Appeared in Proceedings of the 7th IASTED International Conference on Software Engineering and Application (SEA), 2003. (Preprinted-Version)

Acquiring Domain Knowledge in Reverse Engineering Legacy Code into UML

Jianjun Pu , Richard Millham and Hongji Yang
Software Technology Research Laboratory
De Montfort University
England
hyang@dmu.ac.uk

Abstract

The focus of this paper is on the systemization of the acquisition of domain knowledge during the process of reverse engineering legacy systems into UML. The domain knowledge of a legacy system is derived from this system's data environment, input and output data, documents, stakeholders, specific programming language(s) used, source code, and underlying operating system. This knowledge is applied when modelling the static and dynamic aspects of the legacy system. Because class diagram and use case diagram are the most important in reverse engineering legacy system to model its static and dynamic aspects respectively, the domain knowledge is adopted in the realisation of class diagrams and use case diagrams of legacy system.

Key Words

Reverse engineering, Unified Modelling Language (UML), legacy system, domain, domain engineering, domain analysis

1. Introduction

Reverse engineering is the process of first analyzing a subject system in order to identify the system's components and their inter-relationships and then creating the representations of the existing system in another form or at a higher level of abstraction. One of the goals of reverse engineering is to support program comprehension and to extract the abstraction of a legacy system. UML offers the advantage of providing multi-perspectives of the same system model with each perspective addressing the different needs of different groups [2][4].

Legacy systems were designed to deal with business and these systems modelled the business rules of the problem domain in which they resided. Furthermore, every legacy system is located within its specific commercial domain. If the domain of legacy system is not uncovered, it is difficult to understand it. Consequently, in order to understand a legacy system, the domain knowledge of the legacy system must be addressed during its reverse engineering process.

Although legacy software code was written in a specific programming language, because this code was designed and developed in regards to its specified business area, the code is domain-related and not isolated [20].

The research in this paper focuses on acquiring the domain knowledge during reverse engineering legacy system into UML. The domain knowledge is achieved from the operating system in which the legacy system is performed, its data environment, the programming language in which the legacy code was written, the input and output data of the legacy system, the hardware that executes the legacy system, the documents of the legacy system, the stakeholders of the legacy system, and legacy code itself. It is applied in the models of static and dynamic aspects of legacy system. Because the class diagrams and use case diagrams are the most important in reverse engineering legacy system to model its static and dynamic aspects respectively, the domain knowledge is adopted in the realisation of class diagrams and use case diagrams of legacy system.

2. Achieving Domain Knowledge from Legacy Code

2.1 From Parameters

The parameters of legacy code are important for acquiring domain knowledge from legacy code. They include names, types, and lengths.

A parameter name represents the significance that the parameter represents within the legacy code of that domain. Its name often provides the real-world meaning of that parameter within the problem or operational domain. It provides a clue to its place and function within the legacy system

The type of a parameter provides direct information in the way in which that parameter would take its responsibility. This type reflects the real-world state and determines the practical character. The type reflects the characteristics of the group that that parameter belongs to. Its type is related to the domain.

The length of a parameter contains a description of the size that parameter. This size corresponds to the limits of that parameter in terms of size in the real world. This constraint of size of a parameter forms a part of the domain knowledge.

2.2 From System Calls

System calls include the procedures and functions of legacy code. In general, the underlying code of these system calls is not shown; instead, these calls are directly used in the programs with corresponding

parameters. The program, after receiving the results of executing these system calls, represents these results in different ways.

System calls cope with the concrete problems within the domain. The names and parameters of the system calls reflect the functions that these calls and parameters perform within the legacy code and the problems that these calls try to address. Consequently, these system calls also form part of the domain knowledge.

2.3 From Operations

The operations of legacy code are performed in a prescribed sequence in order to accomplish a task. The organisation of these operations is based on the practical rules in the business domain. These rules are rules that need to be followed in order for the business tasks to be performed. Therefore, the operations of legacy code are the presentation of business rules in practice [14]. Consequently, these operations are the description of rules in that domain.

2.4 From Notes

The notes of the legacy system provide information that addresses some important issues regarding the legacy system. Notes can play different roles such as explaining the structures of the programs, providing the preconditions used during the execution of the program, expressing the user environments, or documenting the key conditions and functions of the legacy system.

The notes are dissimilar to source code in that they are not executed in the program. The users of those notes, by reading the domain knowledge contained within, will be able to comprehend the structures of the programs, the place in which the programs are applied, the preconditions and environments, and the effects this system has on other programs. Domain knowledge is implied in these notes.

3. Achieving Domain Knowledge from Other Sources

Legacy systems are designed to perform in their specific business area and address specific business needs. Therefore, a legacy system contains the description of the business data, the business operations, and the business process and the business rules. These descriptions reflect domain knowledge. A legacy system's operating system, the hardware at which the legacy system is performed, its documents, its performing records, the programming language from which it was produced, its input and output data, and its end users and maintainers involve aspects of the domain knowledge. The domain knowledge of legacy system is extracted from these components.

3.1. Operating System

The operating system underlies the legacy system and serves as the foundation software responsible for controlling and launching the installed applications and computer peripherals. It is the software that schedules tasks, allocates storage, handles the interface to peripheral hardware, and presents a default interface to the user when no application program is running [12].

Different operating systems have different characteristics. These different characteristics are suitable for specially designated application areas. Although it is almost impossible to list all the operating systems and demonstrate all its functions and applied domain, the operating systems present certain information about the application domain. Moreover, even though it is possible that some operating systems sustain the overlapping applications among them, that overlapping information reflects the specific domain at which the targeted legacy system is allocated.

The operating system contains some information of the adoption domain of legacy system. Certain domain knowledge is refined from the operating system of designated legacy system. If one legacy system is performed under the hold of the HP-BASIC, it is one industrial system for the real-time control, although it is not determined that whether targeted legacy system is a system controlling the weapon system in the fighter or a system dominating the defence system against the ballistic missiles, or the others.

Therefore, when reverse engineering legacy system into the UML, it is necessary to achieve the certain domain knowledge from the operating system of designated legacy system.

3.2. Data Environment

The data environment includes the databases and data stores of a system, the data triggers, and associated data interfaces of the system that manipulates this data. Because a database of a system is often tightly integrated with the program that manipulates it, the database, along with its associated entities such as data triggers, must be a part of any reengineering process.

Databases, because of their platform dependence and their tight integration with existing programs, must either be considered with their present physical configuration or properly reengineered into an equivalent model. An example, a legacy system may be tightly linked to its VSAM database of non-relational tables. In order to reengineer this present database, the present data elements and tables of the VSAM database are remodelled and mapped to a new, relational model database. Any old relationships among data must be carried forward to the new model. Once these data elements are remapped, any associated entities which manipulate them, whether data elements or program code, must be changed to accommodate this remapping.

Databases and data structures of legacy systems contain a wealth of information about a system's business rules, domain information, and functional requirements. When reengineering a database or data store, this information must be derived from the present system, by analysing the generation and use of each data elements in the system, and transferred over to the new data model. [1]

The structures of the data model, along with the relationships among its data elements, are an important part of a system's domain knowledge. The grouping of the data elements and the relationships among these

clusters indicate a logical design of data and a form of domain knowledge of the system.

The names of the data elements may also provide a clue to their function and meaning, which forms a part of the domain knowledge. Programmers may give meaningful names to their data elements using some logical standard. An example, a data element containing a person's social insurance number may be abbreviated to *SIN*. By analysing a data element's name and its functioning within the system, the meaning and purpose of the data element may be derived.

3.3. Programming Language

The programming language, in which the legacy system was written, is a set of rules, called a syntax, which the developers utilise to write computer applications. A programming language is an artificial language. Statements using the syntax of this programming language form the code of a legacy system. This code is written in an editor, read through a compiler, and then interpreted by the computer, which executes the program comprised of the programming statements [12].

One programming language is a formal language in which computer programs are written. The definition of a particular language consists of both the syntax (how the various symbols of the language may be combined) and the semantics (the meaning of the language constructs). Languages are classified as low level if their programming statements closely resemble machine code and are classified as high level if each language statement corresponds to many machine code instructions. However, this categorisation could also apply to a low-level language with extensive use of macros, in which case it would be debatable whether it still counted as a low-level language [10].

Different legacy systems are written in their specific programming languages. Each programming language has its own functionality and speciality. It is difficult to list all the programming languages and their functions and characteristics. Every legacy system has the nature of that programming language in which it was written. One legacy system written in COBOL has the operational character of simple computations on large amounts of data. COBOL is designed for commercial software systems, just like the recommendation of its name COBOL--COmmon Business Oriented Language. Java, because of its robustness and object-oriented qualities, enable it to support platform-independent Java "applets" (or independent pieces of code). Java's qualities make it well suited for Internet programming. Each programming language was designed for the special purpose and focuses on its specified problem areas, and has its specified application domain.

Consequently, the legacy system has the innate character of the specific programming language in which it is written. This specific programming language helps uncover the type of domain in which the legacy system operates, whether this domain is commercial or Internet based, and hence the programming language provides a form of domain knowledge.

3.4. Input and Output Data

Legacy systems commonly have input and output data. Legacy systems need input data to perform computations and comparisons and they then store the results of this manipulated data with the output data.

The usage of data within a particular legacy reflects both the type of domain that the application was designed to operate in and reflects a type of domain knowledge.

An example, one legacy system, written in COBOL and designed to deal with the customer services at a bank, probably has a large amount of records as the input data and one database as the output data, while one legacy system, written in HP-BASIC and designed to defend against ballistic missile attack, may have only six input parameters and no output data. The type of input/output data such as type, unit, and size, represent domain knowledge to a certain extent.

3.5. Executing Hardware

The hardware supporting the legacy system is the physical, tangible, material parts of a computer or other system. The term is used to distinguish these fixed parts of a system from the more changeable software or data components which executes, stores, or carries instructions or data. [10].

The hardware is also related to the domain. If one legacy system performs the data collection, its hardware has A/D, D/A, filter, and data storage. A large amount of RAM to support this high data usage is a requirement. If a legacy system is used in a bank to calculate a customer's interest earned, the hardware features of A/D and D/A are not necessary; however, hardware must possess high speed and precision characteristics.

3.6. Documents

The documents of legacy system contain the purpose of realising that legacy system, the technologies that were adopted, the architecture of legacy system, and the original design information.

The technologies of producing legacy system are the set of the solutions to the business problems, the mathematical methods, and the way to implement the design model. They include the approaches to coping with the specified business problems. Therefore, these documents also contain domain knowledge.

Architecture documentation is sufficiently abstract that it can be understood by new developers, yet detailed enough to serve as the description for construction. It has enough information so that it is regarded as a basis for analysis [3]. Architecture documentation is both prescriptive and descriptive. It prescribes what should be true by placing constraints on decisions that are about to be made, and it describes what is true by recounting decisions that already have been made. The constraints and decisions are the specifications of the business problems, which is allocated in the problem domain and represents parts of the domain knowledge. So the architecture documents, because they contain constraints and decisions of the architecture within them, represent domain knowledge [15].

3.7. Stakeholders

In the stakeholders, the end users are at a fundamentally important position. System users work with the legacy system. They record the preconditions of the legacy system and confirm the status of the system environment for reverse engineering legacy system. Business managers are also important because they publicly and privately support the system and use it directly or indirectly. Maintainers work with the legacy system using the old technologies and techniques of the system. They present the perspective from the maintenance point of view. Developers of other systems may communicate with the legacy system to interact with the legacy system somehow. It is important that some of stakeholders of legacy system work together to take part in project to achieve the accomplishment of reverse engineering the legacy system [6] [7].

System users should share business knowledge with the developing team and to make both pertinent and timely decisions for system scope and the achievement of the main tasks of legacy system. In fact, they must participate in the reverse engineering work, provide all the information that they know of the legacy system, invest enough time to share their business knowledge with the reverse engineering team, introduce the operational and support environment to the team, and present possible changes of the system environment and the system functions. The developing teams of other systems must exchange their information of their systems with the reverse engineering team to cooperate on the interfaces, the operational and support environments, the expression of the results, and the scopes of their systems and the legacy system with the reverse engineering team. It is important for the maintainers to report the records of legacy system and some changes. They are adept in the technologies and techniques of the legacy system because they maintained and enhanced the legacy system. They not only grasped the knowledge of the system scope, operational and support environment, architecture, and details performing the tasks, but also efficiently modified the bugs of the system. They must interact with the reverse engineering team to make the project of reverse engineering that legacy system successful [17].

The stakeholders of legacy system play an important role in the process of reverse engineering legacy system into the UML. It is necessary to acquire the domain knowledge from the stakeholders of legacy system [8].

4. Application of Domain Knowledge

During the process of reverse engineering legacy system into the UML, the domain knowledge is essential. The models that are extracted from the legacy system are loaders of domain knowledge. Legacy system has static and dynamic aspect. Domain knowledge is adopted in each of these aspects when modeling legacy system [9].

4.1 Static Aspect

When reverse engineering the concrete software system into the UML, the static information is shown as a class diagram, an object diagram, a component diagram, and a deployment diagram. The class diagram is regarded as the most common and the most important diagram used to represent the static model elements of the subject program, their contents and relationships. A class diagram is a graphical presentation of the static aspect that shows a collection of static model elements, such as classes, interfaces, types, as well as their contents and relationships [5].

During the process of reverse engineering static aspect of legacy system, the names of models, classes, objects, components, nodes, subsystems, and interfaces are domain-related. It is necessary to have problem-related names so as to make the models more readable and understandable at an abstract level. When modeling the static aspect of a legacy system, the names of attributes and operations of objects and classes form part of the domain knowledge of a system. The relationships among the classes in class diagrams are the representation of the real-world relations [16].

The application of domain knowledge is of great help in detecting any errors encountered while producing the class diagrams of legacy system. The practical information of that domain is utilized in the extraction of classes and class diagrams. The background knowledge on the problem domain helps to define the overall models of legacy system, concrete attributes and operations of classes, and their relationships.

4.2 Dynamic Aspect

Extracting information about the dynamic behaviour of the software is especially important when examining object-oriented software. Developers can not rely on understanding the dynamic behaviour of a system just by viewing its source code; they must be provided with UML diagrams depicting this dynamic behaviour. This is due to the dynamic nature of object-oriented programs: object creation, object deletion/garbage collection, and dynamic binding make it very difficult, and in many cases impossible, to understand the behaviour by just examining the source code.

In reverse engineering dynamic aspect of legacy system into the UML, abstractions are typically behavioural patterns and use cases that show interaction among high-level static components. The use case diagram is the kernel of the UML and regarded as the most fundamental to model the dynamic aspect of legacy system from the overall point of view [13] [18].

When modelling dynamic aspect of legacy system with UML diagrams, the application of domain knowledge is essential. The behavioural description of legacy system is the presentation of the operations of the solutions to the practical problems and refined in the UML. Every operation in the real world is represented in the problem domain. In order to deal with the business problem that the models of the problem domain present, the dynamic aspect of legacy system must obey the rules of that solution in that domain. In some cases, the legacy system is the representation of business rules. In order to understand the legacy system using reverse engineering, the business rules of that domain are adopted in the process of reverse engineering the legacy system.

5. A Case Study

The legacy Message Queue Program is a sample COBOL program designed to put messages in a message queue; this program provides an example of the use of MQPUT (licensed by IBM Inc). This program forms one part of legacy system. Source code of Message Queue Program is shown in Appendix A.

The Message Queue Program was written in COBOL. Because the COBOL programming language was designed to perform simple computations and comparisons on large amounts of data, the legacy systems written COBOL tend to be used in the bank system and commercial system [11] [19] [28].

The operating system of the Message Queue Program is IBM CICS/ESA Version 3.3. This operating system typically is used for banking and commercial applications that perform simple computation and comparison of a large amount of messages or data.

The input and output data of the Message Queue Program are the messages used by this program. These messages tend to be small in size. Messages are defined from the following parameter definition:

01 BUFFER PIC X(60)
01 BUFFER-LENGTH PIC S9(9) BINARY
01 TARGET-QUEUE PIC X(48)

The hardware platform of the Message Queue Program is IBM 360. Like the IBM CICS/ESA operating system, this platform is largely used to support banking and commercial systems.

No documents of the Message Queue Program are available.

The stakeholders of legacy code Message Queue Program are defined as the company manager and the employee.

Descriptive parameters names were chosen to represent variables. For example, the word "MQ" means "message queue".

The source code represents the step-by-step operations in the execution of a task. Source code provides a description of how execution is to be completed. Source code also constitutes rules on how to handle the message queue.

The calls "MQCONN", "MQOPEN", "MQCLOSE", "MQDISC" and "MQPUT" within the Message Queue program are system calls with some parameters. These calls perform operations such as connecting the queue manager, opening the queue, close the queue, disconnecting message queue, and putting the message into the queue. These calls are related to the message queue.

The static aspect of legacy code Message Queue Program is modeled as the class diagram shown in Appendix B.

The dynamic aspect of legacy code Message Queue Program is modeled as the use case diagram shown in Appendix C.

6. Conclusions

The process of reverse engineering a legacy system into a UML representation tends to be convoluted. The most important activity of this process is acquiring the domain knowledge of legacy system.

The research in this paper focuses on acquiring domain knowledge during this reverse engineering legacy system into the UML. The domain knowledge of a legacy system is derived from this system's data environment, input and output data, documents, stakeholders, specific programming language(s) used, source code, and underlying operating system. This knowledge is applied when modelling the static and dynamic aspects of the legacy system. Because class diagram and use case diagram are the most important in reverse engineering legacy system to model its static and dynamic aspects respectively, the domain knowledge is adopted in the realisation of class diagrams and use case diagrams of legacy system.

Reference

[1] Aiken, P, Alice, M, Russ, R, "DoD Legacy Systems: Reverse Engineering Data Requirements", Vol 37, No 5, Communications of the ACM, May 1994.

[2] Alhir, S S, UML in a Nutshell O'Reilly, Sebastapol, CA, USA, 1998.

[3] Bachmann, F, Bass, L, Clements, P, Garlan, D, Ivers, J, Little, R and Nord, R, "Documenting Software Architecture: Organisation of Documentation Package", http://www.sei.cmu.edu/pub/documents/01.reports/pdf/01tn010.pdf, CMU/SEI-2001-TN-010, 2001.

[4] Bangera, R and Rand, B, "Reverse Engineering and UML: A Case Study of AuctionBot", http://www-personal.engin.umich.edu/~wrand/eecs581/UMLFinal.doc, 2000.

[5] Bennett, S., McRobb, S., and Farmer, R, Object-Oriented Systems Analysis and Design using UML, The McGraw-Hill Companies, 1999.

[6] Booch, G, Rumbaugh, J, and Jacobson, I, The Unified Modeling Language User Guide, Reading, MA: Addison-Wesley-Longman, Inc, 1999.

[7] Comella-Dorda, S et al, "Incremental Modernisation for Legacy Systems", CMU/SEI-2001-TN-006, http://www.sei.cmu.edu/publications/documents/01.reports/01tn006.html, 2001.

[8] Frakes, W, "Advanced Topics in Software Engineering Domain Engineering and systematic Reuse", Virginia Polytechnic Institute and State University, 1998.

[9] Freitas, F G et al, "Reusing Domains for the Construction of Reverse Engineering Tools", Sixth Working Conference on Reverse Engineering, Atlanta, Georgia, 1999.

[10] Howe, D, "FOLDOC: Free On-Line Dictionary of Computing", http://foldoc.doc.ic.ac.uk/foldoc/index.html, 2002.

[11] Hutty, R and Spence, M, "Mastering COBOL Programming", Macmillan Press Ltd, 1997.

[12] Jansen, E, "The Internet Dictionary", Netlingo Inc, http://www.netlingo.com/inframes.cfm, 2002.

[13] Larman, C, Applying UML and Patterns an Introduction to Object-Oriented Analysis and Design, Reading, MA: Prentice-Hall, Inc, 1998.

[14] Li, Y; Yang, H and Chu, W, "A Concept Oriented Brief Revision Approach to Domain Knowledge Recovery from Source Code", Journal of Software Maintenance: Research and Practice, 13(1), 2001.

[15] Osterbye, K et al, "Documentation of Object-Oriented Systems and Frameworks", COT/2-42-V2.4. Centre for Object Technology, http://www.cit.dk/COT/reports/reports/Case2/42/cot-2-42.pdf, 2000.

[16] Page-Jones, M, "Fundamentals of Object-Oriented Design in UML", Dorset House Publishing, 2000.

[17] Samuelson, P and Scotchmer, S, "The Law And Economics Of Reverse Engineering", University of California at Berkeley, http://www.sims.berkeley.edu/~pam/papers/l&e%20reveng5.pdf, 2001.

[18] Seacord, R C, Comella-Dorda S, Lewis, G, Place, P and Plakosh, D, "Legacy System Modernisation Strategies", http://www.sei.cmu.edu/pub/documents/01.reports/pdf/01tr025.pdf, 2001.

[19] Thompson, John B, Structured Programming with COBOL and JSP, John Barrie Thompsom and Chartwell-Bratt Ltd, 1989.

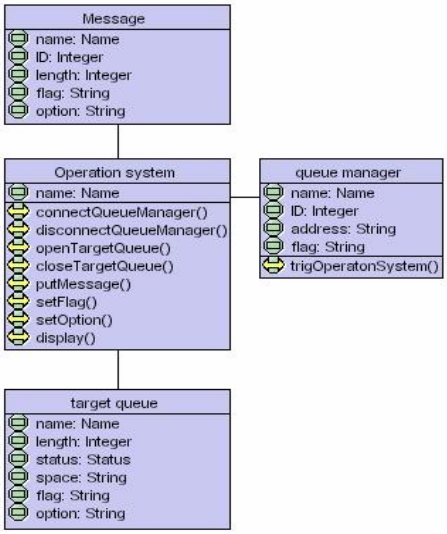
[20] Yang, Hongji, "The Supporting Environment for A Reverse Engineering System --- The Maintainer's Assistant", IEEE Conference on Software Maintenance (ICSM 91) Sorrento, Italy, 1991.

[21] Yang, H; Chu, W and Sun, Y, "Practical System of COBOL Program Reuse for Reengineering", 8th International Workshop on Software Technology and Engineering Practice (STEP'97), pp. 45-57, London, July, 1997.

Appendix A
Legacy Code Message Queue Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. 'AMQ0PUT0'.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 MY-MQ-CONSTANTS.
  COPY CMQV.
01 OBJECT-DESCRIPTOR.
  COPY CMQODV.
01 MESSAGE-DESCRIPTOR.
  COPY CMQMDV.
01 PMOPTIONS.
  COPY CMQPMOV.
01 QM-NAME PIC X(48) VALUE SPACES.
01 HCONN      PIC S9(9) BINARY.
01 Q-HANDLE PIC S9(9) BINARY.
01 OPTIONS   PIC S9(9) BINARY.
01 COMPLETION-CODE PIC S9(9) BINARY.
01 OPEN-CODE      PIC S9(9) BINARY.
01 CON-REASON   PIC S9(9) BINARY.
01 REASON          PIC S9(9) BINARY.
01 BUFFER          PIC X(60).
01 BUFFER-LENGTH PIC S9(9) BINARY.
01 TARGET-QUEUE   PIC X(48).
PROCEDURE DIVISION.
P0.
   DISPLAY 'AMQ0PUT0 start'.
   DISPLAY 'Please enter the name of the target queue '
   ACCEPT TARGET-QUEUE FROM CONSOLE.
   CALL 'MQCONN'
    USING QM-NAME, HCONN,
    COMPLETION-CODE, CON-REASON.
   IF COMPLETION-CODE IS EQUAL TO MQCC-FAILED
     DISPLAY 'MQCONN ended with reason code ' CON-REASON
     MOVE CON-REASON TO RETURN-CODE
     GOBACK
     END-IF.
   DISPLAY 'target queue is ' TARGET-QUEUE.
OPENS.
   MOVE TARGET-QUEUE TO MQOD-OBJECTNAME.
   ADD MQOO-OUTPUT MQOO-FAIL-IF-QUIESCING
         GIVING OPTIONS.
   CALL 'MQOPEN'
    USING HCONN, OBJECT-DESCRIPTOR,
    OPTIONS, Q-HANDLE,
    OPEN-CODE, REASON.
   IF REASON IS NOT EQUAL TO MQRC-NONE
     DISPLAY 'MQOPEN ended with reason code ' REASON
     END-IF.
   IF OPEN-CODE IS EQUAL TO MQCC-FAILED
     DISPLAY 'unable to open target queue for output'
     MOVE REASON TO RETURN-CODE
     GOBACK
     END-IF.
PUTS.
   DISPLAY 'Please enter the message(s) '
   MOVE OPEN-CODE TO COMPLETION-CODE.
   PERFORM PUTR WITH TEST BEFORE
     UNTIL COMPLETION-CODE IS EQUAL TO MQCC-FAILED.
CLOSES.
   MOVE MQCO-NONE TO OPTIONS.
   CALL 'MQCLOSE'
    USING HCONN, Q-HANDLE, OPTIONS,
    COMPLETION-CODE, REASON.
   IF REASON IS NOT EQUAL TO MQRC-NONE
     DISPLAY 'MQCLOSE ended with reason code ' REASON
     END-IF.
DISCS.
   IF CON-REASON IS NOT EQUAL TO MQRC-ALREADY-CONNECTED
     CALL 'MQDISC'
      USING HCONN, COMPLETION-CODE, REASON
     IF REASON IS NOT EQUAL TO MQRC-NONE
      DISPLAY 'MQDISC ended with reason code ' REASON
     END-IF
   END-IF.
OVER.
   DISPLAY 'AMQ0PUT0 end'.
   MOVE ZERO TO RETURN-CODE.
   GOBACK.
PUTR.
   MOVE SPACES TO BUFFER.
   ACCEPT BUFFER.
   IF BUFFER IS NOT EQUAL TO SPACES
    PERFORM PUTIT
   ELSE
    MOVE MQCC-FAILED TO COMPLETION-CODE.
PUTIT.
   MOVE 60 to BUFFER-LENGTH.
   CALL 'MQPUT'
    USING HCONN, Q-HANDLE,
    MESSAGE-DESCRIPTOR, PMOPTIONS,
    BUFFER-LENGTH, BUFFER,
    COMPLETION-CODE, REASON.
   IF REASON IS NOT EQUAL TO MQRC-NONE
     DISPLAY 'MQPUT ended with reason code ' REASON
     END-IF.
```

Appendix B
Class Diagram of Message Queue Program



Appendix C
Use Case Diagram of Message Queue Program