Chapter X

Visualising COBOL Legacy Systems with UML: An Experimental Report

Steve McRobb, De Montfort University, UK
Richard Millham, De Montfort University, UK
Jianjun Pu, De Montfort University, UK
Hongji Yang, De Montfort University, UK

Abstract

This chapter presents a report of an experimental approach that uses WSL as an intermediate language for the visualisation of COBOL legacy systems in UML. Key UML techniques are identified that can be used for visualisation. Many cases were studied, and one is presented in detail. The report concludes by demonstrating how this approach can be used to build a software tool that automates the visualisation task. Furthermore, understanding a system is of critical importance to a developer who must be able to understand the business processes being modeled by the system along with the system’s functionality, structure, events, and interactions with external entities. Such an understanding is of even more importance in reverse engineering. Although developers have the advantage of having the source code available, system documentation is often missing or incomplete, and the original users, whose requirements were used to design the system, are often long gone.
Introduction

A developer requires a model of a system not only in order to understand the business processes being modeled but also the structure and dynamics of the system. Visualisation of the system model is often necessary in order to clearly depict the complex relationships among model elements of that system.

This chapter focuses on the visualisation of a system from a reengineering, rather than a forward engineering, point of view as an attempt is made to extract documentation from the system code into the visual notation of UML. In many legacy systems, the system code is the only surviving artefact and consequently, any reengineering efforts must focus on this artefact. Visualisation, whether through UML or some other graphical notation, is important because it better enables our brains to make the connections between the software system and the ideas represented within this system; this connection is much more difficult if the documentation is based purely on textual form. This chapter also investigates whether it is possible to visualise a system and which of the nine possible UML diagrams are needed to represent this visualisation. A detailed outline of the process of deriving information from an analysis of the selected legacy system, a batch-oriented system, is given, along with the rules to convert this information to a UML model of this system. A brief overview of the methods, along with their inherent difficulty that are used to extract UML diagrams from a legacy system is provided. Given the selected system, a batch-oriented legacy system, and relying on the only surviving form of documentation, it was found that it was not possible to extract statecharts or use cases from this system. Use cases, which model external actors and business processes of a system, cannot satisfactorily be extracted from system code alone but require intensive user intervention and guidance in order to identify the external actors and their roles. Statecharts, which model external events and the system’s response to them, cannot be extracted satisfactorily from a batch-oriented legacy system for two reasons. One reason is that batch-oriented systems have very few external events; usually, their only external event is the arrival of input. The other reason is that an event-response trace of the system, which can be used to model statecharts of this system, cannot be obtained through analysis of system code alone; this event-response trace requires the use of a run-time environment along with a full set of possible external events that this system might encounter. Finally, a tool, TAGDUR, is introduced which has automated some of these analysis processes and which models some aspects of this system in UML.

Why Visualisation of the Legacy System is Necessary for Reverse Engineering

Program understanding can be defined as the process of developing an accurate mental model of a software system’s intended architecture, purpose, and behaviour. This model is developed through the use of pattern matching, visualisation, and knowledge-based techniques. The field of program understanding involves fundamental issues of code representation, structural system representation, data and control flow, quality and