cluster analysis should be seen as one step in a difficult multi-step process of reengineering a COBOL legacy system. Often the approach taken is to develop one's own tools and intermediate language for this reengineering rather than trying to make use of existing tools and existing languages. This paper adopts the latter approach; we build upon other's work in the field such as the Wide Spectrum Language (WSL) for intermediate language representation and the Unified Modeling Language (UML) for documentation and code generation. Our approach is to reverse engineer the source COBOL code into a WSL representation that is then transformed into a UML format. This UML format can then be exported to other UML tools for documentation and re-engineering purposes.

2. Our Investigation

We first convert the COBOL legacy system into WSL using a set of COBOL to WSL conversion rules that were formulated using Martin Ward's [6] paper, *The Syntax and Semantics of the Wide Spectrum Language,* which defined the basis of the Wide Spectrum Language, and Liu's [3]thesis, which extended WSL to encompass common programming constructs like if-then-else and while-do statements. After the legacy source code has been converted into a WSL representation, this WSL representation must, in turn, be analysed in order to locate objects, their associations, and their methods.

In order to discover possible objects from within the WSL code, we developed object identification methods modeled on those of Gall, Burd, and Newcomb [1][2][4]. Gall identifies potential objects from source code by measuring the degree of coupling between different entities such as procedures and variables. If two procedures have a high degree of coupling, or interaction among themselves, these two procedures probably should be assigned to the same object class. Gall also uses data dependencies between procedures and variables in order to identify potential objects. If two variables share a common data dependency, such as both of them belonging to the same record structure, these two variables should probably be assigned to the same object class. [2]

Using one of van Deursen's object identification techniques, we determine which procedures have a high fan-out (procedures which call many other procedures) and which procedures have a high fan-in (procedures that are called by many other procedures). [5] Procedures with a high fan-out are usually control modules and thus should be kept in a separate control class. These procedures with a high fan-in usually perform a technical function, such as error logging, and should likewise be put in their own separate class. I propose in expanding van Deursen's method to incorporate variables which are used as parameters to these high fan-in procedures; these variables are assigned values which are never assigned to other variables nor updated to the database. Likely, these variables serve as an error recording function in conjunction with the procedure and, thus, should be eliminated from any further data analysis.

Although combining these error-logging functions within their most often-used class object may reduce the number of inter-object couplings, it has the undesirable effect of increasing coupling between the class containing these error-logging functions and other classes whose only commonality with the former class is through the logging functions. Hence, an obscuration of the real purpose of the function within the system design is created when the high fan-in and fan-out procedures are included in the data clustering analysis.

Newcomb and Kotik [4] use an object identification methodology which take COBOL records as a starting point for classes and use COBOL record fields as attributes for these classes. However, during the long maintenance cycle of a COBOL program, new record fields have a tendency to be aggregated in existing records rather than be placed in new, smaller, and more relational records.

Thus, given a sample WSL program, I propose to determine the effect that the level of atomicity chosen, whether record level or record field level, for data analysis has on class identification and the degree of coupling.

The first step of this proposal is to scan the program code. Variables that are used within WSL control structures, such as "if" or "while" statements, are classified as control variables. Variables that have been used in WSL data assignment statements are classified as data variables. Control variables are further subdivided into control variables used as guards for WSL constructs that call other procedures and those that do not. This subdivision is used in a UML representation of this WSL program where the latter variables are used to represent guards for transitions in intra-procedural statecharts and the former are used to represent guards for events that invoke these procedures.

Regardless of their classification as data or control variables, variables which are used within WSL procedures will be assigned to objects whose attributes or variables they access the most frequently within the procedure. If a variable is assigned to one class object and a procedure, assigned to a different class object, accesses that variable, an object class association is created between the two object classes. If all the procedures of the non-owning object class access the variable of the owning object class more than once, it is a many-to-one association. If these procedures access this variable only once, it is a one-to-one association. After these tasks have been completed, the WSL representation is then transformed from that of a strictly sequential, procedural program to an object-oriented one.

3. Conclusion

By reengineering the legacy system to a UML representation, we hope to enable the system's maintenance personnel to gain a fuller understanding of the system and consequently, to enable them to better redesign the system to meet changing user requirements. One of the first steps of this multi-step reengineering process, and the purpose of this paper, is to analyze the source of the legacy system in order to identify possible objects and their associations. After analyzing control flow and data flow usage within the sample WSL program, we use a dissimilarity/similarity matrix in order to assign related procedures and variables to the same class object. In order to provide a uniform basis for comparison, three class objects for each experiment were selected.

3.1 Procedures with High Fan-In and Fan-Out

Comparing the class objects identified and the degree of coupling between these objects produced using an analysis of all procedures versus that produced using an analysis of those procedures without high fan-in or fan-out, the former approach seems to group the majority of procedures into one large object with the remaining procedures grouped into two much smaller objects.

According to the data analysis performed on our given WSL program, one procedure had a high fan-in (procedure U076) which commonly utilized five variables as parameters; these variables were assigned literal values yet, other than their use in the procedure call, played no further role in the program's data flow. Visual examination of the code confirmed our suspicions that the variables served to record a particular error state with the high fan-in procedure, U076, serving as an error logging function.

Further data analysis revealed that two procedures had a high fan-out (2000-APPLY-TRANS-TO-MASTER and CR708V07_VAR-INIT); again, visual examination of the code confirmed our suspicions that the procedures served as an initialization/main calling procedure. Eliminating the high fan-in and fan-out procedures from further data analysis helped prevent skewing our data analysis.

3.2 Data Analysis With Record Field Atomicity Vs Record Atomicity

All of the variables, whether control or data, which are shared between more than one procedure were part of a record structure. Had these record structures formed the nucleus of our classes with the record structure forming a closer coupling factor than actual usage of its record, the object identification would be quite different.

Function driven couplings (procedure calls) between procedures remain the same however, shared variables are treated differently. Whereas in record field level atomicity, variables are assigned to the procedure which utilizes them the most; in record level atomicity, records are assigned to the procedure which utilizes them the most and any fields of these records are, therefore, assigned to this procedure regardless of their individual level of usage by that particular procedure. As a result, the class structure looks quite different from that if the level of atomicity was at the record field level; furthermore, the degree of coupling is more evenly spread among the objects (50% among all three classes as opposed to 100% between two classes for record field level atomicity).

3.3 Shared Variables

From our variable usage analysis, we determined that very few variables, 6 out of 135 or approximately 4%, are shared among procedures. However, half of these shared variables act as control variables. These control variables are used to signal a particular state or invoke an event. Of these control variables, a majority (approximately 66%) was used to invoke a method outside of the attribute's owning class.

Consequently, it seems that few variables are shared among the procedures. Half of those variables that are shared are used to signal a particular state or event among procedures. Thus, by utilizing these control variables as signals for a particular event occurrence, this procedural-driven WSL program can be transformed into an event-driven program.

References

1) Burd, Elizabeth, Malcolm Munro and Clazien Wezeman "Analyzing Large COBOL Programs: the extraction of reusable modules". International Conference on Program Comprehension 1997, May 1997, pp 401-410

2) Gall, H.W. Eixelsberger, M. Kalan, M. Ogris, H. Beckman, B. Bellay, and H. Gall. "Recovery of architectural structure: a case study. " in *Development and Evolution of Software Architectures for Product Families* (Second International ESPRIT ARES Workshop, Las Palmas de Gran Canaria, Spain), pages 89-96, F. van der Linden, editor. Springer-Verlag, LNCS 1429, February 1998.

3) Liu, Xiaodong "Abstraction: A Notation for Reverse Engineering", Ph.D. Thesis, De Montfort University, England, 1999.

4) Newcomb, P and Kotik, G. "Reengineering procedural into object-oriented systems" in Second Working Conference on Reverse Engineeing, WCRE95 (1995), IEEE Computer Society, pp. 237-249.

5) van Deursen, Arie, Tobias Kuipers "Identifying Objects using Cluster and Concept Analysis" Proceedings 21st Int'l Conf. on Software Engineering, 1999.

6) Ward, Martin, "The Syntax and Semantics of the Wide Spectrum Language", *Technical Report*, Durham University, England, 1992.