To *my wife*, Jianmei Gao,

*my son*, Zhichen Pu and

*my parents.*
Declaration

I declare that the work described in this thesis was originally carried out by me during the period of registration for the degree of Doctor of Philosophy at De Montfort University, U.K., from December 2002 to February 2008. It is submitted for the degree of Doctor of philosophy at De Montfort University. Apart from the degree that this thesis is currently applying for, no other academic degree or award was applied for by me based on this work.
Acknowledgements

For many years I had been dreaming about receiving a PhD. I would like to thank many people who helped me in achieving this dream in different ways when I undertook the work of this thesis.

I wish that there were words to express my sincere appreciation to thank Professor Hongji Yang for agreeing to be my supervisor, for his invaluable advice and encouragement and for his skilful supervision and administration throughout this research project. I would like to thank him for teaching me, caring for me, and for motivating me all these years. He is the great mentor and role models. I admire him for his insight in so many issues and for his endless enthusiasm for research.

I also wish to thank Dr. Antonio Cau for agreeing to be my supervisor and for his valuable suggestions and inspiring me.

I would also like to thank the Research Office at De Montfort University for their outstanding management.

I would also like to thank Mr. Zhuopeng Zhang, Mr. Jian Kang, Dr. Feng Chen, Dr. Bin Qiao, Mr. Yang Xu and Mr. Hong Zhou for their useful advice in research techniques, support in the working environment and encouragement in completing this thesis.

I wish to express thanks to my wife, Jianmei Gao, my son Zhichen Pu, my parents and parents in law for all their loves, encouragements, patience and supports over the years. This thesis is dedicated to them.

Finally, I would like to thank all of the colleagues in Software Technology Research Laboratory at De Montfort University for their valuable suggestions and discussions, for their encouragement and support, for building great software packages. I am indebted to all of them.
Abstract

With the high demand for renovation of legacy systems, their evolution is becoming an urgent need. Although some approaches have been introduced to evolving legacy systems, they are not sufficient for understanding legacy code.

In this thesis, development/environment-specific models of domain-specific legacy systems are acquired, based on their characteristics and operations. The development/environment-specific model of COBOL legacy systems is based on the characteristics and operations of COBOL, and is a procedure-based model comprising a graph that describes the calling and being-called relationships of those procedures in COBOL legacy systems. It has four types: linear, branch, joint, and synthetic procedure-based models. The link-based model of HTML legacy systems uses a graph that describes the importing or imported relationships of webs in a legacy system. It has three types: sequential, cyclical, and compositive link-based models. The development/environment-specific model of the SQL legacy system comprises association, generation and composition database-based models based on the basic operations of SQL and the two main relationships of generation and association between the databases in an SQL legacy system.

The structural stage of UML extraction in this thesis contains class realisation. The classification of classes from COBOL legacy system is two, which are procedure class and variable class. Every procedure in COBOL legacy system is defined as one procedure class. Variable class is based on the program slicing techniques with two stages of pseudo class and real class extraction from COBOL legacy system. The variable of the sliced criterion is defined as the class name, and the variables contained in its slicing criterion are defined as the attributes of that variable class. Because the behavioural analysis of domain-specific legacy systems is behind the analysis of structural analysis, the operations in variabl class are not described. The classification of classes of HTML legacy system is based on the web pages and their blocks. The classification of SQL legacy system is two, which is procedure class and database class. Selected UML diagrams are used to describe the static aspect of domain-specific legacy systems.
Abstract

The behavioural stage of UML extraction in this thesis focuses on the operations and activities of domain-specific legacy systems. When understanding the operations and activities of domain-specific legacy code, their preconditions and post-conditions must be presented from the source code. Then those operations and activities are ordered according to the time and sequence they are executed by. At last, the operation and activity arrays are presented. Selected UML diagrams describing the dynamic aspect of domain-specific legacy systems are realised based on those operation and activity arrays.

The major contribution of this thesis is the presentation of development/environment-specific models of domain-specific legacy systems and an approach towards software evolution of domain-specific legacy systems using UML diagrams.
Table of Contents

Declaration ................................................................. ii
Acknowledgements ........................................................................ iii
Abstract ............................................................................................... iv
Table of Contents ................................................................................ vi
List of Figures ......................................................................................... xii
List of Tables ......................................................................................... xx

Chapter 1 Introduction ........................................................................... 1
  1.1 Proposed Research ................................................................. 1
  1.2 Overview of Problems .................................................................... 3
  1.3 Scope of Thesis and Original Contributions ..................................... 5
  1.4 Criteria for Success ......................................................................... 6
  1.5 Thesis Structure ............................................................................. 7

Chapter 2 Background ............................................................................ 9
  2.1 Introduction .................................................................................... 9
  2.2 Legacy Systems ............................................................................. 9
    2.2.1 Definition of Legacy Systems ..................................................... 9
    2.2.2 Two Aspects of Legacy Systems ............................................... 10
    2.2.3 Three Types of Legacy Systems to Be Discussed .................... 10
  2.3 Software Evolution .......................................................................... 11
    2.3.1 Software Engineering .............................................................. 11
    2.3.2 Software Reengineering .......................................................... 12
    2.3.3 Reverse Engineering ................................................................ 15
    2.3.4 Software Evolution ................................................................. 18
    2.3.5 Maintainer's Assistant .............................................................. 20
    2.3.6 Refactoring Browser ............................................................... 21
    2.3.7 Cognitive Methods ................................................................. 22
  2.4 UML ............................................................................................ 22
    2.4.1 Definition of UML ................................................................. 22
    2.4.2 History of UML ..................................................................... 23
    2.4.3 Views of UML ..................................................................... 24
# Table of Contents

2.4.4 Properties of UML ................................................................. 24  
2.4.5 UML Diagrams ................................................................. 25  
2.4.6 Importance of Modelling Legacy Systems with UML ................. 26  
2.5 Program Slicing ................................................................. 27  
2.5.1 Introduction ........................................................................ 27  
2.5.2 Static and Dynamic Slicing ............................................. 28  
2.5.3 Applications of Program Slicing ...................................... 28  
2.6 Confining Analysis of Legacy Systems using UML ......................... 30  
2.7 Model-Driven Architecture .................................................... 31  
2.7.1 Model .............................................................................. 31  
2.7.2 Model-Driven Engineering ............................................... 31  
2.7.3 Model-Driven Architecture ............................................. 32  
2.8 Summary .............................................................................. 32  

Chapter 3 Proposed Approach ......................................................... 34  
3.1 Introduction ............................................................................. 34  
3.2 Comparison of Proposed Approach and Traditional Studies ............. 35  
3.2.1 Traditional Studies .......................................................... 35  
3.2.2 Proposed Approach ....................................................... 38  
3.3 Parsing Legacy Systems ....................................................... 42  
3.3.1 Parsing Original Code ..................................................... 42  
3.3.2 Procedure-Based Models of COBOL Legacy Systems .............. 43  
3.3.3 Link-Based Models of HTML Legacy Systems ..................... 43  
3.3.4 Database-Based Models of SQL Legacy Systems .................. 44  
3.4 Suitability of Program Slicing Techniques .................................... 45  
3.4.1 Slicing COBOL Legacy Systems ...................................... 45  
3.4.2 No Need to Slice HTML Legacy Systems ......................... 46  
3.4.3 Slicing SQL Legacy Systems ......................................... 46  
3.5 Unnecessary to Use All UML Diagrams when Modelling Legacy Systems .............................................. 46  
3.5.1 Completeness of UML ..................................................... 46  
3.5.2 Differences between Modelling Tasks ................................ 47  
3.6 Extracting UML Diagrams from COBOL Legacy Systems ............... 47  
3.6.1 UML Class Diagram ....................................................... 47  
3.6.2 UML Composite Structure Diagram .................................. 48  
3.6.3 UML Sequence Diagram .............................................. 48  
3.6.4 UML Interaction Overview Diagram .................................. 49  
3.7 Extracting UML Diagrams from HTML Legacy Systems ................ 49  
3.7.1 UML Class Diagram ....................................................... 49  
3.7.2 UML Composite Structure Diagram .................................. 50  
3.7.3 UML Component Diagram ............................................. 50  
3.7.4 UML Deployment Diagram ............................................. 50
Table of Contents

3.8 Extracting UML Diagrams from SQL Legacy Systems .............................................................. 50
  3.8.1 UML Class Diagram .......................................................................................................... 50
  3.8.2 UML Composite Structure Diagram .................................................................................. 51
  3.8.3 UML Activity Diagram ....................................................................................................... 51
3.9 Summary ...................................................................................................................................... 52

Chapter 4 COBOL Development/Environment-Specific Models .................................................. 53
  4.1 Introduction ................................................................................................................................. 53
  4.2 Rationale of Software Evolution of COBOL Legacy Systems .................................................... 53
  4.3 COBOL Legacy Systems ............................................................................................................. 54
    4.3.1 COBOL ............................................................................................................................. 54
    4.3.2 A Brief History of COBOL ................................................................................................. 54
    4.3.3 Characteristics of COBOL ................................................................................................. 55
  4.4 Parsing COBOL Code ................................................................................................................. 59
    4.4.1 Grouping COBOL Code Operations .................................................................................. 59
    4.4.2 Parsing COBOL Code ........................................................................................................ 61
  4.5 Procedure-Based Model ............................................................................................................... 62
    4.5.1 Structure ............................................................................................................................. 62
    4.5.2 Classification of A Procedure-Based Model ...................................................................... 66
  4.6 Summary ...................................................................................................................................... 70

Chapter 5 HTML Development/Environment-Specific Models ..................................................... 71
  5.1 Introduction ................................................................................................................................. 71
  5.2 Rationale of Software Evolution of HTML Legacy Systems ...................................................... 71
  5.3 HTML Legacy Systems ............................................................................................................... 72
    5.3.1 HTML ................................................................................................................................. 72
    5.3.2 A Brief History of HTML .................................................................................................... 72
    5.3.3 Characteristics of HTML ................................................................................................... 73
  5.4 Parsing HTML Code ................................................................................................................... 74
    5.4.1 Grouping HTML Code Operations .................................................................................... 74
    5.4.2 Parsing HTML Code .......................................................................................................... 76
  5.5 Link-Based Models ...................................................................................................................... 77
    5.5.1 Structure ............................................................................................................................. 77
    5.5.2 Classification of Link-Based Models .................................................................................. 80
  5.6 Summary ...................................................................................................................................... 85

Chapter 6 SQL Development/Environment-Specific Models .......................................................... 86
  6.1 Introduction ................................................................................................................................. 86
  6.2 Rationale of Software Evolution of SQL Legacy Systems .......................................................... 86
  6.3 SQL Legacy Systems ................................................................................................................... 87
    6.3.1 SQL ................................................................................................................................. 87
    6.3.2 Characteristics of SQL ....................................................................................................... 87
<table>
<thead>
<tr>
<th>Chapter 6</th>
<th>Extracting UML Diagrams from COBOL Legacy Systems</th>
<th>97</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>97</td>
</tr>
<tr>
<td>6.2</td>
<td>Using Four UML Diagrams to Model COBOL Legacy Systems</td>
<td>97</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Static Modelling</td>
<td>97</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Dynamic Modelling</td>
<td>99</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Modelling with Four UML Diagrams</td>
<td>101</td>
</tr>
<tr>
<td>6.3</td>
<td>Modelling COBOL Legacy Systems with UML</td>
<td>101</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Class Diagrams</td>
<td>101</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Composite Structure Diagrams</td>
<td>118</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Sequence Diagrams</td>
<td>121</td>
</tr>
<tr>
<td>6.3.4</td>
<td>Interaction Overview Diagrams</td>
<td>133</td>
</tr>
<tr>
<td>6.3.5</td>
<td>Application of COBOL Rules</td>
<td>142</td>
</tr>
<tr>
<td>6.4</td>
<td>Summary</td>
<td>144</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 7</th>
<th>Extracting UML Diagrams from HTML Legacy Systems</th>
<th>146</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>146</td>
</tr>
<tr>
<td>7.2</td>
<td>Using Four UML Diagrams to Model HTML Legacy Systems</td>
<td>146</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Not Having Dynamic Modelling</td>
<td>146</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Static Modelling</td>
<td>147</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Using Four UML Diagrams to Model HTML Legacy Systems</td>
<td>148</td>
</tr>
<tr>
<td>7.3</td>
<td>Not Needing to Slice HTML Legacy Systems</td>
<td>148</td>
</tr>
<tr>
<td>7.4</td>
<td>Modelling HTML Legacy Systems</td>
<td>148</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Class Diagrams</td>
<td>148</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Composite Structure Diagrams</td>
<td>160</td>
</tr>
<tr>
<td>7.4.3</td>
<td>Component Diagram</td>
<td>162</td>
</tr>
<tr>
<td>7.4.4</td>
<td>Deployment Diagram</td>
<td>163</td>
</tr>
<tr>
<td>7.4.5</td>
<td>Application of HTML Rules</td>
<td>164</td>
</tr>
<tr>
<td>7.5</td>
<td>Summary</td>
<td>166</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 8</th>
<th>Extracting UML Diagrams from SQL Legacy Systems</th>
<th>167</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Introduction</td>
<td>167</td>
</tr>
<tr>
<td>8.2</td>
<td>Using Three UML Diagrams to Model SQL Legacy Systems</td>
<td>167</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Static Modelling</td>
<td>167</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Dynamic Modelling</td>
<td>168</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 9</th>
<th>Extracting UML Diagrams from SQL Legacy Systems</th>
<th>167</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>Introduction</td>
<td>167</td>
</tr>
<tr>
<td>9.2</td>
<td>Using Three UML Diagrams to Model SQL Legacy Systems</td>
<td>167</td>
</tr>
<tr>
<td>9.2.1</td>
<td>Static Modelling</td>
<td>167</td>
</tr>
<tr>
<td>9.2.2</td>
<td>Dynamic Modelling</td>
<td>168</td>
</tr>
</tbody>
</table>
Table of Contents

9.2.3 Modelling with Three UML Diagrams ......................................................... 170
9.3 Modelling SQL Legacy Systems ................................................................. 170
  9.3.1 Class Diagrams ......................................................................................... 170
  9.3.2 Composite Structure Diagrams ................................................................. 176
  9.3.3 Activity Diagrams ....................................................................................... 177
  9.3.4 Application of SQL Rules .......................................................................... 183
9.4 Summary ....................................................................................................... 185

Chapter 10 Tool and Experiments ........................................................................ 186
  10.1 Introduction ................................................................................................. 186
  10.2 Tool Design ................................................................................................ 186
    10.2.1 One Unified System—SEASAT ............................................................... 186
    10.2.2 Detail Design Description ....................................................................... 186
  10.3 Case Study of COBOL Legacy System ....................................................... 188
    10.3.1 COBOL Legacy Code—Manager Application ......................................... 188
    10.3.2 Parsing COBOL Legacy Code ................................................................. 189
    10.3.3 Slicing Programs ..................................................................................... 193
    10.3.4 UML Class Diagrams ............................................................................. 194
    10.3.5 UML Composite Structure Diagrams .................................................... 203
    10.3.6 UML Sequence Diagrams ...................................................................... 204
    10.3.7 UML Interaction Overview Diagrams .................................................... 212
  10.4 Case Study of HTML Legacy System ......................................................... 212
    10.4.1 HTML Legacy System ........................................................................... 212
    10.4.2 Parsing HTML Legacy Code ................................................................. 212
    10.4.3 UML Class Diagrams ............................................................................. 214
    10.4.4 UML Composite Structure Diagrams .................................................... 223
    10.4.5 UML Component Diagram .................................................................... 227
    10.4.6 UML Deployment Diagram .................................................................... 228
  10.5 Case Study of SQL Legacy System ............................................................. 228
    10.5.1 SQL Legacy System ............................................................................... 228
    10.5.2 Parsing SQL Legacy Code ..................................................................... 229
    10.5.3 Slicing SQL Source Code ...................................................................... 229
    10.5.4 UML Class Diagrams ............................................................................. 229
    10.5.5 UML Composite Structure Diagrams .................................................... 230
    10.5.6 UML Activity Diagrams ........................................................................ 231
  10.6 Summary ..................................................................................................... 233

Chapter 11 Discussion and Conclusions ............................................................... 235
  11.1 Comparison and Evaluation ...................................................................... 235
  11.2 Summary ..................................................................................................... 236
  11.3 Significance of Contributions and Evaluation ............................................. 237
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.4 Revisiting Criteria for Success</td>
<td>238</td>
</tr>
<tr>
<td>11.5 Conclusion</td>
<td>248</td>
</tr>
<tr>
<td>11.6 Future Work</td>
<td>249</td>
</tr>
<tr>
<td>11.6.1 Limitations</td>
<td>249</td>
</tr>
<tr>
<td>11.6.2 Directions for Future Work</td>
<td>250</td>
</tr>
<tr>
<td>References</td>
<td>251</td>
</tr>
<tr>
<td>Appendix A: COBOL Legacy System</td>
<td>263</td>
</tr>
<tr>
<td>Appendix B: HTML Legacy System</td>
<td>275</td>
</tr>
<tr>
<td>Appendix C: SQL Legacy System</td>
<td>286</td>
</tr>
<tr>
<td>Appendix D: List of Publications by Author</td>
<td>289</td>
</tr>
</tbody>
</table>
List of Figures

Figure 3.1: Comparison of Proposed Approach and Traditional Studies ................................................. 37

Figure 4.1: Process of Parsing COBOL Code ........................................................................................... 61

Figure 4.2 One to One Relationship .......................................................................................................... 65

Figure 4.3 One to Many Relationship ........................................................................................................ 65

Figure 4.4 Many to One Relationship ........................................................................................................ 65

Figure 4.5 Many to Many Relationship ..................................................................................................... 66

Figure 4.6: Linear Procedure-Based Model .............................................................................................. 67

Figure 4.7: Branch Procedure-Based Model ............................................................................................. 68

Figure 4.8: Joint Procedure-Based Model ................................................................................................. 69

Figure 4.9: Synthetic Procedure-Based Model .......................................................................................... 69

Figure 5.1: Process of Parsing HTML Code ................................................................................................ 76

Figure 5.2: Link-Based Model ................................................................................................................... 77

Figure 5.3: A Graphical Example of a Web Layer ..................................................................................... 78

Figure 5.4: First Example of an Ordinal Relationship .............................................................................. 79

Figure 5.5: Second Example of an Ordinal Relationship ......................................................................... 79

Figure 5.6: Third Example of an Ordinal Relationship ............................................................................ 79

Figure 5.7: Fourth Example of an Ordinal Relationship .......................................................................... 80

Figure 5.8: An Example of a Loop Relationship ...................................................................................... 80

Figure 5.9: An Example of a Sequential Link-Based Model ....................................................................... 81

Figure 5.10: Second Example of a Sequential Link-Based Model ............................................................. 81

Figure 5.11: Third Example of a Sequential Link-Based Model .............................................................. 81
List of Figures

Figure 5.12: Fourth Example of a Sequential Link-Based Model ............................................................. 82

Figure 5.13: An Example of a Cyclical Link-Based Model ........................................................................... 83

Figure 5.14: Another Example of a Cyclical Link-Based Model ..................................................................... 83

Figure 5.15: An Example of a Compositive Link-Based Model ....................................................................... 83

Figure 5.16: Second Example of a Compositive Link-Based Model .............................................................. 84

Figure 5.17: Third Example of a Compositive Link-Based Model .................................................................... 84

Figure 6.1: Process of Parsing SQL Code ................................................................................................ 91

Figure 6.2: Generation Relationship of a Database-Based Model .................................................................... 93

Figure 6.3: An Example of an Association Relationship ............................................................................. 93

Figure 6.4: Another Example of an Association Relationship ........................................................................ 93

Figure 6.5: Association Database-Based Model .......................................................................................... 93

Figure 6.6: Generation Database-Based Model .......................................................................................... 94

Figure 6.7: An Example of a Composition Database-Based Model ............................................................. 95

Figure 6.8: A Second Example of a Composition Database-Based Model .................................................. 95

Figure 6.9: A Third Example of a Composition Database-Based Model ....................................................... 96

Figure 6.10: A Fourth Example of a Composition Database-Based Model .................................................. 96

Figure 7.1: An Example of A Slice Criterion (13, {money}) ......................................................................... 105

Figure 7.2: Control Flow Graph of Example Program .............................................................................. 105

Figure 7.3: Definition Examples of an Original Program ........................................................................... 106

Figure 7.4: Computation of $RIN^0 (n)$ of Example Program ................................................................... 106

Figure 7.5: Computation of $S^0$ and $S^1 (n)$ of Example Program ............................................................ 106

Figure 7.6: The First Pseudo Class $V_1$ .................................................................................................... 110

Figure 7.7: The $f$th Pseudo Class $V_f$ of the $f$th Layer Procedure ............................................................ 111

Figure 7.8: Part Code Containing a Systematic Call .............................................................................. 113
List of Figures

Figure 7.9: Example of Systematic Call Class Presentation ................................................................... 113
Figure 7.10: Part Code Containing a Systematic Call ............................................................................ 113
Figure 7.11: Part of Code Example ......................................................................................................... 114
Figure 7.12: Example of Systematic Manager Class ............................................................................... 114
Figure 7.13: Leaf Class ........................................................................................................................... 115
Figure 7.14: Node Class .......................................................................................................................... 115
Figure 7.15: Class Diagram of a Linear Procedure-Based Model .......................................................... 116
Figure 7.16: Class Diagram of a Branch Procedure-Based Model ........................................................ 117
Figure 7.17: Class Diagram of a Joint Procedure-Based Model ............................................................ 117
Figure 7.18: Class Diagram of a Synthetic Procedure-Based Model ..................................................... 117
Figure 7.19: An Example of a Composite Structure Diagram ................................................................. 119
Figure 7.20: Composite Structure Diagram of Class V_1........................................................................ 119
Figure 7.21: A Leaf COBOL Program DateEdit ..................................................................................... 120
Figure 7.22: Composite Structure Diagram of DateEdit ......................................................................... 121
Figure 7.23: A Sequence Diagram for Printing Header .......................................................................... 125
Figure 7.24: A Program Example ............................................................................................................ 128
Figure 7.25: A Sequence Diagram of Printing Rest Header ................................................................... 128
Figure 7.26: An Example of a Sequence Diagram ................................................................................... 131
Figure 7.27: Sequence Diagram of a Linear Procedure-Based Model ................................................... 132
Figure 7.28: Sequence Diagram of a Branch Procedure-Based Model .................................................. 132
Figure 7.29: Sequence Diagram of a Joint Procedure-Based Model ...................................................... 132
Figure 7.30: Sequence Diagram of a Synthetic Procedure-Based Model ............................................... 133
Figure 7.31: An Example of a Interaction Overview Diagram based on Figure 7.1 ............................... 134
Figure 7.32: Fork Node IF...ELSE...based on Figure 7.2 ..................................................................... 135
List of Figures

Figure 7.33: Fork Node IF ................................................................. 136
Figure 7.34: Control Flow Graph of Fork Node IF ......................... 136
Figure 7.35: Fork Node IF...ELSE .................................................. 136
Figure 7.36: Control Flow Graph of Fork Node IF...ELSE .............. 137
Figure 7.37: Control Flow Graph of Fork Node EVALUATE .......... 137
Figure 7.38: Fork Node GOTO ......................................................... 137
Figure 7.39: Control Flow Graph of Fork Node GOTO ................. 138
Figure 7.40: Application of COBOL Rules ................................. 144
Figure 8.1: One HTML Code Example ........................................ 151
Figure 8.2: Class Diagram of Code in Figure 8.1 ......................... 151
Figure 8.3: Text Class ................................................................. 154
Figure 8.4: Image Class ............................................................. 154
Figure 8.5: Table Class ............................................................. 155
Figure 8.6: Frame Class ............................................................ 156
Figure 8.7: Link Class ............................................................... 156
Figure 8.8: Class Diagram of One Web Block ......................... 156
Figure 8.9: First Class Diagram of Sequential Link-Based Model .... 158
Figure 8.10: Second Class Diagram of Sequential Link-Based Model 158
Figure 8.11: Third Class Diagram of Sequential Link-Based Model 158
Figure 8.12: Fourth Class Diagram of Sequential Link-Based Model 159
Figure 8.13: One Class Diagram of Cyclical Link-Based Model .... 159
Figure 8.14: One Class Diagram of Composite Link-Based Model ... 160
Figure 8.15: Composite Structure Diagram of Text Class of HTML 161
Figure 8.16: Composite Structure Diagram of HTML Image Class ... 161
List of Figures

Figure 8.17: Composite Structure Diagram of Table Class .......................................................... 161
Figure 8.18: Composite Structure Diagram of Frame Class of HTML ............................................. 161
Figure 8.19: Composite Structure Diagram of Link Class of HTML ............................................... 162
Figure 8.20: One Component Diagram of Example In Figure 8.1 ..................................................... 162
Figure 8.21: Deployment Diagram of One Example In Figure 8.1 ..................................................... 163
Figure 8.22: HTML Rules Application ............................................................................................. 165
Figure 9.1: One Example of Class Diagram ....................................................................................... 171
Figure 9.2: Database Class .................................................................................................................. 172
Figure 9.3: Leaf Class ........................................................................................................................ 173
Figure 9.4: Node Class ....................................................................................................................... 173
Figure 9.5: Generation Database-Based Model ................................................................................... 174
Figure 9.6: Association Database-Based Model .................................................................................. 174
Figure 9.7: First Example of Composition Database-Based Model ................................................... 174
Figure 9.8: Second Example of Composition Database-Based Model .............................................. 175
Figure 9.9: Third Example of Composition Database-Based Model .................................................. 175
Figure 9.10: Fourth Example of Composition Database-Based Model ............................................. 176
Figure 9.11: Composite Structure Diagram of Database Class ......................................................... 177
Figure 9.12: Composite Structure Diagram of Procedure Class ....................................................... 177
Figure 9.13: Activity Diagram of Getting Integer Part of Number ..................................................... 178
Figure 9.14: Association Database-Based Model ............................................................................... 181
Figure 9.15: Generation Database-Based Model ............................................................................... 181
Figure 9.16: First Example of Composition Database-Based Model .................................................. 182
Figure 9.17: Second Example of Composition Database-Based Model ............................................ 182
Figure 9.18: Third Example of Composition Database-Based Model .............................................. 182
List of Figures

Figure 9.19: Fourth Example of Composition Database-Based Model ................................................... 183
Figure 9.20: SQL Rules Application ........................................................................................................ 184
Figure 10.1: Tool Architecture ................................................................................................................ 187
Figure 10.2: Opening Source Code ......................................................................................................... 190
Figure 10.3: Procedure-Based Model of Manager Application System ................................................... 192
Figure 10.4: Parsing Part COBOL Legacy System .................................................................................. 193
Figure 10.5: Synthetic Procedure-Based Model of Part COBOL Code ................................................... 193
Figure 10.6: Slicing Part COBOL Legacy System ................................................................................... 194
Figure 10.7: Class Diagram of Part COBOL Code ................................................................................. 195
Figure 10.8: Class Diagram NODE-Task-1-Process ............................................................................. 198
Figure 10.9: Class Diagram NODE-Task-2-Process ............................................................................. 198
Figure 10.10: Class Diagram NODE-Task-3-Process ......................................................................... 199
Figure 10.11: Class Diagram NODE-Task-4-Process ......................................................................... 199
Figure 10.12: Class Diagram NODE-Task-5-Process ......................................................................... 199
Figure 10.13: Class Diagram NODE-Task-6-Process ......................................................................... 200
Figure 10.14: Class Diagram NODE-Task-7-Process ......................................................................... 200
Figure 10.15: Class Diagram NODE-Setup-Unix-Server-File-Transfer-Session ................................. 201
Figure 10.16: Class Diagram NODE-Establish-Data-Port-Connection ............................................... 201
Figure 10.17: Class Diagram NODE-Establish-Unix-Data-Port-Connection ....................................... 201
Figure 10.18: Class Diagram Initialisation ............................................................................................ 202
Figure 10.19: Composite Structure Diagram Initialise-File-Copy-Tables ............................................. 203
Figure 10.20: Composite Structure Diagram All-Tasks-Complete-Process ........................................ 203
Figure 10.21: Composite Structure Diagram Get-Download-File-Size ............................................... 203
Figure 10.22: Sequence Diagram NODE-Task-1-Process ................................................................... 206
List of Figures

Figure 10.23: Sequence Diagram NODE-Task-2-Process ................................................................. 207

Figure 10.24: Sequence Diagram NODE-Task-3-Process ................................................................. 207

Figure 10.25: Sequence Diagram NODE-Task-4-Process ................................................................. 208

Figure 10.26: Sequence Diagram NODE-Task-5-Process ................................................................. 208

Figure 10.27: Sequence Diagram NODE-Task-6-Process ................................................................. 209

Figure 10.28: Sequence Diagram NODE-Task-7-Process ................................................................. 210

Figure 10.29: Sequence Diagram NODE-Setup-Unix-Server-File-Transfer-Session ....................... 210

Figure 10.30: Sequence Diagram NODE-Establish-Data-Port-Connection ...................................... 210

Figure 10.31: Sequence Diagram NODE-Establish-Unix-Data-Port-Connection .............................. 211

Figure 10.32: Sequence Diagram Initialisation .................................................................................. 211

Figure 10.33: Compositive Link-Based Model of Weather-ABCNews System ............................. 214

Figure 10.34: Class Diagram Local Header of ROOT-ABC-News .................................................. 215

Figure 10.35: Class Diagram Local Main of ROOT-ABC-News ..................................................... 216

Figure 10.36: Class Diagram Local Footer of ROOT-ABC-News ................................................... 216

Figure 10.37: Class Diagram Additional Note of ROOT-ABC-News ............................................ 217

Figure 10.38: Class Diagram ROOT-ABC-News ........................................................................... 217

Figure 10.39: Class Diagram Former Note of NODE-Weather-ABC News ................................. 218

Figure 10.40: Class Diagram Local Header of NODE-Weather-ABC News .................................. 218

Figure 10.41: Class Diagram Local Main of NODE-Weather-ABC News ..................................... 218

Figure 10.42: Class Diagram Local Footer of NODE-Weather-ABC News .................................... 219

Figure 10.43: Class Diagram Additional Note of NODE-Weather-ABC News ............................ 219

Figure 10.44: Class Diagram NODE-Weather-ABC News ............................................................. 220

Figure 10.45: Class Diagram LEAF-Weather-New South Wales-ABC News ............................ 221

Figure 10.46: Class Diagram LEAF-Weather-Victoria-ABC News .............................................. 222
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.47</td>
<td>Class Diagram LEAF-Weather-Queensland-ABC News</td>
<td>223</td>
</tr>
<tr>
<td>10.48</td>
<td>Composite Structure Diagram of LINK-TopStories</td>
<td>224</td>
</tr>
<tr>
<td>10.49</td>
<td>Composite Structure Diagram of LINK-NSWNews</td>
<td>225</td>
</tr>
<tr>
<td>10.50</td>
<td>Composite Structure Diagram of LINK-NSW</td>
<td>225</td>
</tr>
<tr>
<td>10.51</td>
<td>Composite Structure Diagram of LINK-Section</td>
<td>226</td>
</tr>
<tr>
<td>10.52</td>
<td>Composite Structure Diagram of LINK-NewsHome</td>
<td>226</td>
</tr>
<tr>
<td>10.53</td>
<td>Composite Structure Diagram of TABLE-Weather</td>
<td>227</td>
</tr>
<tr>
<td>10.54</td>
<td>Composite Structure Diagram of InternationalCentres</td>
<td>227</td>
</tr>
<tr>
<td>10.55</td>
<td>Component Diagram of Weather-ABCNews</td>
<td>227</td>
</tr>
<tr>
<td>10.56</td>
<td>Deployment Diagram of Weather-ABCNews</td>
<td>228</td>
</tr>
<tr>
<td>10.57</td>
<td>Generation Database-Based Model of Simulating INSERT</td>
<td>229</td>
</tr>
<tr>
<td>10.58</td>
<td>Class Diagram of Simulating INSERT System</td>
<td>230</td>
</tr>
<tr>
<td>10.59</td>
<td>Composite Structure diagram of LEAF-sp_generate_inserts</td>
<td>230</td>
</tr>
<tr>
<td>10.60</td>
<td>Activity diagram of LEAF-sp_generate Inserts</td>
<td>232</td>
</tr>
<tr>
<td>10.61</td>
<td>Activity Diagram of ROOT-Master</td>
<td>233</td>
</tr>
</tbody>
</table>
List of Tables

Table 2.1: Static and Dynamic Diagrams of UML .................................................................................... 25
Table 7.1: One Example of Node Layer NLO() .......................................................................................... 139
Table 10.1: Table of Program Elements In Source Code ........................................................................ 189
Table 10.2: Table of Procedure Calling Relationships and Procedure Layers ....................................... 189
Table 10.3: Procedure Layers With Procedures.......................................................................................... 190
Table 10.4: Table of Leaf Classes ............................................................................................................ 194
Table 10.5: Table of Node Classes .......................................................................................................... 195
Table 10.6: Table of Leaf Program Elements ........................................................................................... 203
Table 10.7: Table of Node Program Elements ........................................................................................ 204
Table 10.8: Table of Program Elements In Source Code ........................................................................ 212
Table 10.9: Table of Web Link Relationships and Layers ....................................................................... 212
Table 10.10: Web Layers ......................................................................................................................... 213
Table 10.11: Table of Node Classes ........................................................................................................ 215
Table 10.12: Table of Leaf Classes ........................................................................................................ 215
Table 10.13: Table of Program Elements in Source Code....................................................................... 229
Chapter 1

Introduction

1.1 Proposed Research

The research in this thesis presents an approach to the software evolution of domain-specific legacy systems. When software first began to be used, a typical, logical procedure application in programming took input data, processed them, and produced output data. The programming challenge was seen as being how to write the code logic in such a way as to correspond with the rules of business. Although code logic and business rules are still important, how to reuse the valuable information in legacy systems has become even more important than the actual logic required to manipulate them, especially in domain-specific legacy systems.

COBOL legacy systems are very popular in the real world. Up to 75% of all business data are processed in COBOL. There are between 180 billion and 200 billion lines of COBOL code in use worldwide. The use of COBOL is growing by over a billion lines per year, and 15% of all new applications (5 billion lines) during 2005 were in COBOL.

HTML legacy systems are used all over the world because of the Internet. HTML is the description of the structure, content and links of a web page. It concentrates on the information exchange by means of web pages. It is now hard to imagine how the world worked without the WWW.

SQL legacy systems are very important and affect everyone. SQL, which stands for Structured Query Language, includes additional functionality designed to support Microsoft SQL servers. It is common for a large-scale database to use SQL to facilitate the needs of database users and administrators. This language offers a flexible interface for databases of all shapes and sizes.

In order to facilitate software evolution of COBOL, HTML and SQL legacy systems, parsing of those legacy systems is performed to ascertain the judgement conditions of
their development/environment-specific models, based on the characteristics and operations of those legacy systems. The development/environment-specific model of a COBOL legacy system can be one of four procedure-based models: linear, branch, joint, or synthetic; HTML legacy systems have three possible link-based models: sequential, cyclical, and compositive; and SQL legacy systems have three possible database-based models: association, generation and composition.

There are two major stages of UML extraction from domain-specific legacy systems as being structural and behavioural described in this thesis based on the development/environment-specific models of domain-specific legacy systems.

The structural stage of UML extraction in this thesis contains class realisation. The classification of classes from COBOL legacy system is two, which are procedure class and variable class. Every procedure in COBOL legacy system is defined as one procedure class. Variable class is based on the program slicing techniques with two stages of pseudo class and real class extraction from COBOL legacy system. The variable of the sliced criterion is defined as the class name, and the variables contained in its slicing criterion are defined as the attributes of that variable class. Because the behavioural analysis of domain-specific legacy systems is behind the analysis of structural analysis, the operations in variable class are not described. The classification of classes of HTML legacy system is based on the web pages and their blocks. The classification of SQL legacy system is two, which is procedure class and database class. Selected UML diagrams are used to describe the static aspect of domain-specific legacy systems.

The behavioural stage of UML extraction in this thesis focuses on the operations and activities of domain-specific legacy systems. When understanding the operations and activities of domain-specific legacy code, their preconditions and post-conditions must be presented from the source code. Then those operations and activities are ordered according to the time and sequence they are executed by. At last, the operation and activity arrays are presented. Selected UML diagrams describing the dynamic aspect of domain-specific legacy systems are realised based on those operation and activity arrays.

Original COBOL code is sliced according to program slicing techniques, and four UML
diagrams are presented, two of which are class diagram and composite structure diagram to describe the static aspect of COBOL legacy code, and two of which are sequence diagram and interaction overview diagram to describe the dynamic part of the COBOL legacy code. HTML code is depicted with four UML static diagrams, which are class diagram, composite structure diagram, component diagram and deployment diagram. Legacy SQL code is modelled with class diagram, composite structure diagram, and activity diagram.

1.2 Overview of Problems

The research in this thesis has to do with the software evolution of domain-specific legacy systems. With the rapid development of computer science and technology, more and more software systems have become legacy ones. The gaps between the practical needs and the capabilities of legacy systems are becoming deeper and wider. With changes in the business environment, some software systems are old-fashioned, and particularly difficult to modify, but still valuable. On the one hand, these legacy systems are critical to business. Legacy systems may contain vital business information that is central to an organisation. Although the code is probably difficult to read and understand, it may be the only record of the operation’s rationale. On the other hand, it is costly to maintain these systems. The accuracy of static software systems must be set against the changing and dynamic businesses environment. Legacy systems may have a high rate of operational failure. They often cause their own problems and sometimes even present core risks, especially when the developers of the legacy software system have gone, or the documentation is not complete. It is difficult to correct its errors or maintain the system.

Environmental changes generate need for changes in supporting software. Users of legacy systems find that those systems are hard to use and have a high element of risk. But those legacy systems are the essence of businesses and organisations. It is clear to see that with legacy systems a large number of users are working with outdated skills on old technologies and languages. Even if they know some new technologies that correspond with the needs of the new skills, because legacy systems are large and complicated, it is difficult for them to cope with the problems of those legacy systems. Disasters are inevitable for legacy systems within a changing environment. Although
Chapter 1. Introduction

legacy systems were designed using old technologies and maybe often resulting in increased risk, they are central because they are valuable to those organisations and may even be the only record of key information. Evolution of legacy systems is critical to the organisations using them, because reliance on legacy systems could be vital to them.

Many organisations are faced with maintaining old-fashioned software systems that are constructed to run on a variety of hardware types and programmed in obsolete languages, and they suffer from the disorganisation which results from prolonged maintenance.

As the software ages, the task of maintaining it becomes more complex and more expensive. Poor design, unstructured programming methods, and crisis-driven maintenance can contribute to poor code quality, which in turn affects understanding. It is vital for organisations to solve their legacy software problems. It is important to understand the nature of legacy systems and then migrate them to new states so that they become more efficient, more effective, more reusable, and more accurate.

Software evolution techniques include formal methods. Although these formal methods are founded on theoretical work, they still have drawbacks in their practical applications to real world legacy systems. The lack of a common platform leads to confusion and difficulties with the comprehension and reuse of original code.

Traditional studies used Weiser's program slicing approach to compute consecutive sets of transitivity relevant statements based on data flow and control flow dependences. Although a large number of legacy systems have data flow and control flow dependences, some do not, such as HTML legacy system.

Traditional studies have presented all the UML diagrams. But in practice, some of the UML diagrams are similar. For example, the class diagram is the most fundamental of the UML diagrams for modelling the structure of legacy systems. An object has the same characteristics as the corresponding class. The class is the abstraction of the common characters of the object group. Most of the important characteristics of the object are reflected in the corresponding class. If the class diagram is used in modelling, the object diagram is superfluous.

There are some other problems with traditional studies of software evolution. These
problems are potentially dangerous and they need to be dealt with urgently.

1.3 Scope of Thesis and Original Contributions

This thesis presents the development/environment-specific models of domain-specific legacy systems and creates a number of UML diagrams to deal with the problems inherent in traditional studies.

A model is a representation designed to show the structure or workings of a system. It is a study of a miniature version of the actual. A model enables IT to be more efficient in reacting to business users' requests for new systems or changes to existing ones, and makes it possible to build an application once and use it many times so as to be able to react more quickly to business changes. It reduces applications to their basic components, shares various pieces of functionality across applications, and builds a framework. It even has the ability to help break down the applications supporting various products into pieces that can be saved or discarded as duplicates in order to integrate the users' new products with their existing products, resulting in no overlap. The usage of a model overcomes the disadvantages of traditional studies of software evolution.

UML 2.0 uses six diagrams to model the static parts of legacy systems and seven diagrams to model the dynamic parts. In practice, when a class diagram describes the structure of a legacy system, an object diagram is not used because the class is the abstraction of the common characteristics of the object group and the object has the same characteristics as the corresponding class. And there are other similar situations. Moreover, it is almost impossible to analyse a legacy system at any stage, from every point of view at every layer of the system, because of its large size and great complexity, and the complexity of UML. At the same time, it is not necessary to do it. The proposed approach presents some of the UML diagrams to describe domain-specific legacy systems.

Program slicing techniques compute consecutive sets of transitively relevant statements based on data flow and control flow dependences. Although a large number of legacy systems have data flow and control flow dependences, some others do not, such as HTML legacy systems. HTML describes web and the data formats. So program slicing
techniques are not suitable for analysing all legacy systems.

Development/environment-specific models of domain-specific legacy systems are presented. COBOL legacy systems have linear, branch, joint, and synthetic procedure-based models, HTML legacy systems have sequential, cyclical and compositive link-based models, and SQL legacy systems have association, generation and composition database-based models.

The major contribution of this thesis is the presentation of development/environment-specific models of legacy systems and an approach towards the software evolution of domain-specific legacy systems through using a selected number of UML diagrams. In concrete terms, the original contributions (OC=Original Contribution) of this thesis are described as follows:

OC1: Development/environment-specific models of domain-specific legacy systems are defined based on the characteristics and operations of domain-specific legacy systems. The thesis integrates formal methods and cognitive methods of software evolution and contains structural and operational information regarding working flow or executable functions.

OC2: A selected number of UML diagrams are used to represent domain-specific legacy systems.

OC3: A system is developed to demonstrate the effectiveness of the proposed approach by applying evolution rules together with the integration of evolution rules and a model application based on parsing and slicing domain-specific legacy systems.

OC4: A set of rules is devoted to modelling domain-specific legacy systems together with their application conditions.

1.4 Criteria for Success

The following criteria are given for judging the success of the research described in this thesis:

- For those domain-specific legacy systems, is it possible and necessary for them to be modelled with UML?
Chapter 1. Introduction

- Is it necessary for domain-specific legacy systems to be modelled using all of the UML diagrams?
- If it is not necessary, how many UML diagrams are suitable for modelling COBOL legacy systems? What are they? Are they enough for the modelling? Why?
- How many UML diagrams are suitable for modelling HTML legacy systems? What are they? Why?
- How many UML diagrams are suitable for modelling SQL legacy systems? What are they? Why?
- Is it necessary for all legacy systems to be sliced? Why?
- Is it helpful to use models in software evolution?
- Is it appropriate to build development/environment-specific models of COBOL, HTML, and SQL legacy systems?
- Is the realisation of those UML diagrams from COBOL legacy system appropriate?
- Is the realisation of those UML diagrams from HTML legacy system appropriate?
- Is the realisation of those UML diagrams from SQL legacy system appropriate?

The first question is related to Chapter 2. The following five questions are closely related to Chapter 3. The seventh question is related to Chapter 4, 5, and 6. And the final three questions are related to Chapter 7, 8 and 9, respectively.

1.5 Thesis Structure

The thesis is organised as follows:

Chapter 1 gives the scope and original contribution of the thesis.

Chapter 2 introduces the background of the research, including legacy system, reverse engineering, and software evolution.

Chapter 3 presents a detailed description of the nature of the work in this thesis. Program slicing techniques are not suitable for analysing all legacy systems. Not all UML diagrams are used to describe legacy systems. The development/environment-specific models of legacy systems are presented, and based
Chapter 1. Introduction

on those models, legacy systems are modelled with a selected number of UML diagrams.

Chapter 4 describes the four linear, branch, joint and synthetic procedure-based models of COBOL legacy systems based on COBOL's long history, characteristics and operations, and the calling or being-called relationships between two procedures.

Chapter 5 describes the sequential, cyclical, and compositive link-based models of HTML legacy systems based on HTML's operations and characteristics, and the linking or being-linked relationships between two web pages.

Chapter 6 describes the association, generation and composition database-based models of SQL legacy systems based on SQL's operations and characteristics, and generation and association relationships between databases.

Chapter 7 discusses and presents static parts of COBOL legacy code modelled with the UML class and composite structure diagrams for extracting classes from legacy code based on program slicing, and the dynamic parts modelled with the sequence and interaction overview diagrams.

Chapter 8 shows that only static aspects of HTML legacy systems are modelled with the UML class, composite structure, component and deployment diagrams. It does not have dynamic parts.

Chapter 9 explores SQL legacy systems modelling with the UML class, composite structure and activity diagrams.

Chapter 10 describes experiments that model these three domain-specific legacy systems with a selected number of UML diagrams.

Chapter 11 discusses the proposed approach and draws the conclusions.

Appendix A presents a COBOL legacy system used in experiments.

Appendix B presents a HTML legacy system used in experiments.

Appendix C presents a SQL legacy system used in experiments.

Appendix D lists all the relevant publications by the author during the PhD study.
Chapter 2

Background

2.1 Introduction

In this chapter, legacy systems, software evolution, program slicing, model-driven engineering and UML are introduced. Legacy systems are complicated, heavily modified, difficult to maintain and old-fashioned software that is still important to the organisation because of the changes to the environment and practical rules. Software evolution is the process of improving the quality or making use of all or part of software systems. UML is a good modelling platform for legacy systems. Program slicing is useful in debugging and program analysis, program differencing and integration, software maintenance, testing, tuning compilers, and more.

2.2 Legacy Systems

2.2.1 Definition of Legacy Systems

Software systems are becoming larger and more complicated with the rapid development of applications and requirements. Moreover, it is easy for them to become old-fashioned and turn into legacy systems. It is necessary to comprehend legacy systems, refine the users' requirements, and model them in order to maintain, modify and reuse them [3].

A legacy system is typically large, complicated, old, heavily modified, difficult to maintain and old-fashioned software that is still important to the organisation. A legacy system is a computer system or application program that continues to be used because of the cost of replacing or redesigning it and at the same time despite its poor competitiveness and compatibility with modern equivalents.

Many descriptions of legacy systems are presented from different points of view.
Chapter 2 Background

Legacy systems are large software systems that people do not know how to deal with, but that are vital to the organisation [14]. A legacy system could be any information system that significantly resists modification and evolution to meet new and constantly changing business requirements [26]. It is critical software that cannot be modified efficiently. It is a system that was developed at some time in the past and which is critical to the business in which the system operates [109]. Many legacy systems remain supportive of core business functions and are "indispensable" to the business [90].

A legacy system is therefore one that is large, monolithic and difficult to modify. If legacy software only runs on antiquated hardware, the cost of maintaining legacy software may eventually outweigh the cost of replacing both the software and hardware, unless some form of emulation or backward compatibility allows the software to run on new hardware. It is important to note that the term "legacy" refers to the state of a system before the strategic change. Legacy is a function of the change of a system. It is a result of the change of environment. Without change, there would be no legacy.

It is essential to realise that legacy systems are not useless. On the contrary, legacy systems are important and valuable.

2.2.2 Two Aspects of Legacy Systems

Legacy systems have static and dynamic aspects. The static aspects include software elements and their relationships. The dynamic aspects mainly concern the sequential events that perform tasks.

In static modelling, abstract high-level elements to be found and constructed might represent subsystems or other logically connected software elements. In dynamic modelling, abstractions are typically behavioural descriptions that show interactions amongst high-level static elements.

Both static and dynamic aspects of legacy systems can be modelled in different layers.

2.2.3 Three Types of Legacy Systems to Be Discussed

The research in this thesis focuses on three commonly-used domain-specific legacy software systems—COBOL, HTML and SQL legacy systems.
Chapter 2 Background

COBOL legacy systems perform batch information tasks, and contain many valuable messages. Software written in COBOL has characteristics common to that written in other multi-programming languages, such as BASIC, C, COBOL, DELPHI, FORTRAN, PASCAL, etc., all of which contain data flow and control flow.

HTML legacy systems are used in WWW-like systems written in XML, JAVA, etc. HTML presents the data format and does not have dynamic characteristics.

SQL legacy systems contain numerous tables and databases. SQL defines the methods used to create and manipulate relational databases on all major platforms. It is common for large-scale databases to use SQL to facilitate database users and administrators interactions. It offers a flexible interface for databases of all shapes and sizes.

2.3 Software Evolution

2.3.1 Software Engineering

Software engineering is important for software development, improvement, maintenance, modification and reuse [47]. It is one of the most important areas of computer science. In the IEEE standards [71], the definition of software engineering is presented as follows:

"Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software."

Although there are many characteristics of software evolution, change and complexity are the two main themes [85, 140]. It is the change that leads to software aging. Systems evolution is the selective application of scientific and engineering efforts to transform an operational need into a description of the system configuration which best satisfies the operational need, according to the measures of effectiveness being used. It integrates related technical parameters and ensures compatibility of all physical, functional and technical program interfaces in a manner which optimises the total system definition and design; and integrates the efforts of all engineering disciplines and specialties into the total engineering effort [41, 54]. It results in saving time and decreasing the costs of software applications.
2.3.2 Software Reengineering

2.3.2.1 Definition of Software Reengineering

Reengineering is the bridge used by legacy software to migrate to an organisation’s new maintenance environment. Reengineering is the examination and modification of a system to reconstitute it in a new form and the subsequent implementation of the new form. Because the legacy software’s quality, performance, reliability and maintainability are deteriorating, it is necessary for legacy systems to be reengineered. Reengineering legacy systems is fundamentally different from software maintenance, system redevelopment, and continuous improvement [7]. It involves three main steps: restructuring, reverse engineering, and forward engineering [13].

2.3.2.2 Reasons for Reengineering

The reasons for reengineering legacy systems are as follows:

- Reengineering can capture design information from source code. It can help to supplement the legacy documentation in order to comply with documentation standards, give structure to previously unstructured software and adapt initial data and code to new programming languages, configurations or platforms [137].

- It allows legacy software to adapt quickly to changing environments. The changing of the environment is pervasive and happens with speed. It is important for the business to act consistently in the light of changing situations.

- It complies with new organisational standards. New standards represent new business rules and methods. To avoid failure in business, the company must apply new standards. Its software system should be reengineered to be sufficient for its new business needs.

- It upgrades to newer technologies, platforms or paradigms. New approaches are presented continuously for producing the company’s products. Those new approaches mean saving time and earning more money. In order to be ahead of its competitors, the company should upgrade its legacy systems in good time.

- It extends the software’s life. After being reengineered, software systems have new environmental characteristics. They are able to be continuously utilised.

- It identifies class candidates for reuse. The name, attributions and operations of
Chapter 2 Background

every class candidate are recognised through the process of reengineering the original code.

- It improves software maintainability by increasing the productivity of every maintenance programmer, thus reducing reliance on programmers who have specialised in a given software system, and reducing maintenance errors and costs.

Software maintenance is the modification of a software product after delivery to correct errors, improve performance (or other attributes), or adapt to new requirements [12, 35]. Software maintenance is defined in IEEE Standard 1219 as: the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment [70]. A similar definition is given by ISO/IEC [73], again stressing the post-delivery aspect: The software product undergoes modifications of code and associated documentation, due to a problem or the need for improvement [72].

The objective is to modify the existing software product whilst preserving its integrity. The increasing problems of maintaining software are as follows [121]:

- Software errors can be very expensive. Because environmental change is pervasive, it is difficult to predict the errors resulting from legacy systems. Only when the damage has been done is the need realised to modify the corresponding software.

- Software maintenance is very costly. Continual software system maintenance means terminating working procedures, delaying delivery dates, disengaging machines and employees and resulting in expensive knock-on effects for the services offered by the business.

- Maintenance people are getting scarce as increasingly convenient tools and programming languages are designed that do not require the same levels of skill that were needed in the past. Maintaining a software system that depends on a programming language that is seldom used is hard work. It is difficult to find people with those skills.

- Software maintenance results in frequent failures, complex designs, unpredictable effects, unreliable or missing documentation, obsolete hardware platforms, loss of experienced maintenance programmers or original developers and growing backlogs.
Redevelopment of the software system is quite different from reverse engineering because:

- Critical corporate knowledge is contained within legacy software. Although the legacy system may be poor in many respects, its loss could be fatal to the business organisation. Legacy software represents an enterprise model of the business.

- A legacy system is a valuable asset. It can be of fundamental value to the business and contain central business information.

- Reusable and reengineered software costs much less than redeveloped software.

Continuous improvement of a software system is different from its reverse engineering. Some organisations have decided that adopting software reengineering and a new maintenance environment are steps that are too radical to take. Instead, these organisations improve their maintenance environment gradually by using better tools, processes or people. This is defined as continuous improvement in quality. In fact, it is almost impossible to continuously improve a large, complicated software system. Continually modifying software will inevitably result in frequent training of the employees, many man-made errors, and maybe even complete failure of the business. In some special cases, the hardware running the software is no longer produced. Software must be transformed to a new platform in order to protect the useful information it contains. On the other hand, the benefits of software reengineering far outweighs those acquired through continuous improvement.

### 2.3.2.3 Classification of Reengineering

Reengineering is classified into forward engineering and reverse engineering, from the point of view of basic operations. There are four levels of software abstraction in reengineering, namely axioms, requirements, design, and implementation.

An axiom is any fundamental, self-consistent, universally accepted tenet about a software program’s context. Axioms may include the software’s purpose, the concepts that imply the requirements, and details of the source and nature of the information to be processed by the software. Axioms place the software against a background of non-software subjects. They explain the nature and purpose of the software without providing details about the actual software, such as requirements, design, and implementation. A complete software axiom set allows the creation of a complete set of...
Chapter 2 Background

requirements through appropriate forward engineering operations, without the need for additional information.

A requirement is a well-formed statement with respect to the axioms of the conditions necessary for successful software use, conditions prevailing after software use, and constraints on software use. Requirements state what the system does with the resources provided, not how the system does it. A complete software requirements set allows the creation of a complete design through appropriate forward engineering operations without the need for additional information [61].

A software design is any non-compliable software description that faithfully reflects requirements and states how outputs are produced from inputs. Generally, many design levels sequentially provide greater and greater detail about how the software operates. Each successive design level refines requirements into implementations. The design defines software architectures, organisation, communication, and other details about how the software acts without containing enough detail to actually be compilable. A complete software design allows the creation of a complete implementation through appropriate forward engineering operations.

A software implementation faithfully reflects the design and is syntactically correct with respect to a compilable language. A complete implementation could be compiled to produce an executable code.

2.3.3 Reverse Engineering

Reverse engineering is an important aspect of reengineering [32]. Reverse engineering is the process of transforming code into a model through mapping from a specific implementation language [13, 21]. Reverse engineering is the process of analysing an existing system to identify its components and their interrelationships, and creating representations of the system in another form or at a higher level of abstraction. Reverse engineering is usually undertaken in order to redesign the system for better maintainability or to produce a copy of a system without access to the design from which it was originally produced.

Reverse engineering is a development process based on the notion of taking something apart to see how it works and then putting it back together again [91]. Reverse
Chapter 2 Background

engineering often means decoding or analysing a device or program with the intention of using its technology to create another product [65].

Reverse engineering has the goal of analysing software systems so that the software is more understandable for maintenance, evolution, and reengineering purposes [105]. It analyses a subject system in order to identify its current components and their dependencies so as to extract and create system abstractions and design information.

The rationale of reverse engineering can be described as follows:

- Abstraction stresses the palpable aspects of a software system and conceals details.
- Representations of a software system at higher levels of abstraction are more compact and easier to understand [140].
- The entities of the abstraction of the system are easier to reason about and are closer to the application domain than the source code.
- Opportunity increases for coarse-grained reuse and modern design techniques.
- Abstraction mechanisms serve as organisational axes and design methods.
- Abstractions represent architectural documentation and test plans.

The modelling foundation for reverse engineering can be summarised as follows:

- Abstraction mechanisms.
- Artifact. Defined as a component relationship attribute.
- Classification. This abstraction captures common properties shared by a collection of artifacts.
- Aggregation. Establishes a partial relationship between a composite artifact and its constituents.
- Generalisation. Relates an element to a more general element.
- Grouping. Groups a set of elements and relationships pertinent to those elements to form a context.
- Abstraction hierarchies. Abstraction mechanisms can all be applied recursively to form aggregation, generalisation, and grouping hierarchies.

Reverse engineering activities include
Chapter 2 Background

- Gathering artifacts, which include source code, design documents, specification documents, documentation, corporate knowledge, application knowledge, syntactic pattern matching, developer knowledge and experience.

- Populating a repository. Selecting relevant information, filtering out immaterial information, modelling information according to schema, and loading the repository.

- Building layers of abstractions. These layers are: structure level, function level, and application level. Also, semantic and behavioural pattern matching.

- Building taxonomies and hierarchies. Includes classification, aggregation, generalisation and grouping.

Definitions of reverse engineering views are

- A view represents a perspective on a software system and the reverse engineering process.

- A view is a bundled set of software artifacts.

- A view captures the state of the user interface, memory data structures and repository.

- A view classification scheme includes goal structure, environment interaction, functional substructure, entity structure, process structure and system dynamics, allocation aspects, and realisation consideration.

- Implementation-level views include text and source text, syntax trees, resource flows that contain control flow and data flow, cross references, data structures and statistics.

- Structural views include structure charts, call graphs, module and subsystem graphs, metrics and complexity views, and organisational views.

- Functional views include design documents, specification documents, and requirements documents.

- Behavioural views include conceptual views, temporal views, user interaction, process views, and domain views.

Reverse engineering is the process of analysing an existing system in order to identify its components and their interrelationships and create representations of the system in
another form or at a higher level of abstraction [136]. Reverse engineering is usually undertaken in order to redesign the system for better maintainability or to produce a copy of a system without access to the design from which it was originally produced [36].

2.3.4 Software Evolution

2.3.4.1 Definition

Software evolution is recognised as one of the most essential activities of software systems. It has become one of the most important research subjects in the field of software systems in recent years. Software evolution is regarded as the main driver of software maintenance activities [141].

The first definition of Software Evolution is presented by Research Institute for Software Evolution (RISE) formerly the Centre for Software Maintenance (CSM), which was established in April 1987, at the University of Durham, England. It is the first such centre world-wide to concentrate its research on software evolution.

Informally, software evolution refers to all those activities that take place after a software product has been delivered to the customer, but the more formal definition used by the RISE is [111]:

"Software evolution is the set of activities, both technical and managerial, that ensures that software continues to meet organisational and business objectives in a cost effective way."

Software evolution is a long-term process involving the execution, usage, improvement, extension and update of software systems [8]. It is the instinctive demand for software to correspond with changes of context and the new necessities of the real world.

A software system undergoes many complex maintenance activities throughout its lifecycle, such as correcting faults, improving performance, adapting the product to a new environment, or adding new functionality. Software evolution forms the main part of software maintenance.

With improvements in technology and environmental changes, more and more legacy systems exist. With the high demand for renovation of these software systems, their
evolution is becoming an urgent need.

2.3.4.2 Benefits of Software Evolution

Software evolution of a legacy system aims at improving the quality of the legacy system in order to modify and reuse it for software reengineering. It is regarded as the basis of software reengineering. Better understanding of a program aids in common activities such as performing corrective maintenance, reusing and keeping documentation up to date [139]. To minimise the likelihood of errors being introduced during the change process, the legacy software should be understood sufficiently well for changes made to the source code to have predictable consequences [30]. However, such understanding is difficult to achieve with a legacy system after many years of operation under conditions of changes in the business environment, operational environment and support environment. The goal of software evolution is to acquire sufficient knowledge about a software system so that it can evolve in a disciplined manner. The essence of software evolution is mastering the main tasks that the legacy system performs. The requirements of the software system must be viewed at a high level in order for it to be improved or reused afterwards. Software evolution of a legacy system involves identifying artifacts and understanding their relationships, restructuring the legacy source code that is to be cleaned, eliminating the dead code, refining the requirements of the system, and describing the main tasks of the legacy system at the top level [86].

Software evolution of legacy systems results in many benefits [66, 130]. It displays hard-to-read code visually. During the process of software evolution of a legacy software system, the original code and specification are difficult to read and understand. Because a legacy system is vital to the company’s business and it contains records that are central to the business, software evolution of the source code will uncover them [25]. Software evolution of a legacy system starts from the existing source code. Displaying the original code is the most important start point, no matter when it was designed and how it was programmed. The programming language of the legacy code may be obsolete at present time. The hardware within which the legacy software system is executed may no longer be produced. But the legacy system contains information that is central to the business organisation. It is important to display that hard-to-read source code so as to analyse it [39].
Software evolution identifies poorly written code early. Poorly written code is known as spaghetti code. It is unstructured and extremely difficult to understand. The legacy software system may contain GOTO statements in the programming language COBOL and be confusing and unstructured. The spaghetti code is examined and modified through the software evolution process and it becomes structured and object-oriented. When a local system was designed using older techniques and because of the rapid change of the users' environment the system has become a legacy one. It will be difficult to maintain and modify but still valuable to the business. Software evolution of a legacy system can help to uncover the original code. In the source code, many spaghetti codes are identified through a software evolution approach. Those codes hamper further use of the software system and they can easily present the risks. It is necessary to identify those codes.

Software evolution promotes the following of new project standards. Because of continuous environmental change, new business standards replace old ones and many software systems are becoming legacy ones. The legacy system would have been designed and programmed using old technologies, some of which may be out of date or even harmful in the implementation of the software at the present time. The operational environment, including the hardware and operational system, will have been changed as a result of technological improvements. The legacy software system represents old business standards. In order to be competitive and advanced, the company must adopt new standards and improve its software systems by evolving them. This will result in less time being consumed, thus saving money, and the company's position being strengthened.

Software evolution improves the quality of the software. Software evolution of legacy systems extends the functional life of the source code, develops the reusability and maintainability of the software system, and leads to new approaches to adapting the business [53].

2.3.5 Maintainer's Assistant

Maintainer's Assistant was developed at the University of Durham in the UK. The tool is part of the BYLANDS project, which concentrates on reverse engineering of existing code using formally-proven, semantic-preserving program transformations using the new language, which is known as the Wide Spectrum Language (WSL) [128].
WSL is used in program transformation work, which includes low-level programming constructs. By working within a single formal language, the transformation of a legacy system is able to prove that a program correctly implements a specification, or that a specification correctly captures the behaviour of a program by means of formal transformations in the language [10, 37, 134]. Martin Ward has utilised program transformations in WSL to derive efficient algorithms from abstract specifications. One such algorithm removes inefficient recursion from programs. One of the focuses of Ward's work is to develop transformations that refine a program's specification into an efficient algorithm [127, 129].

WSL was originally designed to simplify proofs of program equivalence, which formally defines syntax and semantics. It is based on the mathematical basis of set theory and first order infinite logic, and every transformation has been rigorously proved [9, 78, 97]. The transformations are represented by MetaWSL, an extension to WSL that incorporates pattern matching, template filling functions, and statements for moving within the AST, etc [6, 27, 29]. The user can select available transformations or write his own transformations using MetaWSL. The use of infinite logic eliminates the need to determine loop invariants or fixed points of functions when transforming loops [25, 31].

Using Maintainer's Assistant, the program code is first translated into WSL. An automatic translator is provided for IBM 370 Assembler. Once in WSL, the user can interactively apply transformations to the code or the assertions of WSL.

2.3.6 Refactoring Browser

Refactoring Browser is a useful tool, proposed by Roberts [114] at the University of Illinois at Urbana-Champaign, that is implemented in VisualWorks and VisualAge for the Smalltalk language. The success of the tool is based on its complete integration with the Smalltalk environment. Refactoring Browser can be considered as an extension to the Smalltalk development browser.

Refactoring Browser operates by first parsing the code to be refactored and creating an Abstract Syntax Tree (AST). The available transformations are encoded as templates in the form of ASTs, which may contain template variables. The transformation is accomplished by a parse tree rewriter that matches the concrete AST with a template AST and performs tree manipulation [60].
Refactoring Browser implements the preconditions proposed by Opdyke, and it also uses postconditions. Postconditions help to eliminate some of the analysis involved in proving preconditions inside composite refactorings. Preconditions are implemented as instances of class conditions that are created and evaluated before applying a transformation. A condition, when evaluated, checks certain information from the Smalltalk environment. Another component of this framework is a change manager, which is responsible for recording which refactorings are performed [113, 114]. This allows for the implementation of undo and logging.

2.3.7 Cognitive Methods

A cognitive model describes the mental process or faculty of knowing a software system. A hierarchy of cognitive design elements to support the construction of a mental model explains how to improve program understanding by supporting the actions of identifying software artifacts and the relationships between them, by browsing code in delocalised plans, and by building abstractions [120]. These actions comprise canonical reverse-engineering activities.

Cognitive methods rely mainly on domain knowledge. In order to jump from one level up to another abstract level in the process of reverse engineering, one has to throw away some information. This abstraction is creative work. In order to achieve correct and practical abstraction, a knowledge base is necessary.

2.4 UML

2.4.1 Definition of UML

UML stands for Unified Modelling Language. The Unified Modelling Language is a graphical language for visualising, specifying, constructing and documenting the artifacts of a software-intensive system. It is a standard language for writing software blueprints and appropriate for modelling ranging from enterprise information systems to distributed Web-based applications and even to hard real-time embedded systems. It is a very expressive language, addressing all the views needed to develop and then deploy such systems.

UML is a non-proprietary, third generation modelling language. A model is a
description of observed behaviour, simplified by ignoring certain details. Models allow complex systems to be understood and their behaviour predicted within the scope of the model. A model may be used as the basis for simulation. The Unified Modelling Language is an open method used to specify, visualise, construct and document the artifacts of a software-intensive system. It represents a compilation of best engineering practices that have proven successful in modelling large, complex systems [57]. It succeeds the concepts of Booch, OMT and OOSE by fusing them into a single, common and widely usable modelling language [102].

UML is an approach to modelling both data and processes that combines the best practices of many professionals in the industry. It is a modelling notation that eliminates the need for different notations for different existing software development methodologies. In addition, UML as a modelling notation has been officially adopted by Object Management Group (OMG), the international standard organisation which consisted of about 800 different organisations as of May, 2002. Many different CASE tools support UML, and these can create both relational and system schemas for implementing the modelled relationships. These tools can generate source code in more than a dozen different languages and thus support the initial creation of the software that implements the application's functionality. UML is also the basis of Microsoft's repository and is being extended to support enterprise modelling [75, 118].

2.4.2 History of UML

UML came about from the combined efforts of Grady Booch, with his Booch method, James Rumbaugh with his Object Modelling Technique (OMT), and Ivar Jacobson, with his Object-Oriented Software Engineering (OOSE) methods. Under the auspices of Rational Software Inc, UML began to take shape in 1994 [99].

The first object-oriented analysis and design (OOAD) methods were published in the late 80s and early 90s. In addition, three independent core UML methods, namely Booch'91, object-oriented modelling and design [116], and object-oriented software engineering [74], were published. The development of UML began in 1994. The first draft, entitled Unified Method 0.8, was released in 1995. It merged the second editions of Booch'91 and OMT-1, namely Booch'93 [20], and OMT-2 [117]. When OOSE was merged into the Unified Method in 1996, the name was changed to the Unified
Chapter 2 Background

Modelling Language (UML). The first official version, UML 1.0, was published in 1997, followed by version 1.1 and 1.3. Another attempt to combine different OOAD methodologies was Fusion [38], which included concepts of OMT, Booch’91, and CRC [133]. Now UML 2.0 has been released, containing thirteen modelling diagrams.

It is obvious that the separation of data and function in structured analysis and design has resulted in a large number of weak points, whilst Object-Oriented Software Engineering (OOSE) modifies many of those weaknesses by regarding a real world system as a whole from an overall perspective. UML is a standard language for modelling systems [75, 118]. It is neither a software analysis and design process nor a software development life cycle. It does not provide a standard process for developing a software system. It is a mechanism for revealing the essence of the system’s business rules and ultimate design. It is a modelling notation for proper system modelling and implementation with consistency and clarity. UML describes the static aspects of the modelled system using object diagrams, class diagrams, collaboration diagrams, component diagrams, package diagrams, composite structure diagrams and deployment diagrams; the dynamic aspects of the system are depicted utilising use case diagrams, sequence diagrams, timing diagrams, interaction overview diagrams, activity diagrams and statechart diagrams [99].

2.4.3 Views of UML

The definition and functionalities of UML are presented here from two points of view: static and dynamic. UML is neither a software process model nor a systems development life cycle. It is merely a notation. UML is a mechanism for uncovering the essence of a project’s business rules and ultimate design. It provides a consistent model for proper software implementation and consistent feedback so as to ensure that a project sponsor understands the project [15].

In UML, there are static diagrams and dynamic diagrams that model the static and dynamic aspects of the software respectively. The number and type of diagrams to be used depend on the purpose and needs.

2.4.4 Properties of UML

As a collection of modelling methods, UML has a list of properties that satisfy the
needs of legacy systems modelling [142].

- Simple navigation with minimum disorientation: UML is structured and includes features to aid the user in navigating the modelling.
- More information presented: UML presents as much information as possible without overwhelming the user.
- Low modelling complexity: UML is an organic group of existing modelling techniques, containing thirteen diagrams that cover static and dynamic modelling. It visualises the abstraction of system, reducing the complexity of modelling.
- Well-structured presentation: The thirteen UML diagrams each have their own characteristics as a theoretical foundation. They are used graphically in modelling.
- Varying levels of detail: Details, abstraction, information content and type of information vary so as to accommodate users’ interests in the presentation of the thirteen UML diagrams. The class diagram and component diagram, for example, have different levels of modelling.
- Flexibility to change: Small changes of content do not cause major differences in the modelling.
- Convenient interface: The user interface of UML is flexible and intuitive, and avoids unnecessary overheads.
- Suitability for automation: UML has a good level of automation in order to make the modelling of practical value.
- Desirable integration: UML is able to link the modelling and the original information it represents.

2.4.5 UML Diagrams

UML 2.0 includes thirteen diagrams to improve its modelling quality (see Table 2.1).

Table 2.1: Static and Dynamic Diagrams of UML

<table>
<thead>
<tr>
<th>UML Diagrams</th>
<th>Modelling Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>class diagram</td>
<td>Static modelling</td>
</tr>
</tbody>
</table>
UML uses six diagrams to model the static parts of software system, which are the class diagram, object diagram, composite structure diagram, component diagram, deployment diagram and package diagram; and seven diagrams to model the dynamic parts of software system, which are the sequence diagram, activity diagram, communication diagram, timing diagram, interaction overview diagram, state machine diagram and use case diagram. There are four new diagrams in UML 2.0 compared with UML 1.3: package diagram, composite structure diagram, timing diagram and interaction overview diagram.

### 2.4.6 Importance of Modelling Legacy Systems with UML

The need for maintaining, reusing and reengineering existing software systems has increased dramatically over the past few years. Changed requirements need software migration. Reusing and modifying legacy system are complex and expensive tasks because of the time-consuming process of program comprehension. Software evolution
Chapter 2 Background

aims at analysing the software and representing it in an abstract form so that it is easier to understand for software maintenance, reuse and documenting purposes.

To evolve an existing legacy software system, both static and dynamic information are useful. Static information describes the structure of the software when it is written in the source code, while dynamic information describes the runtime behaviour. Both static and dynamic analyses result in information about the software artifacts and their relationships. The dynamic analysis also produces sequential event trace information about concurrent behaviour, code coverage, and control management [5].

Software evolution is supported by producing design models from the legacy software. The software evolution approach is useful when building legacy software into high-level information. The extracted static models are utilised to get an overall picture of the current state of the legacy software. The dynamic models are used to support tasks such as debugging, finding dead code, and understanding the current behaviour of the legacy software.

UML has static and dynamic modelling advantages. It satisfies the needs of software evolution. At the same time, UML presents a visual description of the system and makes the process of software evolution easily acceptable. Meanwhile, a large number of tools support the transformation from UML diagrams to code, and UML facilitates the reusability of software evolution, which is also helpful for forward engineering in the process of reengineering.

The rise of new programming languages and paradigms has driven changes in current software evolution approaches and methods. The present legacy systems are written in COBOL, BASIC, C, FORTRAN, or PASCAL; in future, legacy systems will be written in C++, Smalltalk, or Java. UML is capable of flexibility with object-oriented programming software, and this has led to the long life and widespread use of UML. Therefore, using UML is very important for software evolution.

2.5 Program Slicing

2.5.1 Introduction

Program slicing consists of part of a program that affects the values computed at some
Chapter 2 Background

point of interest, which is termed a slicing criterion. Weiser first gave the formal definitions and algorithms in this area [131, 132]. A program is essentially a sequence of statements and the slicing involves isolating a set of statements to be included in the slice. The obtained new program is still an executable program and is a program slice.

2.5.2 Static and Dynamic Slicing

Program slicing comprises static and dynamic slicing. Dynamic slicing is defined as the kind of isolation that may be for a specific input to the program, whilst static slicing is for all possible inputs.

According to Weiser’s introduction, slices are computed by computing consecutive sets of transitively relevant statements based on data flow and control flow dependencies. In this case, only statically available information is used for computing slices. So this type of slice is referred to as a static slice. A typical method for computing static slices was introduced by Ottenstein [100] in terms of a directed graph termed a Program Dependence Graph (PDG), with vertices corresponding to statements and control predicates, and edges corresponding to data and control dependencies. Another typical method was suggested by Bergeretti and Carre [16] in terms of information-flow relationships in a syntax-directed fashion.

Dynamic program slicing was proposed by Korel and Laski [82], which focuses on how information flows obtain a particular value throughout a program. Only the dependencies that occur in a specific execution of the program are taken into account. A dynamic slicing criterion specifies the input and distinguishes between different occurrences of a statement in the execution history. Dynamic program slicing is especially useful in debugging, with the specific wrong value as the input at the break point of interest. A number of applications have been presented by Choi [34], Duesterwald [50], Kamkar[79], Venkatesh [126], etc.

The main difference between static and dynamic slicing is that dynamic slicing assumes a fixed input for a program, whilst static slicing contains a common input and it does not make assumptions or designate specific values regarding the input.

2.5.3 Applications of Program Slicing

A number of hybrid applications using static and dynamic slicing methods have been
Chapter 2 Background

suggested for solutions involving slicing programs containing procedures, unstructured control flow, composite data types and pointers, and concurrency in terms of accuracy and efficiency. Program slicing is applied in debugging and program analysis, program differencing and integration, software maintenance, testing, tuning compilers, and other situations.

2.5.3.1 Debugging and Program Analysis

Program slicing is fundamentally useful for debugging. Potentially, it allows for the ignoring of many statements in the process of localising a bug. If a program computes an erroneous value for a variable, only the statements in the slice that possibly contributed to the computation of that value are considered. In this case, it is possible that the error has occurred in one of the statements in the slice. And it is also probable that more or different statements will show up in the slice than one would expect. Program slicing shows whether a value is being used afterwards, detects the dead code that does not affect the output of the program, reveals how the computations of values depend on the earlier computations, and compares the intended program behaviour with the actual program behaviour.

2.5.3.2 Program Differencing and Integration

Program differencing is the task of analysing an old and a new version of a program in order to determine the set of program components of the new version that represent syntactic and semantic changes. Such information is useful because only the program components reflecting changed behaviour need to be tested. The key issue in program differencing consists of partitioning the components of the old and new versions in such a way that the two components are in the same partition only if they have equivalent behaviours.

The program integration algorithm compares slices in order to detect equivalent behaviours. Program slicing is used to restate the algorithm of program integration and to prove properties such as associativity of consecutive integrations.

2.5.3.3 Software Maintenance

Program slicing is helpful in determining whether a change at some place in a program will affect the behaviour of other parts of the program in software maintenance. Static
slicing decomposes a program into a set of components, and show how each of those components captures part of the original program’s behaviour, and how changes in a component can be merged back into the complete program in a semantically consistent way.

2.5.3.4 Other Applications

Program slicing can also be used in program testing, for tuning compilers, and in other circumstances.

2.6 Confining Analysis of Legacy Systems using UML

The analysis of legacy systems when using UML must be confined to the real world.

A legacy system implies software that is large, complicated, old, heavily modified, difficult to maintain and old-fashioned. A legacy system is a computer system or application program that continues to be used because of the cost of replacing or redesigning it, despite its poor competitiveness and incompatibility with modern equivalents. It is almost impossible to analyse a legacy system at any stage from every point of view at every layer of the system, because of its large size and convoluted structure, and the complexity of the UML. However, it is not necessary to do this.

When modelling a legacy system with UML, the information in the legacy system is refined by using the UML diagrams [93]. Modelling has the goal of analysing the legacy system in order that the software code be more understandable for maintenance, evolution and reengineering purposes. Analysing a designated system to identify its current classes and their relationships is an operation that extracts and creates system abstractions and design information. Additional knowledge about the system is produced, and legacy software code is analysed for the specific purpose [138]. Therefore, modelling a legacy system with UML is undertaken from a specific point of view. A legacy system is large and complicated because of the use of old technologies. It is almost impossible to depict all the information in a legacy software system from every aspect at every layer to the most detailed extent. However, it is not necessary to use all the thirteen UML diagrams to model a legacy system.

A legacy system has static and dynamic characteristics that display its functionality,
Chapter 2 Background

representing, respectively, its structural and behavioural characteristics. The modelling of a legacy system concentrates on the reflection and comprehension of the legacy system at a high level. The main purpose is to understand the structure of the targeted legacy system and its main tasks. It is necessary to use the appropriate UML diagrams to model different legacy systems.

2.7 Model-Driven Architecture

2.7.1 Model

A model is a pattern, plan, representation (especially in miniature), or description designed to show the main objects or workings of an object, system, or concept [58, 95].

2.7.2 Model-Driven Engineering

Model-Driven Engineering (MDE) refers to the systematic use of models as primary engineering artifacts throughout the engineering life cycle [55, 56]. MDE can be applied to software, system, and data engineering [58, 59].

Model-driven Engineering (MDE) is the unification of initiatives that aim to improve software development by employing high-level, domain-specific models in the implementation, integration, maintenance and testing of software systems [58, 59, 95]. To overcome the abstraction barrier, MDE introduces models that capture designs at a higher level of abstraction. Unlike technical documentation that has a fragile connection to the implementation of a software system, models are an integral part of the software evolution process. Developers represent designs using models that conform to an appropriate metamodel, which are then automatically transformed into implementations. Thus, with an appropriate modelling language, the effort of producing a new software system decreases and maintenance is reduced to model maintenance.

Prominent among MDE initiatives is OMG’s Model-Driven Architecture (MDA) [80, 95], in which software development is envisioned as a series of model transformation steps, starting with a high level specification using a vocabulary that is familiar to the practitioners of the domain in question, and ending with a platform-specific model describing how, for example, the system makes use of certain J2EE features. Related industrial efforts include Microsoft’s DSL framework of software factories [63] for
building stacks of domain-specific languages, and Jetbrain’s Meta Programming System for language-oriented programming.

Model-driven engineering is strongly related to the field of domain-specific languages (DSLs) and generative programming [45, 48, 94]. A domain-specific programming language (domain-specific language, DSL) is a programming language designed for, and intended to be useful for, a specific kind of task. For instance, GraphViz is designed to create images of graphs in a variety of formats, but GraphViz lacks the capability to perform some basic tasks, e.g., accessing a network socket, receiving additional user inputs, or manipulating strings. Creating a domain-specific language while developing a system is a method of language-oriented programming. This is in contrast to a general-purpose programming language, such as C, or a general-purpose modelling language like UML.

While MDE promises to improve productivity and maintainability, widespread adoption and scaling to large software systems requires research into the evolution of model-based systems, the scope and expressivity of modelling languages, and the interaction and integration of models. Software evolution is concerned with the complete life cycle of software systems, from initial development to maintenance, and includes introducing new features, improving old features, and repairing bugs. Whilst the introduction of model-driven engineering brings advantages, it also calls for a new style of evolution [43].

2.7.3 Model-Driven Architecture

Model-driven architecture (MDA) is a software design approach launched by OMG in 2001. MDA supports model-driven engineering of software systems, and provides a set of guidelines for structuring specifications expressed as models [95]. MDA principles can also apply to other areas such as business process modelling [80].

2.8 Summary

The research in this thesis focuses on the software evolution of domain-specific legacy systems. It is based on the background that legacy systems are important and valuable, that they have static and dynamic aspects, and software evolution is the instinctive demand for software to correspond with changes of context and the new necessities of
the real world. This chapter has also introduced the history and various aspects of UML. Because of the large size and complexity of legacy systems, it is not necessary to analyse legacy systems at any stage from every point of view or at every layer of the system.
Chapter 3
Proposed Approach

3.1 Introduction

Software evolution is an integral part of the software life cycle. Furthermore in the recent years the issue of keeping legacy systems operational in new platforms has become critical and one of the top priorities in IT departments worldwide. The research community and the industry have responded to these challenges by investigating and proposing techniques for analyzing, transforming, integrating, and porting software systems to new platforms, languages, and operating environments. Through the extraction of UML diagrams from legacy code, the transformation has realised the new analysis platform on UML in order to be helpful on the comprehension of legacy systems based on the general analysis language UML.

During software evolution, several different facets of the system need to be related to one another at multiple levels of abstraction. Current software evolution tools have limited capabilities for effectively visualizing and evolving multiple system facets in an integrated manner. Many tools provide methods for tracking and relating different levels of abstraction within a single facet. However, it is less well understood how to represent and understand relationships between and among different abstraction hierarchies, i.e. for inter-hierarchy relations. Often, these are represented and explored independently, making them difficult to relate to one another. As a result, engineers are likely to have difficulty understanding how the various facets of a system relate and interact. In this thesis, an approach towards software evolution of domain-specific legacy systems through a selected number of UML diagrams is described to enhance the inter-hierarchy visualization capabilities of an existing software evolution. UML visually helps the comprehension of legacy systems with its different layers of diagrams. This visualization integrates - or "fuses" - facets of architecture, behavior and data of legacy systems.
Chapter 3. Proposed Approach

At the beginning of the software development process, the ideal situation would be for the software used in the design to be well-structured, the testing to be designed alongside the code, the documentation to reflect the appropriate models and rationale, and the changes in requirements and modifications of the software to be reflected in the documentation. However, in the real world things are rarely ideal. After the software system has been used in the specific area, business rules will change, the users of the software system will be replaced, the techniques of developing software systems will become outdated, and the hardware available for executing the software system will prove insufficient to satisfy the user's needs. As teams change and documentation becomes out of date, the code becomes the only guide to system structure and behaviour, and maintenance and evolution tasks are hindered by the inability of developers to comprehend system components and their interactions.

In order to facilitate modelling, development/environment-specific models are introduced based on the characteristics and operations of their domain-specific legacy systems. COBOL legacy systems have four procedure-based models: linear, branch, joint and synthetic; HTML legacy systems have three link-based models: sequential, cyclical, and compositive; and SQL legacy systems have three database-based models: association, generation and composition.

Original COBOL code is sliced, and it is described according to class, composite structure, sequence and interaction overview diagrams. An HTML legacy system is a description of the structure, content, and links of web pages. It does not have dynamic characteristics, does not need slicing, and is depicted with class, composite structure, component and deployment diagrams. An SQL legacy system defines the methods for creating and manipulating relational databases, and is described by a model that uses class, composite structure and activity diagrams.

3.2 Comparison of Proposed Approach and Traditional Studies

3.2.1 Traditional Studies

Nowadays there are many techniques for the software evolution of legacy systems and
they all have drawbacks to a greater or lesser degree [1, 18, 19, 22, 44, 46, 64, 77, 98, 110, 112].

- Formal methods have drawbacks in their practical applications to real world legacy systems.
  
a) When using automatic translators from other programming languages and presenting the original designing thoughts using formal methods, it is not clear how algorithms present in conventional languages get translated and manipulated in translators.
  
b) Those methods that apply to a program do not make the code more readable and reusable when translated back into the original code.
  
c) Reusing code for other environments or programming languages is complicated. It is not only a problem of the interface with the environment and the user, but also a problem of coupling between the subsystem and the tools.

- Using models in software evolution should integrate both formal and cognitive methods of software evolution. Using models is not only founded on theoretical work, but also based on domain knowledge.

- UML has proved to be a good platform for modelling real systems. However, in practice, it is not necessary to use all of its thirteen diagrams to model those legacy systems. Some of the UML diagrams are similar. For example, the class diagram is the most fundamental of the diagrams for modelling the structure of legacy systems. An object has the same characteristics as the corresponding class. A class is the abstraction of the common characteristics of the object group. Most of the important characteristics of an object are reflected in the corresponding class. If the class diagram is used in modelling, the object diagram is superfluous. Therefore, when using UML in the designated area, it should be edited so as to be suitable for systems modelling.

Although there are many approaches towards software evolution starting from legacy systems, nowadays they are mainly approached through program slicing techniques to want or try to acquire UML diagrams from legacy code (Figure 3.1), some of which are not able to obtain UML diagrams.
Restructuring code is intended to make legacy code cleaner and easier to be understood. In the legacy code, there may be dead code that is useless to the execution of the tasks. That dead code maybe existed at the stage of the development. Or with the change of the environment, especially with the improvement of the hardware, some ways of inputting or outputting data have been modified; or some ways of storing data have been improved. Therefore the corresponding code become useless; on some occasions it even results in the failure of the system. Consequently that dead code must be recognised and removed from the legacy code.
3.2.2 Proposed Approach

3.2.2.1 Development/Environment-Specific Models

A model is a representation or description designed to show the structure of a system. It is a study of a miniature of the actual system. A model enables IT to be more efficient at reacting to business users' requests for new systems or changes to existing ones. A model makes it possible to build an application once and use it many times.

A model creates an IT infrastructure that can react quickly to business changes, rather than it being a constraint to the business. A model gives the users immediate answers to the questions that arise. A model makes the IT more flexible and quicker to react to changes in the business.

One of the important abilities of a model is to reduce applications to their basic components, so that parts of them can be reused or dropped by various divisions of the company as needed. It shares various pieces of functionality across applications and builds a framework. It is attractive both to IT professionals and business managers. It is convenient when engineers are trying to slice and dice the functionality into discrete elements of business functions.

A model has the ability to help break down the applications supporting various products into pieces that can be saved, or discarded as duplicates, in order to integrate the users' new products with their existing products, so that there will be no overlap.

A development/environment-specific model is a representation designed to present an internal structure or a behavioural description, simplified by ignoring certain details of a domain-specific legacy system that is based on the development environment. It allows domain-specific software systems to be understood and their behaviour predicted within the scope of the development environment. It is language-based, and contains the structural and operational information of working flow or executable functions. It integrates formal methods and cognitive methods of software evolution.

The proposed approach is based on development/environment-specific models. Their differences are based on the characteristics and operations of different domain-specific legacy systems. The development/environment-specific model of a COBOL legacy
Chapter 3. Proposed Approach

system is a procedure-based model; the development/environment-specific model of an HTML legacy system is link-based; and the development/environment-specific model of an SQL legacy system is database-based.

3.2.2.2 Slicing

Based on Weiser's program slicing approach, slices are computed by computing consecutive sets of transitively relevant statements based on their data flow and control flow dependencies [131, 132]. Program slicing concerns the parts of a program that affect the values computed at some point of interest. It comprises static and dynamic slicing.

A number of hybrid applications, using both static and dynamic slicing methods, have been suggested for solutions involving the slicing of programs containing procedures, unstructured control flow, composite data types and pointers, and concurrency, in terms of accuracy and efficiency [2, 17, 76, 79, 82, 123, 126]. Program slicing is applied in debugging and program analysis, program differencing and integration, software maintenance, testing, tuning compilers, and other situations.

COBOL and SQL legacy systems involve data flow and control flow, and program slicing techniques are suitable for modelling them. However, HTML legacy systems comprise the presentation of web and data formats, and they do not focus on data flow or control flow dependencies. They present data under a designed format, especially with tables and frames, and offer a common platform for the different data. So program slicing techniques are not suitable for analysing HTML legacy systems.

3.2.2.3 Using Selected UML Diagrams

UML is regarded as the new modelling standard because most of the currently existing methods have been integrated within UML. As a theory, UML is sufficient for modelling real systems. It is the integration of many different methods comprising different aspects and different layers. It is necessary for UML to be comprehensive because it is a general modelling theory and it should have the capability of being suitable for different systems. But some of the UML diagrams are similar or even redundant in some ways and in some areas, especially in software systems.

In practice, UML has been used to deal with the given problem. It is not easy to
comprehend the systems because of their complexity, and the influence of different aspects of the systems. Different understandings are required from different views. So it is difficult to model the systems without considering the given task.

Therefore, when using UML in the designated area, it should be edited so as to be suitable for the modelling systems in question. It is not necessary to use all of its thirteen diagrams to model those legacy systems.

Consequently, in this thesis, four UML diagrams, which are the class, composite structure, sequence and interaction overview diagrams, are used to model COBOL legacy systems; four UML diagrams, which are the class, composite structure, component and deployment diagrams, are used to model HTML legacy systems; and three UML diagrams, which are the class, composite structure and activity diagrams, are used to model SQL legacy systems.

3.2.2.4 Two Major Stages of UML Extraction

There are two major stages of UML extraction from domain-specific legacy systems as being structural and behavioural described in this thesis.

The structural stage of UML extraction in this thesis contains class realisation. The classification of classes from COBOL legacy system is two, which are procedure class and variable class. Every procedure in COBOL legacy system is defined as one procedure class. Variable class is based on the program slicing techniques with two stages of pseudo class and real class extraction from COBOL legacy system. The variable of the sliced criterion is defined as the class name, and the variables contained in its slicing criterion are defined as the attributes of that variable class. Because the behavioural analysis of domain-specific legacy systems is behind the analysis of structural analysis, the operations in variabl class are not described. The classification of classes of HTML legacy system is based on the web pages and their blocks. The classification of SQL legacy system is two, which is procedure class and database class.

The behavioural stage of UML extraction in this thesis focuses on the operations and activities of domain-specific legacy systems. When understanding the operations and activities of domain-specific legacy code, their preconditions and post-conditions must be presented from the source code. Then those operations and activities are ordered
Chapter 3. Proposed Approach

according to the time and sequence they are executed by. At last, the operation and activity arrays are presented. Selected UML diagrams describing the dynamic aspect of domain-specific legacy systems are realised based on those operation and activity arrays.

3.2.2.5 Advantages of Proposed Approach

The proposed approach overcomes the disadvantages of traditional studies and gains many benefits.

The proposed approach overcomes the disadvantages of traditional studies by using development/environment-specific models. Modelling enables IT to be more efficient in reacting to business users' requests for new systems or changes to existing ones, and it is able to build applications once and use them many times. It satisfies the users by giving them immediate answers to questions as they arise. It enables IT to be nimbler and quicker in reacting to business changes, rather than having IT be a constraint on the business. Modelling has the ability to help break down the applications supporting various products into pieces that can be saved or discarded as duplicative.

The proposed approach is based on development/environment-specific models. They are different for different legacy systems. The proposed development/environment-specific model of COBOL legacy systems is based on the characteristics and operations of COBOL, and is the linear, branch, joint, and synthetic procedure-based model; the proposed development/environment-specific model of HTML legacy systems is the sequential, cyclical, and compositive link-based model; and the proposed development/environment-specific model of SQL legacy systems is based on the characteristics and operations of SQL, and is the association, generation and composition database-based model.

The proposed approach presents the differences in applying program slicing techniques between COBOL and SQL legacy systems on the one hand, and HTML legacy systems on the other. The first two types can be sliced, but not the last one. The traditional studies sliced legacy code with program slicing techniques based on Weiser's approach. However, the slices are computed by computing consecutive sets of transitively relevant statements based on data flow and control flow dependencies. COBOL and SQL legacy systems involve data flow and control flow, and program slicing techniques are suitable
for modelling them, whilst HTML legacy systems are the presentation of web and data formats, and do not focus on data flow or control flow dependencies, and program slicing techniques are not suitable for analysing HTML legacy systems.

The proposed approach does not use all of its thirteen diagrams to model domain-specific legacy systems. Traditional studies presented all the UML diagrams. Some of the UML diagrams are similar or even redundant in some ways and in some areas, especially in software systems. When using UML in the designated area, it should be edited so as to be suitable for the given modelling systems. The proposed approach uses four UML diagrams, which are the class, composite structure, sequence and interaction overview diagrams, to model COBOL legacy systems; uses four UML diagrams, which are the class, composite structure, component and deployment diagrams, to model HTML legacy systems; and three UML diagrams, which are the class, composite structure and activity diagrams, to model SQL legacy systems.

3.3 Parsing Legacy Systems

3.3.1 Parsing Original Code

In order to find the judgement conditions of the models, the first step is to parse the domain-specific legacy systems. Those domain-specific legacy systems have their own models to show their structures and operational processes.

In computer science and linguistics, parsing is the process of analysing a sequence of tokens in order to determine its grammatical structure with respect to a given formal grammar. It is sometimes termed a syntactic analysis. A parser is the component of a compiler that carries out this task.

The first stage is token generation, or lexical analysis, by which the input character stream is split into meaningful symbols defined by a grammar of regular expressions.

The next stage is parsing or syntactic analysis, which is checking that the tokens form an allowable expression. This is usually done with reference to a context-free grammar which recursively defines components that can make up an expression and the order in which they must appear.
Chapter 3. Proposed Approach

The final phase is semantic parsing or analysis, which is working out the implications of the expression just validated and taking the appropriate action.

3.3.2 Procedure-Based Models of COBOL Legacy Systems

The development/environment-specific models for COBOL legacy systems correspond to the procedure relationships in COBOL legacy systems. The procedure relationship describes the calling or being-called relationship between two procedures in COBOL legacy systems. It has four types: one to one, one to many, many to one, and many to many.

In COBOL legacy systems, all the procedure relationships are included in the four types. All the procedures call or are called in the following statements:

- "PROGRAM-ID procedure-name",
- "PERFORM procedure-name",
- "GOTO procedure-name".

By searching those statements in COBOL legacy systems through the key works "PROGRAM-ID", "PERFORM" and "GOTO", all the procedures are identified including their calling or being called relationships of those four types with removing their redundancy.

The proposed development/environment-specific model for COBOL legacy systems is based on the characteristics and operations of COBOL, and is a procedure-based model that is a graph describing the calling and being-called relationships of those procedures in COBOL legacy systems. It has four types: linear, branch, joint, and synthetic procedure-based models. The procedure-based model entity for COBOL legacy systems is a procedure.

3.3.3 Link-Based Models of HTML Legacy Systems

The development/environment-specific models for HTML legacy systems correspond to the web relationships in HTML legacy systems. The web relationship depicts the linking or being-linked relationship between two web pages in an HTML legacy system.
Chapter 3. Proposed Approach

It has two types: ordinal and loop.

In HTML legacy systems, all the web relationships are included in the two types mentioned above. The statements in HTML legacy systems that contains the web links are described as:

"<A> linking-web-name",

"<BASE> linking-web-name",

"<LINK> linking-web-name".

Based on the key words "<A>" , "<BASE>" and "<LINK>", all the candidates containing web links relationships are identified. With removing the redundancy of the webs, the webs and their linking or being linked relationships are confirmed. It composes the base of the development/environment-specific model of HTML legacy systems.

The proposed development/environment-specific model for HTML legacy systems is a link-based model based on HTML operations that are divided into five groups. It comprises a graph that describes the importing or imported relationships of those webs in HTML legacy systems. It has three types: sequential, cyclical, and compositive link-based models. The link-based model entity for HTML legacy systems is a web page.

3.3.4 Database-Based Models of SQL Legacy Systems

The relationships between the databases in SQL legacy systems are mainly of two types: generation and association.

In SQL legacy systems, all the database relationships are included in those two types mentioned above. The statements in SQL legacy systems probably containing databases are described as:

"USE database-name",

"CREATE database-name",
Chapter 3. Proposed Approach

"ALERT database-name",

"DROP database-name".

By searching the key words "USE", "CREATE", "ALERT" and "DROP", all the candidates of statements in SQL legacy systems containing the databases and their relationships are identified. After removing the redundancy of the databases, the pure databases and their relationships in SQL legacy systems are confirmed. It produces the base of the development/environment-specific model of SQL legacy system.

The proposed development/environment-specific model for SQL legacy systems is based on the characteristics and operations of SQL, and is a database-based model which is a graph that describes the database relationships in SQL legacy systems. The three database-based models for SQL legacy systems are association, generation and composition. The database-based model entity for SQL legacy systems is a database.

3.4 Suitability of Program Slicing Techniques

3.4.1 Slicing COBOL Legacy Systems

According to Weiser's introduction, slices are computed by computing consecutive sets of transitive relevant statements based on data flow and control flow dependencies.

Program slicing consists of the computed part of a program that affect the values computed at some point of interest. It comprises static and dynamic slicing. Dynamic slicing is defined as the isolation that may relate to a specific input to the program, while static slicing relates to all possible inputs.

Program slicing is helpful in determining whether a change at some point in a program will affect the behaviour of other parts of the program in software maintenance. Static slicing decomposes a program into a set of components, and shows how each of those components captures part of the original program’s behaviour, and how changes in a component can be merged back into the complete program in a semantically consistent way.

Because COBOL legacy systems involve data flow and control flow dependencies, they
3.4.2 No Need to Slice HTML Legacy Systems

The main characteristic of HTML legacy systems is the presentation of web and data formats, which is the biggest restriction and most distinctive feature when compared with other programming languages. HTML does not focus on data flow or control flow dependencies. It is the best tool for presenting data under a designed format, especially with tables and frames, and is the common platform for different data types that have few variables.

Consequently, program slicing techniques are not suitable for analysing HTML legacy systems.

3.4.3 Slicing SQL Legacy Systems

SQL is a common platform for different relational databases, giving a common base for the usage and development of databases, and providing a common interface for different databases. The description of databases is an important task of SQL.

SQL has procedures, including many executable procedures. These procedures provide the control and searching functions of the databases. SQL legacy systems have some variables and execute the tasks of judging, looping, and computing.

Therefore, program slicing techniques are suitable for analysing SQL legacy systems.

3.5 Unnecessary to Use All UML Diagrams when Modelling Legacy Systems

3.5.1 Completeness of UML

Since the creation of UML, it has been regarded as the new modelling standard because most of the currently existing methods have been integrated within UML. UML allows information to be exchanged between different tools and it replaces these methods. As a theory, UML is all that is needed for modelling real systems. It is the integration of
many different methods in different views from different layers. It is necessary for UML to be comprehensive because it is a universal modelling theory and it should have the capability of being suitable for different systems. It therefore contains a certain number of concepts which are similar or even redundant in some ways and in some areas, especially in software systems. For example, it is very hard to determine whether it is correct for one element of original code written in one programming language to be described as the OBJECT, or CLASS, or COMPONENT. A large number of definitions are presented that are not needed in the specific area but only for theoretical reasons. Therefore, when using UML in the designated area, it should be edited so as to be suitable for the modelling system in question. It is not necessary to use all of its thirteen diagrams to model those legacy systems.

### 3.5.2 Differences between Modelling Tasks

The comprehensive nature of UML as a modelling theory makes the use of UML widespread, and suitable for different modelling tasks. In practice, for a given modelling task in a given modelling area, UML is well placed to deal with the given problem. Full understanding of complex systems is difficult, and it is influenced by different aspects of a system that could be its structure, behaviour, execution process, error detection, etc. Each of those requires a different understanding from a different point of view [11]. So it is difficult to model a system without considering the given task. Therefore, when modelling legacy systems, the research in this thesis identifies the most appropriate modelling approach using UML for domain-specific legacy systems. Only some of those thirteen UML diagrams are selected to be used to model these systems.

### 3.6 Extracting UML Diagrams from COBOL Legacy Systems

#### 3.6.1 UML Class Diagram

With respect to the slicing approach of Jiang [76] and Agrawal [2], some definitions are given below concerning the original COBOL program and its criterion for slicing.
Chapter 3. Proposed Approach

The call is defined as a procedure or function call in a programming language, and is divided into four groups—root program elements, leaf program elements, node program elements, and isolated program elements. The original code contains "GO TO" lines written in the COBOL language that make the source code become spaghetti code. In order to make it necessary to slice the program, the statements including GOTO and corresponding labels are contained in the slice. The original code is composed of statements, which are classified into three groups. After acquiring all the classes of leaf program elements, the node program elements of program are analysed until all the pseudo classes of the root program element are generated.

The slicing algorithm is presented. From the information in the tables and the definitions of the programming slices within the iterating conditions of program slicing, the slice criterion of the original program is acquired. All the variables in the slice $S_c$ with respect to the slice criterion $C = (p, V)$ are composed of the relevant object set, and its kernel is the variable $V$. After checking the validity and redundancy of the pseudo classes, the real classes are acquired. The relationships between two or more classes are displayed. Finally, three layers of class diagrams of COBOL legacy code, which are the leaf class diagram, node class diagram, and root class diagram, are realised.

3.6.2 UML Composite Structure Diagram

Composite structure diagrams are composed of parts (classes with a defined role in the context of the enclosing composite), and connections (associations with limitations on potential links in the context of the enclosing composite). This diagram is used to show the hidden internal details of a class.

3.6.3 UML Sequence Diagram

Based on the program slicing theorem, every parameter of the original code $P$ is sliced and forms a pseudo object set. An algorithm is introduced to find the real object set.

It is important to identify who is going to be using the legacy code directly. This should be done from outside the legacy code that is deeply involved in the human interaction and closely related to the domain knowledge. Candidates for the actors include the people who interact with the code, the hardware that is external from the code, and the
other systems that have interaction with that code.

When understanding the operations of legacy code, the preconditions and post-conditions of each operation must be presented. Moreover, it is fundamental to order the operations of the legacy code in sequence diagrams according to the time and sequence they are executed by. The objects from which and to which the information is sent are recorded. The sequential array is the basis for the knowledge theory and is transferred into the messages which will form the new message array. A sequence diagram is described that has the vertical dimension representing time and the horizontal dimension representing object interaction.

3.6.4 UML Interaction Overview Diagram

The UML interaction overview diagram comprises three layers: leaf, node, and root. During control flow and other operations, a fork node, joint node and action node are introduced. In order to clearly describe the action, a number that is termed an action layer is used. After the basic control structure of a leaf program element is formed, its leaf interaction overview diagram is realised, then the node and root interaction overview diagrams are produced.

3.7 Extracting UML Diagrams from HTML Legacy Systems

3.7.1 UML Class Diagram

The statements in HTML code are composed of five groups: text, image, link, frame, and table. Each group contains different operations. Three different types of web link layer are distinguished, which are root web elements, leaf web elements and node program elements. A link-based model is a graph used to describe the importing or imported relationships of those webs in program P, indicated as PG. It is composed of nodes and lines. The sequence of the link-based model PG is top-to-bottom. The web that the first node represents links the webs represented by the next nodes. The sequence of the next nodes is the sequence being imported in the first web.
Chapter 3. Proposed Approach

In response to the three web layers, three class diagrams are introduced, which are leaf, node, and root class diagrams. The analysis of an HTML legacy system starts from the root program element. HTML code has five types of presentation: text, image, table, frame, and link. So the corresponding five class types are defined. Based on an algorithm, the class diagrams of HTML legacy systems are realised.

3.7.2 UML Composite Structure Diagram

The composite structure diagram is used to show hidden internal details of a class produced from an HTML legacy system. This diagram is used to describe these five classes: text, image, table, frame, and link.

3.7.3 UML Component Diagram

An HTML legacy system depicts web pages. It uses web browsers, databases, application systems, etc. The component diagram of a legacy HTML system is used to describe the HTML legacy system.

3.7.4 UML Deployment Diagram

An HTML legacy system is the presentation of web usage. It uses web browsers, such as Internet Explorer and Netscape, which are allocated in the client; databases, such as FoxPro and Oracle, which are stored in the database server; and application systems, such as search engines and email, which are deposited in the user client. The UML deployment diagram is used.

3.8 Extracting UML Diagrams from SQL Legacy Systems

3.8.1 UML Class Diagram

The Structured Query Language (SQL) comprises one of the fundamental building blocks of modern database architecture. SQL defines the methods used to create and manipulate relational databases on all major platforms. It is a full-featured relational
database management system that offers a variety of administrative tools to ease the burdens of database development, maintenance and administration.

The operations in SQL code are divided into four groups—data, procedure, control, and safety. The data group is the set of operations in SQL code that deals with the databases, tables, views, lines and columns of tables, and single elements of tables. The procedure group is the set of operations in SQL code that deals with the procedures and functions of SQL. The safety group is the set of operations that improves the safety of the database in SQL code. The control group is the set of operations that controls the data flow and checks the conditional environment.

Procedures are layered based on the calling relationships. They are classified into three groups—root, node, and leaf. The class diagrams of SQL legacy systems are divided into three—root, node, and leaf.

The databases that are used in SQL code are regarded as classes, and the procedures are defined as the node or leaf class. With the help of the procedure layers and a graph, three class diagrams are produced.

**3.8.2 UML Composite Structure Diagram**

The database class of legacy SQL code is composed of name, table, size, etc. The procedure class is composed of name, used database, created database, deleted database, changed database, etc. The composite structure diagram presents those SQL classes in detail.

**3.8.3 UML Activity Diagram**

An activity diagram shows the flow from activity to activity. It is essentially a flowchart, showing the flow of control from activity to activity. A single code operation that belongs to the group that is exchanging information is defined as a single action. The control operations that produce the judgment form the control graph.

Those actions that execute a single function are regarded as a single activity, and the SQL legacy code produces the activity array. With the help of the control graph, a leaf activity diagram is produced with the nodes that represent the activity. The node and
root activity diagrams are then realised.

3.9 Summary

In this chapter, the proposed approach towards software evolution of domain-specific legacy systems has been described.

The proposed approach establishes the development/environment-specific models. In support of the development/environment-specific models, COBOL legacy systems have four linear, branch, joint, and synthetic procedure-based models; HTML legacy systems have sequential, cyclical, and compositive link-based models; and SQL legacy systems have association, generation and composition database-based models. Program slicing techniques are suitable for modelling COBOL and SQL legacy systems, but not for HTML legacy systems. Four UML diagrams are used to model COBOL legacy systems, four UML diagrams are used to model HTML legacy systems, and three UML diagrams are used to model SQL legacy systems.
Chapter 4

COBOL

Development/Environment-Specific Models

4.1 Introduction

Before analysing COBOL legacy systems, it is necessary to present the operations and characteristics of COBOL, and especially its importance. In this chapter, these issues are described in detail. Because a COBOL legacy system has parameters and is used to control the procedures of data files and compute the data values, it has behavioural characteristics.

In order to depict the structure of a COBOL legacy system, the procedure relationship is presented, with the calling or being-called relationship between the two procedures. The model of a COBOL legacy system is a procedure-based model which is a graph that describes the calling and being-called relationships of those procedures in a COBOL legacy system. There are four types of procedure-based model: linear, branch, joint, and synthetic.

4.2 Rationale of Software Evolution of COBOL Legacy Systems

The proposed approach parses COBOL legacy systems and presents all the candidates of the statements containing the procedures in COBOL legacy system. Then it computes the model entities and their relationships from those candidates and displays them. At last, it removes the redundancy of those model entities and shows the pure model entities of COBOL legacy system. Based on the model conditions of COBOL legacy
systems (4.9) to (4.17) in Chapter 4, the model of COBOL legacy system is acquired.

In understanding static part of COBOL legacy system, the classification of classes from COBOL legacy system is two, which are procedure class and variable class. Every procedure in COBOL legacy system is defined as one procedure class. Variable class is based on the program slicing techniques with two stages of pseudo class and real class extraction from COBOL legacy system. The variable of the sliced criterion is defined as the class name, and the variables contained in its slicing criterion are defined as the attributes of that variable class. Because the behavioural analysis of domain-specific legacy systems is behind the analysis of structural analysis, the operations in variable class are not described. The classification of classes of HTML legacy system is based on the web pages and their blocks. The classification of SQL legacy system is two, which is procedure class and database class.

In understanding dynamic part of COBOL legacy system, its operations are described. When extracting the operations of COBOL legacy system, their preconditions and post-conditions are presented from the source code. Then those operations are ordered according to the time and sequence they are executed by. At last, the operation arrays are presented. Sequence and interaction overview diagrams are realised based on those operation arrays.

4.3 COBOL Legacy Systems

4.3.1 COBOL

COBOL stands for COmmon Business Oriented Language. As the expanded acronym indicates, COBOL is designed for developing business, typically file-oriented applications [68]. Its parameters include table, record, file, etc. It is not designed for writing systems programs. For instance, it would not be appropriate to develop an operating system or a compiler using COBOL.

4.3.2 A Brief History of COBOL

COBOL is a high-level programming language that has worldwide popularity. It was first developed by the CODASYL Committee (Conference on Data Systems Languages) in 1960. Since then, responsibility for developing new COBOL standards has been
assumed by the American National Standards Institute (ANSI). Three ANSI standards for COBOL have been produced: in 1968, 1974 and 1985 [69]. A new COBOL standard, introducing object-oriented programming to COBOL, is due within the next few years.

For more than four decades, COBOL has been the dominant programming language in the business computing domain. In that time it has seen off the challenges of a number of other languages, such as PL1, Algol68, Pascal, Modula, Ada, C, and C++. All these languages have found a niche but none has yet displaced COBOL [122].

### 4.3.3 Characteristics of COBOL

#### 4.3.3.1 Wide Usage

COBOL has its own special location in the programming world. COBOL's dominance is underlined by the detailed reports from the Gartner Group [42].

"In 1997 they estimated that there were about 300 billion lines of computer code in use in the world. Of that they estimated that about 80% (240 billion lines) were in COBOL and 20% (60 billion lines) were written in all the other computer languages combined."

"In 1999 they reported that over 50% of all new mission-critical applications were still being written in COBOL, and their recent estimates indicate that through 2004-2005 15% of all new applications (5 billion lines) will be developed in COBOL while 80% of all deployed applications will include extensions to existing legacy (usually COBOL) programs."

"Gartner estimates that in 2002 there were about two million COBOL programmers worldwide compared to about one million Java programmers and one million C++ programmers."

Software workers are often surprised when presented with the evidence for COBOL's dominance in the marketplace. The hype that surrounds some computer languages might persuade programmers to believe that most of the production business applications in the world are written in Java, C, C++ or Visual Basic, and that only a small percentage are written in COBOL. In fact, the reverse is actually the case.
Chapter 4. COBOL Development/Environment-Specific Models

4.3.3.2 Long History

COBOL was produced in 1960, and it is still used now with three standards—COBOL 1968, COBOL 1974, and COBOL 1985. It has been used for more than four decades. It is also used all over the world. There was much useful information stored in COBOL software in the early days.

4.3.3.3 Large Investment

COBOL applications cost many millions of dollars to produce, they are tailored to a specific company, encapsulate the business rules of that company, and only a limited number of copies of the software may be in use. A good example of this kind of application is the DoD MRP II system [42]. This system is "used to manage almost 550,000 spare and repair parts and equipment items with an inventory value of $28 billion. The system runs on Amdahl mainframes at multiple locations throughout the U.S. and contains over 4,000,000 lines of COBOL code."

In the horizontal software market, applications may still cost millions of dollars to produce but thousands, and in some cases millions, of copies of the software are in use. As a result, these applications often have a very high profile, a short life span, and a relatively low per-copy replacement cost. The Microsoft Office suite (Word, Excel, and Access) is an example of an application in the horizontal software market. Because of the highly competitive nature of the marketplace, considerations of speed, size and efficiency often make languages like C or C++ the language of choice for creating these applications.

Applications written for the vertical market, on the other hand, often have a low profile (because they are usually written for use in one particular company), a very high per-copy replacement cost, and, consequently, a very long lifespan. For example, the cost of replacing COBOL code has been estimated at approximately twenty-five dollars ($25) per line of code. At this rate, the cost of replacing the DoD MRP II system mentioned above, with a system written in some other language, would be some one hundred million dollars ($100,000,000). The importance of ease of maintenance often makes COBOL the language of choice for these applications.

The high visibility of horizontal applications like Microsoft Word or Excel persuades programmers that the languages used to write these applications are the market leaders.
But however many copies of Excel are sold, it is just a single application produced by a limited number of programmers. Many more programmers are involved in coding or maintaining one-off, "bespoke" applications. These programmers generally write their programs in COBOL.

As exemplified by the DoD MRP II example above, COBOL applications are often very large. Many COBOL applications consist of more than 1,000,000 lines of code - with 6,000,000+ line applications not considered unusually large in many organisations.

COBOL applications are widely used and long-lived. The huge investment in creating a software application consisting of some millions of lines of COBOL code means that the application cannot simply be discarded when some new programming language or technology appears. As a consequence, business applications that are between 10 and 30 years old are common. This accounts for the predominance of COBOL programs in the year 2000 problem (12,000,000 COBOL applications vs. 375,000 C and C++ applications in the US alone). Twenty years ago, when programmers were writing these applications, they just didn't anticipate that they would last into the new millennium [143].

**4.3.3.4 Location**

COBOL applications often run in critical areas of business. For instance, over 95% of finance/insurance data is processed with COBOL.

COBOL applications often deal with enormous volumes of data. Single production files and databases measured in terabytes are not uncommon.

According to research firm Gartner, there are roughly 30 billion COBOL transactions processed each day. The issue is the expense associated with running these systems.

IBM admits that at least $1.5 trillion has been spent by enterprises to create COBOL/CICS applications, and the expense associated with maintaining those applications is increasing rather than decreasing.

**4.3.3.5 Connotations**

COBOL applications contain important information in the commercial area [107]. Because of its long history and important locations, many COBOL software systems are
vital to business companies and organisations. The connotations are critical to the existence of those users.

4.3.3.6 Importance

The following COBOL facts are of importance in understanding the wide usage and long life of COBOL [125, 142].

- "75% of all business data is processed in COBOL. - Gartner Group"
- "There are between 180 billion and 200 billion lines of COBOL code in use worldwide. - Gartner Group"
- "The use of COBOL is growing by over a billion lines per year. - Gartner Group"
- "15% of all new applications (5 billion lines) through 2005 will be in COBOL. - Gartner Group"
- "CICS transaction volume (such as COBOL-based ATM transactions) grew from 20 billion per day in 1998 to 30 billion per day in 2002. - The COBOL Report"
- "Replacement costs for COBOL systems, estimated at $25 per line, are in the hundreds of billions of dollars. - Tactical Study Group"
- ""Integration with Legacies" is the number one concern of IT managers in 2003. - Gartner Group."
- ""Where are we going to get the bodies?" is the primary question from User Groups. - Microfocus International"
- "There are over 90,000 COBOL programmers in North America in 2002. Over the next four years there will be a 13% decrease in their number due to retirement and death. - Gartner Group"
- "There are at least 10,000 "Free Agent" COBOL programmers in the US today. - The Senior Staff"
- "The most highly paid programmers in the next ten years are going to be COBOL programmers who know the Internet. - GIGA Group"
- "Any programmer with above average skills in COBOL can quickly learn the basics of Web enabling, at home, through self-training. - Bill Lockhart, Legacy Reservist"
4.4 Parsing COBOL Code

4.4.1 Grouping COBOL Code Operations

The procedures in a COBOL program consist of statements, which are executable operations coded in the Procedure Division. A sentence, which is terminated by a full stop, is made up from a number of statements. Because each statement begins with a verb that is included in the reserved words list, the statements of COBOL procedures are divided into the following parts:

Group One operations change the values of variables, including record, file, table, and parameter, indicated as SS. The verbs contained in the statements in SS are indicated as VERB(SS)={ ACCEPT, DISPLAY, MOVE, INITIALISE, SET, ADD, COMPUTE, DIVISION, MULTIPLY, SUBTRACT, STRING, UNSTRING, INSPECT, INITIATE, GENERATE, TERMINATE, SUPPRESS, USE BEFORE REPORTING, IN, OF, FUNCTION, DELETE, OPEN, CLOSE, READ, WRITE, REWRITE, COPY, RELEASE, SORT, RETURN, START, USE...AFTER..., USE FOR DEBUGGING, SELECT }.

- Input-output operations, which are used to define the values of parameters and support transfers between a program and a terminal or visual display unit. They include ACCEPT and DISPLAY statements.

- Value operations, which are used to contribute values to the variables of the data items. They include MOVE, INITIALISE, and SET statements.

- Arithmetic operations, which are similar to the layman's understanding of performing addition, subtraction, multiplication, division, and compound calculations. They include ADD, COMPUTE, DIVISION, MULTIPLY, and SUBTRACT statements.

- String operations, which produce strings in different ways. They include the STRING, UNSTRING, and INSPECT statements.

- Report manipulation operations, which manipulate the report. They include the INITIATE, GENERATE, TERMINATE, SUPPRESS, and USE BEFORE REPORTING statements.
Chapter 4. COBOL Development/Environment-Specific Models

- Qualification operations, which are used to confirm the qualification of data items. They are the **IN** and **OF** statements.

- Systematic call operation, which gets functions directly from the resource of the operational environment. This is the **FUNCTION** statement.

- Record deletion operation, which deletes a record from a file. This is the **DELETE** statement.

- File handling operations, which are used to install and delete a file, and initialise and terminate the processing of a file, and result in the records being transmitted from and to the file. They include the **OPEN**, **CLOSE**, **READ**, **WRITE**, **REWRITE**, **COPY**, **RELEASE**, **SORT**, **RETURN**, **START**, **USE...AFTER...**, **USE FOR DEBUGGING** and **SELECT** statements.

Group Two comprises conditional operations, indicated as **SCD**. Here, \( \text{VERB(SCD)} = \{ \text{IF, IF...ELSE...}, \text{EVALUATE, PERFORM...UNTIL (BY)..., CONTINUE, SEARCH } \} \).

- Condition operations, which provide preconditions and direct the process of the execution of programs to different directions according to the different results of the preconditions. They include the **IF, IF...ELSE...**, **EVALUATE**, **PERFORM...UNTIL (BY)...**, and **CONTINUE** statements.

- Searching operation, which is used to search the tables for specified items within the tables. This is the **SEARCH** statement.

Group Three comprises control flow operations, indicated as **SCT**, and \( \text{VERB(SCT)} = \{ \text{CALL, GOTO, PERFORM, CANCEL, EXIT, EXIT-PROGRAM, STOP RUN } \} \).

- Procedural control operations, which are utilised to lead the program control to move to the specified paragraph name, or cause one set of statements within a paragraph or group of consecutive paragraphs to be executed continually before proceeding to the next instruction, or result in the execution of a program being stopped. They include the **CALL, GOTO** and **PERFORM** statements.

- Terminal operations, which are used to terminate the execution of the program. They are the **EXIT, CANCEL, EXIT-PROGRAM** and **STOP RUN** statements. Sometimes a **GOTO** statement shifts the control to the end.
4.4.2 Parsing COBOL Code

Parsing is the process of analysing a sequence of tokens to determine its grammatical structure with respect to a given formal grammar. A parser is an earlier term for the diagramming of sentences of programming languages. As a compiler, it parses the source code of a computer programming language to create some form of internal representation. Programming languages tend to be specified in terms of a context-free grammar because fast and efficient parsers can be written for them [23]. Parsers are usually not written by hand but are generated by parser generators [24].

COBOL code parsing is based on domain knowledge [87]. It transforms input text into a data structure, which is suitable for later processing and which captures the implied hierarchy of the input. Lexical analysis of COBOL code creates tokens from a sequence of input characters and it is these tokens that are processed by a parser to build a data structure such as a parser tree or abstract syntax trees. The transformed context-free grammars of COBOL source code are limited in the extent to which they can express all of the requirements of a language.

The normal process of parsing COBOL code is presented in Figure 4.1. COBOL source code is regarded as the raw material for parsing COBOL legacy systems. The starting point is the lexical analysis and that acquires COBOL tokens. These COBOL tokens are then syntactically analysed and the production is outputted. The output from parsing a COBOL legacy system is the procedures and their calling or being-called relationships.

Figure 4.1: Process of Parsing COBOL Code
Chapter 4. COBOL Development/Environment-Specific Models

4.5 Procedure-Based Model

A procedure-based model is a graph used to describe the calling or being-called relationships of those procedures in program P, indicated as PBM(). It is composed of nodes and lines. The sequence of a procedure-based model PBM() is top-to-bottom. The procedure that the first node represents calls the procedures that the next nodes represent.

4.5.1 Structure

4.5.1.1 Dividing Calls into Four Groups

What is meant here by a call is a procedure or function call in a programming language. The starting point in analysing the structure of the legacy system is to develop a call graph. Examination of the calling structure of the legacy system can be used to identify program elements, with minimal dependencies, that could easily be migrated. Four different types of program element are distinguished: root program elements that call other program elements but are not called by any; leaf program elements that are called by other program elements but do not call any; node program elements that both call and are called by other program elements; and isolated program elements that neither call nor are called by other program elements.

A program PPₙ calling another program PPₜ is indicated as PPₙ > > PPₜ. A program PPₙ not calling another program PPₜ is indicated as PPₙ ==:= PPₜ. A program PPₙ called by another program PPₜ is indicated as PPₙ << PPₜ. A program PPₙ not called by another program PPₜ is indicated as PPₙ <= <= PPₜ.

For program P, its procedures and its functions being PPᵢ, i≥0, let PP(P) be the procedure and function set of program P, which is indicated as PP(P)={ PPᵢ | PPᵢ << P, i≥0}. PPₙ is termed a root program element if and only if

(∃PPᵢ, PPᵢ∈PP(P) ⇒ (PPₙ > > PPᵢ))

AND
Chapter 4. COBOL Development/Environment-Specific Models

\[(\forall PP_j, PP_j \in PP(P) \Rightarrow (PP_n \preceq PP_j))\]  \( (4.1) \)

PP\(_n\) is termed a leaf program element if and only if

\[(\forall PP_i, PP_i \in PP(P) \Rightarrow (PP_n \gg PP_i))\]

AND

\[(\exists PP_j, PP_j \in PP(P) \Rightarrow (PP_n \preceq PP_j))\]  \( (4.2) \)

PP\(_n\) is termed a node program element if and only if

\[(\exists PP_i, PP_i \in PP(P) \Rightarrow (PP_n \gg PP_i))\]

AND

\[(\exists PP_j, PP_j \in PP(P) \Rightarrow (PP_n \preceq PP_j))\]  \( (4.3) \)

PP\(_n\) is termed an isolated program element if and only if

\[(\forall PP_i, PP_i \in PP(P) \Rightarrow (PP_n \gg PP_i))\]

AND

\[(\forall PP_j, PP_j \in PP(P) \Rightarrow (PP_n \preceq PP_j))\]  \( (4.4) \)

Here, \(i \neq n, j \neq n\).

In order to understand the source code as a whole, it is necessary to describe the calling or being-called relationships of those procedures in program P.

4.5.1.2 Procedure Layer

The procedure layer of procedure P is a number that represents the depth of one procedure calling other procedures, indicated as PL(P).

The procedure layer of leaf program elements is 0, the procedure layer of the program elements that only call leaf program elements is 1, the procedure layer of program elements that call other program elements, the maximum of whose procedure layers is 1, is 2, etc.

Let PP\(_1\), PP\(_2\), and PP\(_3\) be three procedures, and assume that PP\(_1\) is a leaf program element,

\[(PP_2 \gg PP_1) \text{ AND } (PP_2 \gg \gg PP_3),\]

\[(PP_3 \gg PP_1) \text{ AND } (PP_3 \gg PP_2) \text{ AND } (PP_1 \gg \gg PP_3),\]

then

\[\text{PL}(PP_1) = 0,\]

63
Chapter 4. COBOL Development/Environment-Specific Models

\[ PL(PP_2)=1, \]
\[ PL(PP_3)=2. \]

The root procedure layer is changeable although the leaf procedure layer is always 0.

4.5.1.3 COBOL Model Entities and Their Relationships

Every procedure in a COBOL legacy system is regarded as a single and essential unit of a procedure-based model of a COBOL legacy system, and the procedure in a COBOL legacy system is termed a COBOL model entity.

The procedure relationship is defined as the calling or being-called relationship between the two procedures in a procedure-based model. There are four types of calling relationship: one to one, one to many, many to one, and many to many. Any one of the procedure relationships in COBOL legacy system is contained in one of these four types.

Let PP_i, PP_j, PP_k be three procedures in a legacy system. If

\[ (PP_i \gg PP_j) \]
\[ AND \]
\[ (((\forall k, k \neq j) \Rightarrow (PP_i \gg PP_k)) \]
\[ AND ((\forall k, k \neq i) \Rightarrow (PP_k \gg PP_j)) \]  

then the relationship between PP_i and PP_j is termed one to one (Figure 4.2).

If

\[ (PP_i \gg PP_j) \]
\[ AND ((\exists k, k \neq j) \Rightarrow (PP_i \gg PP_k)) \]
\[ AND ((\forall m, m \neq i) \Rightarrow (PP_m \gg PP_j)) \]
\[ AND ((\forall m, m \neq j) \Rightarrow (PP_m \gg PP_k)) \]

then the relationship between PP_i and PP_j is termed one to many (Figure 4.3).

If

\[ (PP_i \gg PP_j) \]
\[ AND ((\exists k, k \neq i) \Rightarrow (PP_k \gg PP_j)) \]
\[ AND ((\forall m, m \neq i) \Rightarrow (PP_i \gg PP_m)) \]
\[ AND ((\forall m, m \neq j) \Rightarrow (PP_k \gg PP_m)) \]

then the relationship between PP_i and PP_j is termed many to one (Figure 4.4).
If
\[(PP_i \gg PP_j) \quad \text{AND} \quad ((\forall k, k \neq j) \Rightarrow (PP_i \gg PP_k)) \quad \text{AND} \quad ((\exists m, m \neq i) \Rightarrow (PP_m \gg PP_j))\] (4.8)
then the relationship between PP_i and PP_j is termed many to many (Figure 4.5).

Figure 4.2 One to One Relationship

Figure 4.3 One to Many Relationship

Figure 4.4 Many to One Relationship
4.5.2 Classification of A Procedure-Based Model

COBOL Model 1: Linear procedure-based model

A linear procedure-based model is a procedure-based model in which the relationships between a root program element and a node program element, between node program elements, or between a node program element and a leaf program element, are one to one (Figure 4.6).

Let $PP_i$, $PP_j$, $PP_k$ be three procedures in COBOL legacy system $P$, and let the set $SAP\{}$ be the set of all procedures in $P$ ($SAP=$Set of All Procedures) as indicated

$$SAP\{} = \{ PP_i \mid PP_i \in P \}, \quad (4.9)$$

and let the set $SOOP\{}$ be the set of the one to one procedures in $P$ ($SOOP=$Set of One to One Procedures) as indicated

$$SOOP\{} = \{(PP_i, PP_j) \mid (PP_i \Rightarrow PP_j, PP_j \Rightarrow PP_i) \}, \quad (4.10)$$

If

$$SOOP\{} = SAP\{} , \quad (4.11)$$

then the procedure-based model of the COBOL legacy system is linear.
COBOL Model 2: Branch procedure-based model

A branch procedure-based model is a procedure-based model in which one or more relationships between root program and node program elements, between node program elements, or between node program and leaf program elements, are one to many, and other relationships are one to one (Figure 4.7).

Let the set $SOMP \{ \}$ be the set of the one to many procedures in $P$ ($SOMP=$Set of One to Many Procedures) as indicated

$$SOMP\{ \} = \{(PP_i, PP_j) | (PP_i \gg PP_j) \}
\quad \text{AND} \quad ((\exists k, k \neq j) \Rightarrow (PP_i \gg PP_k))$$

$$\quad \text{AND} \quad ((\forall m, m \neq i) \Rightarrow (PP_m \gg PP_j))$$

$$\quad \text{AND} \quad ((\forall m, m \neq i) \Rightarrow (PP_m \gg PP_k)). \quad (4.12)$$

If

$$SOMP\{ \} = SAP\{ \}. \quad (4.13)$$

then the procedure-based model of COBOL legacy system is branch.

COBOL Model 3: Joint procedure-based model

A joint procedure-based model is a procedure-based model in which one or more relationships between node program elements, or between node program elements and one leaf program element, are many to one, and other relationships are one to one (Figure 4.8).
Chapter 4. COBOL Development/Environment-Specific Models

Let the set SMOP\{\} be the set of the many to one procedures in COBOL legacy system P except for the root program element PP_0 (SMOP=Set of Many to One Procedures), then

$$SMOP\{\} = \{ (PP_i, PP_j) | (PP_i \gg PP_j)$$

AND $$((\exists k, k \neq i) \Rightarrow (PP_k \gg PP_j))$$

AND $$((\forall m, m \neq j) \Rightarrow (PP_i \gg PP_m))$$

AND $$((\forall m, m \neq j) \Rightarrow (PP_k \gg PP_m)).$$ \hspace{1cm}(4.14)

If

$$SMOP\{\} = SAP\{} - \{PP_0\},$$ \hspace{1cm}(4.15)

then the procedure-based model of the COBOL legacy system is joint.

![Figure 4.7: Branch Procedure-Based Model](image-url)
Figure 4.8: Joint Procedure-Based Model

Figure 4.9: Synthetic Procedure-Based Model
Chapter 4. COBOL Development/Environment-Specific Models

COBOL Model 4: Synthetic procedure-based model

A synthetic procedure-based model is a procedure-based model in which the relationships between root program and node program elements, between node program elements, or between node program and leaf program elements, include one to many and many to one (Figure 4.9). Sometimes the relationships in a synthetic procedure-based model may contain many to many.

Let the set SSP{} be the set containing the many to many procedures in P (SSP=Set of Synthetic Procedures),

\[
SSP\{}\{ PP_k \mid (\exists I, k, i \neq j, j \neq k, i \neq k) \Rightarrow ((PP_i \rightarrow PP_j) \text{ AND } (PP_i \rightarrow PP_k)) \text{ AND } ((\exists r, s, t, r \neq s, s \neq k, r \neq k) \Rightarrow ((PP_r \rightarrow PP_s) \text{ AND } (PP_r \rightarrow PP_k))) \}
\] (4.16)

If

\[
SSP\{} \neq \emptyset
\] (4.17)

then the procedure-based model of the COBOL legacy system is synthetic.

4.6 Summary

In this chapter, COBOL characteristics have been introduced. COBOL is widely used and has a long history. COBOL legacy systems have characteristics of wide usage, long history, large investment, location, connotation and importance.

The procedure relationship describes the calling or being-called relationship between the two procedures in a procedure-based model PCM( ). It has four types: one to one, one to many, many to one, and many to many.

The model of a COBOL legacy system is based on the characteristics and operations of COBOL, and the procedure-based model is a graph that describes the calling and being-called relationships of those procedures in the COBOL legacy system. It has four types: linear, branch, joint, and synthetic.
Chapter 5

HTML Development/Environment-Specific Models

5.1 Introduction

Because of the numerous users of the Internet, HTML is fundamentally important to information exchange and data presentation. It is necessary to analyse the useful contents of HTML legacy systems in order to maintain and reuse them.

In this chapter, the history and operations of HTML are briefly introduced. HTML characteristics are also presented.

In order to better describe HTML legacy systems, three link-based models are presented. They describe linking or being-linked relationships between two or more web pages.

5.2 Rationale of Software Evolution of HTML Legacy Systems

The proposed approach described in this thesis parses HTML legacy systems and presents all the candidates of the statements containing the web links in HTML legacy system. It computes the model entities and their relationships from those candidates and displays them. It removes the redundancy of those model entities and shows the pure model entities of HTML legacy system. Based on the model conditions of HTML legacy systems (5.3) to (5.8) in Chapter 5, the model of HTML legacy system is acquired.

In understanding HTML legacy system, the classification of classes from HTML legacy
system is two, which are web class and block class. Every web page in HTML legacy system is defined as one class in order to facilitate the link understanding between those web pages by leaf web class and node web class. Every web contains several blocks, and each block has five types of classes: text, image, table, frame, and link class. HTML legacy system is described with class diagram, composite structure diagram, component diagram and deployment diagram.

5.3 HTML Legacy Systems

5.3.1 HTML

HTML, which stands for Hyper Text Markup Language, is a computer language that has been devised to create websites which can then be viewed by means of a connection to the Internet. It is constantly undergoing revision and evolution to meet the demands and requirements of the growing Internet. Hyper Text is the method by which the next web page is linked to, and through which any place on the Internet can be reached. The Markup is a certain type of text to be marked. HTML is a programming language and it has code and syntax like other programming languages. HTML consists of a series of short codes contained in a text file. Those text words are saved as an “html” file and operated through special software which is termed a Browser, as in Internet Explorer or Netscape Navigator. Those browsers read that text file and transfer those text words into the visible style which is termed a Web Page. Those text words are termed Tags which are located between < > in order to separate them from ordinary text. The desired web page is realised through correct use of the tags by means of translation of the web browsers. Tables, forms, images, and sounds can then appear on the web pages in the designed format [33].

5.3.2 A Brief History of HTML

HTML was originally developed by Tim Berners-Lee while at CERN, and popularised by the Mosaic browser developed at NCSA. During the course of the 1990s, it has blossomed with the explosive growth of the Web. During this time, HTML has been extended in a number of ways. The Web depends on Web page authors and vendors sharing the same conventions for HTML. This has motivated joint work on specifications for HTML.
Chapter 5. HTML Development/Environment-Specific Models

HTML 2.0 was developed under the aegis of the Internet Engineering Task Force (IETF) to codify common practice in late 1994. HTML+ (1993) and HTML 3.0 proposed much richer versions of HTML. Despite never receiving consensus in standards discussions, these drafts led to the adoption of a range of new features. The efforts of the World Wide Web Consortium's HTML Working Group to codify common practice resulted, in 1996, in HTML 3.2.

Most people agree that HTML documents should work well across different browsers and platforms. Achieving interoperability lowers costs to content providers since they need develop only one version of a document. If the effort is not made, there is much greater risk that the Web will devolve into a proprietary world of incompatible formats, ultimately reducing the Web's commercial potential for all participants.

Each version of HTML has attempted to reflect greater consensus among industry players so that the investment made by content providers will not be wasted and their documents will not become unreadable within a short period of time.

HTML has been developed with the vision that all manner of devices should be able to use information on the Web: PCs with graphics displays of varying resolutions and colour depths, cellular telephones, hand held devices, devices for output and input of speech, computers with high or low bandwidth, and so on.

5.3.3 Characteristics of HTML

HTML uses cascading stylesheets to control the presentation of web pages, and provide basic special effects and interaction of information through the Internet.

Although the functions of HTML have been greatly improved with the help of other languages and methods, HTML is still confined to Internet usage without replacing or being replaced by other programming languages. It has its own characteristics that are different from those of other programming languages [40].

5.3.3.1 Presentation of Web Pages

HTML is designed for the presentation of web pages. When it was born, programming languages such as BASIC, COBOL, FORTRAN, PASCAL, C, etc, were mature and widely used in the real world.
It was not necessary for the designer of HTML to create a programming language to substitute the functions of those languages, and in fact HTML does not take on the responsibilities of calculation, real-time control, judgments and immediate response, batch disposal, database data management, man-made intelligence, etc. The essential task of HTML is the service of the Internet, and this is its biggest restriction and difference from other programming languages.

5.3.3.2 Data Format

HTML is an excellent tool for presenting data within the designed format, especially data with tables and frames. It is easy to present different data in different areas, even when it is in very different forms. HTML is a common platform for varied data.

5.3.3.3 Basic in Usage

One of the biggest advantages of HTML is that it is basic. The nature of HTML means that it is the basis for web applications. It does not focus on complicated control or computation. It is the foundation for the Internet. This makes HTML source code essential for the development of the Internet.

5.4 Parsing HTML Code

5.4.1 Grouping HTML Code Operations

The statements in HTML code are composed of five groups: text, image, link, frame and table. Each group contains different operations that are realised by the executions of some statements combination.

Group One covers description of text, indicated as GTEXT. The operations of program PP in GTEXT are \(\text{GTEXT}(\text{PP}) = \{<\text{H}>, <\text{META}>, <\text{APPLET}>, <\text{SCRIPT}>, <\text{NOS scripted}>, <\text{P}>, <\text{BR}>, <\text{BLOCKQUOTE}>, <!-->, <\text{PRE}>, <\text{SPAN}>, <\text{DIV}>, <\text{STYLE}>, <\text{B}>, <\text{BIG}>, <\text{SMALL}>, <\text{EM}>, <\text{I}>, <\text{STRONG}>, <\text{BASEFONT}>, <\text{FONT}>, <\text{SUB}>, <\text{SUP}>, <\text{TT}>, <\text{U}>, <\text{DEL}>, <\text{HR}>, <\text{INS}>, <\text{STRIKE}>, <\text{CODE}>, <\text{MARQUEE}>, <\text{Q}>, <\text{SAMP}>, <\text{KBD}>\}.

- Interpretative operations are used to interpret the structure of the program PP. They include \( <\text{H}>, <\text{META}>, <\text{APPLET}>, <\text{SCRIPT}>, <\text{NOScript}>, <\text{P}>, <\text{BR}>, \)
Chapter 5. HTML Development/Environment-Specific Models

- **Word Style operations** are the display descriptions of letters or words. They include `<B>`, `<BIG>`, `<SMALL>`, `<EM>`, `<I>`, `<STRONG>`, `<BASEFONT>`, `<FONT>`, `<SUB>`, `<SUP>`, `<TT>`, `<U>` statements.

- **Line operations** insert and delete lines in the text. They include `<DEL>`, `<HR>`, `<INS>`, `<STRIKE>` statements.

- **Special operations** describe special displays of text. They include `<CODE>`, `<MARQUEE>`, `<Q>`, `<SAMP>`, `<KBD>`, `<CENTER>` statements.

Group Two covers description of images, indicated as GIMAGE. The operation set of program PP written in HTML is GIMAGE(PP)={ `<IMG>`, `<MAP>`, `<EMBED>`, `<NOEMBED>`, `<AREA>`, `<LAYER>`, `<SPACER>` }.

- **Image presentation** is used to present the images in HTML code. Operations include `<IMG>`, `<MAP>`, `<EMBED>`, `<NOEMBED>`, `<AREA>`.

- **Image location** is used to locate images on web pages. Operations include `<LAYER>`, `<SPACER>`.

Group Three covers description of link operations, indicated as GLINK. The operation set of arbitrary program PP written in HTML is GLINK(PP)={ `<A>`, `<BASE>`, `<LINK>` }.

- **Link creation** is used to create links and link windows. Operations include `<A>`, `<BASE>`.

- **External link** is used to set up external links. The operation is `<LINK>`.

Group Four covers description of frame operations, indicated as GFRAME. The operation set of any program PP written in HTML is GFRAME(PP)={ `<FRAME>`, `<FRAMESET>`, `<IFRAME>` }.

- **Frame creation** creates the frames or the frame set in HTML code. Operations are `<FRAME>`, `<FRAMESET>`.

- **Embedded frame creation** is used to create embedded frames. The operation is `<IFRAME>`.

Group Five covers description of table operations, indicated as GTABLE. The operation
set is GTABLE(PP)={ <TABLE>, <TR>, <TH>, <TD>, <THEAD>, <TBODY>,
<TFOOT>, <COL>, <COLGROUP>, <CAPTION>, <FORM>, <INPUT>,
<BUTTON>, <FIELDSET>, <LEGEND>, <SELECT>, <OPTION>, <OPTGROUP>,
<LABEL>, <TEXTAREA>, <UL>, <DL>, <DT>, <DD>, <LI> }.

- Common table operations are used to create and format the common table. They include <TABLE>, <TR>, <TH>, <TD>, <THEAD>, <TBODY>, <TFOOT>, <COL>, <COLGROUP>, <CAPTION> operations.

- Form table operations are used to present and format the form table. They include <FORM>, <INPUT>, <BUTTON>, <FIELDSET>, <LEGEND>, <SELECT>, <OPTION>, <OPTGROUP>, <LABEL>, <TEXTAREA> operations.

- The sequential table operation is used to represent the sequential table. It is <OL>.

- The unsequential table operations is used to describe the unsequential table. It is <UL>.

- Table definition and creation operations are used to create tables. They are <DL>, <DT>, <DD>, <LI> operations.

### 5.4.2 Parsing HTML Code

HTML legacy systems have their own models to show their structures and operation processes. Normally, parsing HTML legacy systems involves two levels of grammar analysis: lexical and syntactic (Figure 5.1).
The first step is to input the HTML source code and acquire HTML tokens through lexical analysis. From the mathematical point of view, HTML source code is the set of data having the format of tables, images, etc. It is relocated in a coordinate system whereby the X-axis is the format of web pages and the Y-axis is the contents of web pages.

The second step is syntactic analysis in which these HTML tokens are processed to build a data structure such as parser trees or abstract syntax trees. It outputs web pages and their linking or being-linked relationships.

### 5.5 Link-Based Models

A link-based model is a graph that describes the importing or imported relationships of those webs in program P, indicated as LBM(). It is composed of nodes and lines. Every node represents a web page, and a line represents a link relationship. The web that the first node represents imports the web pages that the next nodes represent. One example of a link-based model is presented in Figure 5.2.

![Figure 5.2: Link-Based Model](image)

#### 5.5.1 Structure

##### 5.5.1.1 Three Kinds of HTML Link Element

An HTML link imports new web pages written in HTML programming language. The starting point in analysing the structure of a legacy HTML system is developing a
link-based model. Examining the link structure of the legacy HTML system can be used to identify web threads with minimal dependencies that could be migrated easily [108]. Three different kinds of web link layer are distinguished: root web elements that lead to other web element links but are not quoted by any other, or is the home web page or the first web page although linked to other web pages; leaf web elements that are quoted by other web elements but do not create any other web pages, or are the last web pages although they quote other web pages; and node web elements that both lead to and are quoted by other web elements except for root web and leaf web elements.

In order to understand the source code as a whole, it is necessary to describe the importing or imported relationships of those webs in program P.

5.5.1.2 Web Layer

The web layer is a number that represents the depth of a web page that is importing other webs, indicated as WL( ). The web layer of a root web element is 0, the web layer of the node web elements that are only imported by that root web element is 1, the web layer of web elements that are only imported by those node web elements, the maximum of whose web layers is 1, is 2, etc.

Let PP₁, PP₂, and PP₃ be three web elements, and assume that PP₁ is a root web element,

(PP₁ ⇒ PP₂) AND (PP₁⇒⇒PP₁), i≥3,
(PP₂ ⇒ PP₃) AND (PP₂⇒⇒PP₂), i>3,
(PP₃⇒⇒PP₁), i!\neq3,

then

WL(PP₁)=0,
WL(PP₂)=1,
WL(PP₃)=2 (Figure 5.3).

Figure 5.3: A Graphical Example of a Web Layer
Chapter 5. HTML Development/Environment-Specific Models

5.5.1.3 Web Relationships

In a link-based model, a web page is regarded as the essential unit of an HTML legacy system, and is termed an HTML model entity.

There are two kinds of linking relationship between HTML model entities: ordinal and loop. The web relationship is defined as the linking or being-linked relationship between two web pages in an HTML legacy system [108]. It has two kinds: ordinal and loop.

In a link-based model LBM( ), PP₁ is termed the dominator of PP₅ if there exists a path in LBM( ) from PP₅ to PP₁, indicated as PP₁ ∈ {DOM(PP₅)}. PP₁ is termed the immediate dominator of PP₅ if PP₁ is the immediate successor of PP₅, indicated as PP₁ = IM(PP₅).

Figure 5.4: First Example of an Ordinal Relationship

Figure 5.5: Second Example of an Ordinal Relationship

Figure 5.6: Third Example of an Ordinal Relationship
Let $PP_i$, $PP_j$, $PP_k$ be three web pages in an HTML legacy system. If

$$\forall k, k \neq i, PP_k \in \{DOM(PP_i)\} \Rightarrow PP_i \notin \{DOM(PP_k)\}$$ (5.1)

then the relationship between $PP_i$ and $PP_j$ is termed ordinal (Figure 5.4, 5.5, 5.6 and 5.7).

If

$$(PP_i=IM(PP_j) \text{ AND } PP_j=IM(PP_i))$$ (5.2)

then the relationship between $PP_i$ and $PP_j$ is termed loop (Figure 5.8).

### 5.5.2 Classification of Link-Based Models

**HTML Model 1: Sequential link-based model**

A sequential link-based model is a link-based model in which the relationships between root web and node web elements, between node web elements, or between node web elements and leaf web elements, are ordinal (Figure 5.9, 5.10, 5.11 and 5.12).
Chapter 5. HTML Development/Environment-Specific Models

Figure 5.9: An Example of a Sequential Link-Based Model

Figure 5.10: Second Example of a Sequential Link-Based Model

Figure 5.11: Third Example of a Sequential Link-Based Model
Chapter 5. HTML Development/Environment-Specific Models

Let \( PP_0 \) be a root web element in HTML legacy system \( P \), let \( PP_i, PP_j, PP_k \) be three web elements, let the set \( SAW{} \) be the set of all web elements in \( P \) (\( SAW{}=\text{Set of All Webs} \)), so that \( SAW{}={ PP_i | PP_i \in P } \), and let the set \( SOW{} \) be the set of the web elements whose relationships are ordinal in \( P \) (\( SOW{}=\text{Set of Ordinal Webs} \)), so that

\[
SOW{}={ PP_k | (\forall k, k \neq 0, PP_k \in \{\text{DOM}(PP_0)\}) \Rightarrow (PP_0 \notin \{\text{DOM}(PP_k)\}) }. \tag{5.3}
\]

If

\[
SOW{} = SAW{} - \{PP_0\}, \tag{5.4}
\]

then the link-based model of the HTML legacy system is sequential.

Figure 5.12: Fourth Example of a Sequential Link-Based Model

HTML Model 2: Cyclical link-based model

A cyclical link-based model is a link-based model in which one or more relationships between root web and node web elements, between node web elements, or between node web elements and leaf web elements, are looped. There is no ordinal relationship in this link-based model (Figure 5.13 and 5.14).

Let the set \( SCW{} \) be the set of the web elements whose relationships are cyclical in \( P \) (\( SCW{}=\text{Set of Cyclical Webs} \)), so that

\[
SCW{}={ PP_j | ((\forall k, k \neq j) \text{ AND } (PP_k \in SCW{})) \Rightarrow ( (PP_k \in \{\text{IM}(PP_j)\}) \text{ AND } (PP_j \in \{\text{IM}(PP_k)\}) )}, \tag{5.5}
\]

If

\[
SCW{} = SAW{}, \tag{5.6}
\]

then the link-based model of the HTML legacy system is cyclical.
then the link-based model of the HTML legacy system is cyclical.

**Figure 5.13: An Example of a Cyclical Link-Based Model**

![Diagram](image1)

**Figure 5.14: Another Example of a Cyclical Link-Based Model**

![Diagram](image2)

**HTML Model 3: Compositive link-based model**

A compositive link-based model is a link-based model in which there exist two relationships, loop and ordinal, in the same link-based model, between root and node web elements, between node web elements, or between node web elements and leaf program elements (Figure 5.15, 5.16 and 5.17).

**Figure 5.15: An Example of a Compositive Link-Based Model**

![Diagram](image3)

Let the set \( SPW{} \) be the set of web elements whose relationships contain ordinal and cyclical kinds in the same link-based model in \( P \) (\( SPW={} \text{Set of comPositive Webs} \)), so that

\[
SPW{}={} \{ PP_{k} \mid ((PP_{k} \in \{ \text{DOM}(PP_{0}) \}) \Rightarrow (PP_{0} \notin \{ \text{DOM}(PP_{k}) \})) \}
\]
Chapter 5. HTML Development/Environment-Specific Models

\[
\text{AND}((\exists m, m \neq k) \Rightarrow (\text{PP}_m \in \{\text{IM}(\text{PP}_k)\} \text{ AND} (\text{PP}_k \in \{\text{IM}(\text{PP}_m)\})), \quad (5.7)
\]

If

\[
\text{SPW}\{\} \neq \emptyset, \quad (5.8)
\]

then the link-based model of the HTML legacy system is compositive.

![Diagram of a second example of a compositive link-based model](image)

Figure 5.16: Second Example of a Compositive Link-Based Model

![Diagram of a third example of a compositive link-based model](image)

Figure 5.17: Third Example of a Compositive Link-Based Model
5.6 Summary

HTML legacy systems have distinctive characteristics of presentation of web and data formats, are incapable of scaling web pages, and are basic in usage. In a link-based model, a web page is regarded as an HTML model entity. There are two kinds of linking relationship between HTML model entities: ordinal and loop. HTML operations were divided into five groups, then the link-based model was introduced. The link-based model is a graph describing the importing or imported relationships of those webs. There are three kinds of link-based model: sequential, cyclical, and compositive.
Chapter 6
SQL Development/Environment-Specific Models

6.1 Introduction

Because of SQL’s flexible interface for databases of all shapes and sizes, it presents a full-featured relational database management system that offers a variety of administrative tools to provide benefits in the areas of database development, maintenance and administration.

In this chapter, SQL’s history, operations and characteristics are presented. Based on the relationships of generation and association between databases in SQL legacy systems, three database-based models are built up.

6.2 Rationale of Software Evolution of SQL Legacy Systems

The proposed approach parses SQL legacy systems and presents all the candidates of the statements containing the databases in SQL legacy system. Then it computes the model entities and their relationships from those candidates and displays them. It removes the redundancy of those model entities and shows the pure model entities of SQL legacy system. Based on the model conditions of SQL legacy systems (6.1) to (6.7) in Chapter 6, the model of SQL legacy system is acquired.

In understanding static part of SQL legacy system, the classification of classes from SQL legacy system is two, which are procedure class and database class. Every procedure in SQL legacy system is defined as one procedure class. Every database is defined as one class as well. The static part of SQL legacy system is described with
Chapter 6. SQL Development/Environment-Specific Models

class diagram and composite structure diagram.

In understanding dynamic part of SQL legacy system, its activities are described. Selected UML diagrams describing the dynamic aspect of domain-specific legacy systems are realised based on those activity arraies.

6.3 SQL Legacy Systems

6.3.1 SQL

The Structured Query Language (SQL) comprises one of the fundamental building blocks of modern database architecture. SQL defines the methods used to create and manipulate relational databases on all major platforms, and it is a full-featured relational database management system that offers a variety of administrative tools to ease the burdens of database development, maintenance and administration. It includes additional functionality designed to support Microsoft SQL Server.

It is common for large-scale databases to use SQL to facilitate database users and administrators.

The language offers a flexible interface for databases of all shapes and sizes. Most popular databases, including ORACLE, are searched or changed by SQL.

6.3.2 Characteristics of SQL

6.3.2.1 Common Platform

SQL is a common platform for different relational databases. Oracle databases utilise their proprietary PL/SQL. Microsoft SQL Server makes use of Transact-SQL. All of these variations are based upon the industry standard ANSI SQL. It provides a common base for the use and development of databases.

6.3.2.2 Executable Procedures

Microsoft SQL Server 2000 provides many executable procedures. They can be used easily in programming because of their well-formed pre-compilation. SQL compiles each executable procedure once and then reutilises the execution plan. This mechanism simplifies the database development process by grouping those SQL statements that
Chapter 6. SQL Development/Environment-Specific Models

represent executable procedures into manageable blocks.

6.3.2.3 Reduction of Transformation

In the information transformation of SQL legacy systems, network bandwidth is important. Stored procedures can reduce long SQL queries to a single line that is transmitted online. This results in reduced transformation.

6.3.2.4 Reuse of Parts of a System

SQL sets up a platform to exchange information in an efficient way. The data are grouped together, based on tables or databases. Having the same characteristics they are dealt with in the same way. This method can be used by multiple users and client programs.

6.3.2.5 Enhanced Security Strategy

The database is regarded as an element and the basic unit of data storage, creation, deletion, and exchange. This enhances the security strategy of data usage.

6.3.2.6 Index Benefit

SQL supports two types of index. One type is the clustered index that defines the physical sorting of a database table's rows in the storage media, and the other is the non-clustered index that is created outside of the database table and contains a sorted list of references to the table itself. Indexes provide flexible searching on database tables, speed the query performance on commonly used columns, and improve the overall processing quality of the database. Convenient and rapid searching is one of the most important characteristics of databases.

6.4 Parsing SQL Code

6.4.1 Grouping SQL Code Operations

The statements in SQL code represent the operations of data flow. The operations in SQL code are divided into four groups—data, procedure, control, and safety.

The data group is the set of operations in SQL code that deal with databases, tables,
views, lines and columns of tables, and single elements of tables. The data group of SQL code TP is indicated as GroupD(TP), GroupD(TP)={CREATE DATABASE, ALERT DATABASE, DROP DATABASE, CREATE TABLE, ALERT TABLE, DROP TABLE, INSERT, UPDATE...SET..., DELETE, CREATION VIEW, ALERT VIEW, DROP VIEW. CREATE...INDEX..., EXEC SP_HELPINDEX, DROP INDEX, SELECT, CREATE DEFAULT, EXEC SP BINDEFAULT, EXEC SP UNBINDEFAULT, DROP DEFAULT, CREATE RULE, EXEC SP BINDRULE, EXEC SP UNBINDRULE, DROP RULE, DECLARE CURSOR, OPEN, FETCH, CLOSE, DEALLOCATE}.

- Database operations. Database operations, which are responsible for the creation, change and deletion of the database, include CREATE DATABASE, ALERT DATABASE, and DROP DATABASE.

- Table operations. Table operations, which cope with the creation, change and deletion of the tables of the database, include CREATE TABLE, ALERT TABLE, and DROP TABLE.

- Table data operations. Table data operations execute the insertion, update and deletion of the data in the tables of the database. They include INSERT, UPDATE...SET..., and DELETE.

- View operations. View operations are used to deal with the creation, change and deletion of the views. They include CREATION VIEW, ALERT VIEW, and DROP VIEW.

- Index operations. Index operations are used to execute the creation, check and deletion of the index of tables in a database. They include CREATE...INDEX..., EXEC SP_HELPINDEX, and DROP INDEX.

- Selection operations. Selection operations are the most powerful in the usage of SQL server 2000 code, and are used to filter the designated data from the tables or the database. They include SELECT.

- Default operations. Default operations are used to deal with the creation, binding, change and deletion of the defaults. They include CREATE DEFAULT, EXEC SP BINDEFAULT, EXEC SP UNBINDEFAULT, and DROP DEFAULT.

- Rule operations. Rule operations are used to create, bind, and delete the rules in
Chapter 6. SQL Development/Environment-Specific Models

SQL code. They include CREATE RULE, EXEC SP_BINDRULE, EXEC SP_UNBINDRULE, and DROP RULE.

- Cursor operations. Cursor operations are used to create, open and delete the cursors in SQL code. They include DECLARE CURSOR, OPEN, FETCH, CLOSE, and DEALLOCATE.

The procedure group is the set of operations in SQL code that deals with the procedures and functions of SQL, which is indicated as GroupP(). The procedure group is composed of the operations GroupP()={CREATE PROCEDURE, EXEC procedure-name, ALERT PROCEDURE, DROP PROCEDURE, CREATE TRIGGER, EXEC SP_HELPTRIGGER, ALERT TRIGGER, DROP TRIGGER, S-FUNCTIONS}.

- Stored procedure operations. Stored procedure operations are used to create, execute and delete the stored procedures in SQL code. They include CREATE PROCEDURE, EXEC procedure-name, ALERT PROCEDURE, and DROP PROCEDURE.

- Trigger operations. Trigger operations are used to cope with the creation, execution and deletion of the triggers in the code. They include CREATE TRIGGER, EXEC SP_HELPTRIGGER, ALERT TRIGGER, and DROP TRIGGER.

- Systematic functions. Systematic functions are provided by Microsoft SQL Server 2000 and present the common operations in normal definitions, especially in the mathematical area. They are indicated as S-FUNCTIONS.

The safety group is the set of operations aimed at improving the safety of the database in SQL code. It is indicated as GroupS(), GroupS()={EXEC SP_ADDLOGIN, EXEC SP_HELPLOGONS, EXEC SP_DROPOLOGON, EXEC SP_GRANTDBACCESS, EXEC SP_HELUSER, EXEC SP_REVOKEDBACCESS, EXEC SP_ADDSRVROLEMEMBER, EXEC SP_DROPSRVROLEMEMBER, EXEC SP_ADDROLEMEMBER, EXEC SP_DROPROLEMEMBER, EXEC SP_ADDROLE, GRANT, DENY, REVOKE, EXEC SP_ADDUMPDEVICE, BACKUP DATABASE, RESTORE DATABASE}.

- Logon operations. Logon operations are used to create, check and delete the logon information, and include EXEC SP_ADDLOGIN, EXEC SP_HELPLOGONS, and EXEC SP_DROPOLOGON.
Chapter 6. SQL Development/Environment-Specific Models

- User account operations. User account operations are used to create, check and delete the user accounts of the database. They include EXEC SP_GRANTDBACCESS, EXEC SP_HELPUSER, and EXEC SP_REVOKEDBACCESS.

- Actor change operations. Actor change operations are used to add and delete actors from user account actor to server actor or database actor. They include EXEC SP_ADDSRVROLEMEMBER, EXEC SP_DROPSRVROLEMEMBER, EXEC SP_ADDROLEMEMBER, EXEC SP_DROPROLEMEMBER, and EXEC SP_ADDROLE.

- Authority operations. Authority operations are used to restrict the rights of users of the database. They include GRANT, DENY, and REVOKE.

- Backup operations. Backup operations are used to backup the whole or part of a database and perform recovery. They include EXEC SP_ADDUMPDEVICE, BACKUP DATABASE, and RESTORE DATABASE.

The control group is the set of operations that control the data flow and check the conditional environment. It is indicated as GroupC()={IF...ELSE..., GOTO, WHILE, BREAK, CONTINUE, BEGIN...END..., CASE...WHEN..., WAITFOR}.

6.4.2 Parsing SQL Code

The parsing of SQL legacy systems has two steps. The first step is to input SQL source code and acquire SQL tokens at the lexical level. The second step is to analyse these tokens at the syntactic level and to output the databases in the SQL legacy system and their relationships (Figure 6.1).

![Diagram of Parsing SQL Code]( Attached diagram with labels: SQL Source Code, SQL Lexical Analysis, Token, SQL Syntactic Analysis, Databases and Their Relationships. )

Figure 6.1: Process of Parsing SQL Code
6.5 Database-Based Model

A database-based model is a graph that describes the relationships between the databases in an SQL legacy system, indicated as DBM(). It is composed of nodes and lines. A node represents the database used in the program. The sequence of the database-based model DBM() is based on the relationships between the databases.

6.5.1 Structure

6.5.1.1 Database Classification

Databases are the main objects in SQL programs. If a database generates one or more other databases, or two or more database combine together to generate other databases, then the former database or databases are termed root databases, and the latter database or databases are termed node or leaf databases. The difference between a node database and a leaf database is that a node database must create the new database, and a leaf database must not create the new database. The database is regarded as the essential unit of SQL legacy systems, and is termed an SQL model entity.

6.5.1.2 Database Layer

The database layer is a number that represents the depth of a database DBi producing other databases, indicated as DL(DBi). The database layer of a root database is 0, the database layer of the node databases that are directly produced by root databases is 1, and the database layer of databases that are immediately produced by the database whose layer is 1, is 2, etc.

6.5.1.3 Relationships

The relationships between the databases in SQL legacy systems are mainly of two types: generation and association.

Let DBi, DBj, DBk be three databases in a database-based model DBM(). In the database-based model DBM(), database DBi is represented with one node.

The term generation means that a database is generated from another one or more databases, or a database is part of another database (Figure 6.2), presented as DBi >>
Chapter 6. SQL Development/Environment-Specific Models

DBj. In the database-based model DBM( ), if database DBs is connected below one or more other databases, such as DBi and DBj, it means that database DBs is generated from databases DBi and DBj, indicated as (DBi + DBj) >> DBs.

![Diagram: Generation Relationship of a Database-Based Model](image)

The term association means that a database works together with other databases. For example, if databases DBi and DBj join together to generate a new database DBk, then the relationship between DBi and DBj is an association (Figure 6.3), indicated as DBi DBj.

![Diagram: An Example of an Association Relationship](image)

Or, if database DBp generates two new databases DBq and DBr, then the relationship between DBq and DBr is an association (Figure 6.4), DBq DBr.

![Diagram: Another Example of an Association Relationship](image)

6.5.2 Classification of Database-Based Models

SQL Model 1: Association database-based model

An association database-based model is a database graph in which the relationships between databases are all associations (Figure 6.5).

![Diagram: Association Database-Based Model](image)
Let $DB_0$ be the original database, the set $SAD\{}$ be the set of all databases in $P$ ($SAD=\text{Set of All Databases}$), so that

$$SAD\{} = \{ DB_i | DB_i \in P \},$$

(6.1)

and the set $SSD\{}$ be the set of the databases whose relationships are the associations in $P$ ($SAD=\text{Set of Association Databases}$), so that

$$SSD\{} = \{ DB_i | (\exists k, k\neq i) \Rightarrow (DB_k \vartriangleleft DB_i) \}.$$  

(6.2)

If

$$SSD\{} = SAD\{},$$

(6.3)

then the database-based model of the SQL legacy system is of the association type.

SQL Model 2: Generation database-based model

A generation database-based model is a database graph in which the relationships between databases are all generations (Figure 6.6).

Let $DB_i, DB_j, DB_k$ be three databases in SQL legacy system $P$, the set $SGD\{}$ be the set of the databases whose relationships are the generations in $P$ ($SGD=\text{Set of Generation Databases}$), so that

$$SGD\{} = \{ DB_i | (DB_0 \gg DB_i) \}
\quad \text{OR} \quad ((\exists k, k\neq 0) \Rightarrow (DB_k \gg DB_i))
\quad \text{OR} \quad ((\exists j, j\neq 0) \Rightarrow ((\sum DB_j) \gg DB_i)) \}.$$  

(6.4)

If

$$SGD\{} = SAD\{} - \{DB_0\},$$

(6.5)

then the database-based model of the SQL legacy system is of the generation type.
Chapter 6. SQL Development/Environment-Specific Models

SQL Model 3: Composition database-based model

A composition database-based model is a database graph in which the relationships between databases are associations and generations existing in the same database graph. There are at least three states: two databases generate only one database; one database generates more than one database; two or more databases combine together to generate one or more databases (Figure 6.7, 6.8, 6.9 and 6.10).

Let the set $SCD\{\}$ be the set of the databases whose relationships include generation and association in the same model in $P$ ($SCD=$Set of Composition Databases), so that

$$SCD\{\} = \{ DB_i | (\exists k, k \neq i) \Rightarrow (DB_k \rightarrow DB_i) \}$$

$$AND ((DB_0 >> DB_i)$$

$$OR ((\exists s, s \neq 0) \Rightarrow (DB_s >> DB_i))$$

$$OR ((\exists t, t \neq 0) \Rightarrow ((\sum DB_i >> DB_i)) \}).$$

(6.6)

If

$$SCD\{\} \neq \emptyset,$$

(6.7)

then the database-based model of the SQL legacy system is of the composition type.

Figure 6.7: An Example of a Composition Database-Based Model

Figure 6.8: A Second Example of a Composition Database-Based Model
6.6 Summary

A brief history of SQL and the characteristics of a common platform, executable procedures, reduction of transformation, reuse of parts of a system, enhanced security strategy, and its index benefit, have been presented. Based on the basic operations of SQL, the two main relationships of generation and association between the databases in an SQL legacy system have been presented. Then three database-based models of SQL legacy systems, namely association, generation and composition, were described.
Chapter 7

Extracting UML Diagrams from COBOL Legacy Systems

7.1 Introduction

Based on the development/environment-specific model of COBOL legacy systems, which is a procedure-based model, the original code is restructured. Based on program slicing techniques, the definitions and slice criteria of the original COBOL program are acquired.

It is not necessary to use all the UML diagrams to model a COBOL legacy system. The static aspect of COBOL legacy systems is described with the UML class and composite structure diagrams. The dynamic aspect of COBOL legacy systems is depicted with the UML sequence and interaction overview diagrams.

The application of modelling rules is also discussed.

7.2 Using Four UML Diagrams to Model COBOL Legacy Systems

7.2.1 Static Modelling

UML 2.0 uses six diagrams to model the static parts of legacy systems, which are the class, object, component, package, deployment and composite structure diagrams. The object and composite structure diagrams are low-level modelling diagrams, whilst the class, component, package and deployment diagrams are high-level.

Among the UML diagrams, the class diagram is the most fundamental in modelling the structure of legacy systems. Because of the popular acceptance of the definition CLASS,
the class diagram is easy to understand when modelling legacy systems. The class diagram should therefore be used in this context.

An object has the same characteristics as the corresponding class. The class is the abstraction of the common characteristics of the object group. Most of the important characteristics of the object are reflected in the corresponding class. Therefore, after the class diagram has been used in modelling a COBOL legacy system, it is unnecessary to use the object diagram in modelling.

Although the composite structure diagram is low-level, its connotation is different from that of the object diagram. It is the description of the internal structure of a class and thus has different purposes from the object diagram. It can be used to model the complication of a class in detail. So the composite structure diagram is used in modelling the internal structure of the classes of a COBOL legacy system.

The component diagram shows the dependencies amongst software components, including the classifiers that specify them and the artifacts that implement them; these include source code files, binary code files, executable files, scripts and tables. A component represents a software entity in a system. The component diagram is used at a high level when analysing a COBOL legacy system. After the class diagram has been used in modelling a COBOL legacy system at a high level, the component diagram is pleonastic and redundant for modelling the functionality of COBOL code. In order to reduce redundancy and repetition when modelling COBOL legacy systems, the component diagram is not used.

The package diagram is composed only of packages and the dependencies between them [4]. A package is a UML construct that organises model elements, such as use cases or classes, into groups. Packages are depicted as file folders and can be applied to any UML diagram, although they are most common on use-case diagrams and class diagrams because these models have a tendency to grow [92]. Because a COBOL legacy system describes business rules and is complicated in practice, the package diagram applies at too high a level when modelling COBOL legacy systems. After the class diagram has been used, it is not necessary to use the package diagram to model COBOL code.

The deployment diagram presents processors, devices and the connections between
them, which can comprise the run-time configuration of hardware nodes and the software components that run on those nodes. A COBOL legacy system is modelled from the functional aspect. The deployment diagram is too high-level for analysing COBOL legacy systems from the static and structural points of view. Therefore, the deployment diagram is not used when modelling COBOL legacy systems.

7.2.2 Dynamic Modelling

UML 2.0 uses seven diagrams to model the dynamic parts of systems, namely the sequence, collaboration, activity, state machine, interaction overview, timing and use case diagrams. The timing, state machine and interaction overview diagrams are low-level modelling diagrams, whilst the sequence, collaboration, activity and use case diagrams are high-level.

A sequence diagram is a high-level UML diagram that models the dynamic aspect of systems. Sequence diagrams present the interactions between objects that achieve a result. A sequence diagram describes how groups of objects collaborate in presenting certain system behaviours. Typically, a sequence diagram describes the detailed implementation of how a legacy system accomplishes its main tasks. A sequence diagram presents an interaction in terms of a set of messages sent between objects that all work together to provide a desired operational result, which is its main difference from a collaboration diagram, which shows collaboration and association between instances in a system. The sequence diagram is useful when modelling a COBOL legacy system. It is used in presenting the dynamic aspects of the system.

A collaboration diagram shows the message flow between objects in an application, and also implies the basic relationships between classes. It is another form of sequence diagram although it is in a static style. A collaboration diagram shows details of how the objects within a scenario interact, such as showing visibility, etc. The UML collaboration diagram is used to model interactions between objects, and objects interact by invoking messages on each other [51, 101]. Because the composite structure diagram has been used to model the internal structure of the classes in COBOL legacy systems, it is not necessary to use the collaboration diagram.

An activity diagram describes the dynamic aspects of a system, also at a high level. It is
essentially a flowchart, showing the flow of control from activity to activity. An activity is an ongoing nonatomic execution within a state machine. Activity diagrams may stand alone to visualise, specify, construct, and document the dynamics of a society of objects, or they may be used to model the flow of control in an operation. Because the sequence diagram is used at a high level to describe the messages and the objects of a COBOL legacy system, and presents the operations with messages and the message senders or receivers, the activity diagram does not need to be used when modelling COBOL legacy systems.

A use case diagram is defined as a high level diagram that shows the relationships amongst actors and use cases within a system in terms of the UML specification. It shows a set of use cases and actors and their relationships. Use case diagrams can be used to model the context and requirements of a system from the stakeholders' point of view. A use case is a requirement that the users of the system, termed actors, want the system to do [28, 115]. A use case contains a special function that can be specified as a set of usage scenarios. As a user-centred analysis technique, the purpose of a use case is to yield a result of measurable value to an actor in response to the initial request of that actor. Use cases can be utilised to model the requirements of a system from the stakeholders' point of view [74]. Because the modelling of a COBOL legacy system focuses on the modelling of business rules in COBOL code and does not need to determine the requirements of the users, the use case diagram is not used when modelling COBOL legacy systems.

An interaction overview diagram is a diagram that is new in UML 2.0. It overviews the control flow within a system or business process at a low level. It focuses on overviewing the flow of control of the interactions. It describes the interactions where messages and lifelines are hidden. Because the interaction overview diagram is low-level and describes the control flow within a legacy system, it is suitable for modelling COBOL legacy systems and is therefore used in the approach to modelling COBOL legacy systems adopted by this thesis.

A timing diagram is used to explore the behaviours of one or more objects throughout a given period of time. The timing diagram is new in UML 2.0 for UML 1.3. It depicts the change of the state or condition of a classifier instance or role over time. Typically, it is used to show the change in the state of an object over time in response to external
Chapter 7. Extracting UML Diagrams from COBOL Legacy Systems

events. Timing diagrams are used to show changes and their relationships to clock times. It provides a visual representation of objects changing state and interacting over time. Because the timing diagram is suitable for the description of embedded systems, it is not used in modelling COBOL legacy systems.

A state machine diagram describes the possible states of a single class and the events that cause state transitions. It shows the sequence of states that an object goes through during its life cycle in response to stimuli. It is useful for showing the life cycle of the class. Generally, it is attached to a class of objects with an interesting dynamic behaviour. When a transition in a statechart is triggered, the object leaves its current state, initiates the action(s) for that transition, and enters a new state. Any internal or external event is broadcast to all states of all objects in the system [84, 89]. Because the objects in a COBOL legacy system are extracted from variables and they often have different values, the states of objects are changed so often that it is hard to capture their states and describe them. Therefore, the state machine diagram is not used when modelling COBOL legacy systems.

7.2.3 Modelling with Four UML Diagrams

Consequently, four UML diagrams: class, composite structure, sequence and interaction overview, are used to model COBOL legacy systems.

7.3 Modelling COBOL Legacy Systems with UML

7.3.1 Class Diagrams

*Rule 1: Using four UML diagrams to model COBOL legacy systems.*

There are four COBOL development/environment-specific procedure-based models as mentioned in Section 4.4.3: linear, branch, joint and synthetic.

Different models of programs in legacy systems have different structures, and different structures result in different UML class diagrams when modelling legacy systems with UML. So there are four kinds of UML class diagram corresponding to those four COBOL development/environment-specific models.
Chapter 7. Extracting UML Diagrams from COBOL Legacy Systems

A COBOL legacy system has a behavioural part, and is modelled using a UML sequence diagram. The four COBOL development/environment-specific models present not only the structural part but also the behavioural part, which comprises calling or being-called relationships. Those calling or being-called relationships represent transformations of the behaviour control from one procedure to other procedures. Therefore, four UML sequence diagrams are presented, based on the four COBOL development/environment-specific models.

A UML composite structure diagram depicts the internal structure of a classifier (such as a class, component, or use case), including the interaction points of the classifier to other parts of the system. It is most often used to show hidden internal details of a class, an object, or a component, and how the static architecture will achieve a requirement between elements that work together within a classifier. Although the COBOL development/environment-specific models are different from each other, all of them are composed of the procedure and variable classes that are extracted from the variables of the programs. Both of those two classes are the same in the four different COBOL development/environment-specific models. So in different COBOL development/environment-specific models, the procedure classes and the variable classes can be modelled with the same composite structure diagram. The composite structure diagrams of the procedure and variable classes will not be influenced by the differences between the four COBOL development/environment-specific models.

A UML interaction overview diagram overviews the control flow within a system or business process at a low level. It focuses on the overview of the flow of control of the interactions. It describes the interactions where messages and lifelines are hidden. No matter what the differences are between the sequence diagrams modelling the COBOL legacy system, the process of modelling the control behaviour and data flow of a procedure in a COBOL legacy system is the same. So the application of the interaction overview diagram is the same when modelling the low-level behaviours of the different procedures in the four different COBOL development/environment-specific models.

The development/environment-specific models of COBOL legacy systems are of four types: linear, joint, branch and synthetic procedure-based models. In order to simplify the research in this thesis and not lose universality, the computing of development/environment-specific models of COBOL legacy systems will focus on the
Chapter 7. Extracting UML Diagrams from COBOL Legacy Systems

synthetic procedure-based model, which is the most complicated one out of the four procedure-based models of COBOL legacy systems. The other three models will not be computed, but the algorithms are similar.

**Rule 2: Restructuring COBOL code.**

Software restructuring is the modification of software to make the software easier to understand and to change, or to make it less susceptible to error when future changes are made [24]. The definition of software restructuring excludes software changes for other purposes, such as code optimisation. Code optimisation does imply restructuring in a sense, but normally does not concern the key element of improving software maintainability.

Restructuring COBOL code includes deleting COBOL dead code, and valuating the isolated program elements, etc [67, 119].

**Rule 3: Slicing COBOL code of model entities with the Program slicing method.**

Some definitions are presented here, based on the approach of Weiser [132], Jiang [76] and Agrawal [2].

A flowgraph G is a 3-triple G=<N, E, n₀>, where N is the set of nodes, E is the set of edges, and n₀ is the initial node. If m and n are two nodes in N, m dominates n if and only if m is on every path from n₀ to n. It is indicated by m DOM n or DOM(n)=m.

A hammock graph HG is a quadruple HG=<N, E, n₀, n₁> with the property that <N, E, n₀> and <N, E⁻¹, n₁> are both flowgraphs, where E⁻¹={<a, b>|<b, a> ∈ E}. If m and n are two nodes in N, m reversely dominates n if and only if m is on every path from n₁ to n. It is indicated by m RDOM n or RDOM(n)=m.

Let G=<N, E, n₀>, n ∈ N and assume

\[
d_i \text{ DOM } n, \quad 1 \leq i \leq s \text{ and } d_i \in N
\]  

(7.1)

If there exists d_j satisfying

\[
d_i \text{ DOM } d_j, \quad \text{where } i=1,2,...,j-1,j+1, ...,s
\]  

(7.2)

then d_j is termed the immediate dominator of n. It is indicated as ID(n)= d_j.

Let G=<N, E, n₀>, n ∈ N and assume
Chapter 7. Extracting UML Diagrams from COBOL Legacy Systems

\[ r_j \text{ RDOM } n, 1 \leq j \leq t \text{ and } r_j \in N \quad (7.3) \]

If there exists \( r_i \) satisfying

\[ r_j \text{ DOM } r_i \text{, where } j=1,2,\ldots,l-1, l+1, \ldots,t \quad (7.4) \]

then \( r_i \) is termed the reversely immediate dominator of \( n \). It is indicated as \( \text{RID}(n) = r_i \).

For an arbitrary statement \( n \) and a slicing criterion \( C=<p, V> \), let

- \( \text{IMS}(n) \) represent the set of immediate successors of \( n \);
- \( \text{USE}(n) \) represent all the variables whose value might be used at \( n \);
- \( \text{MOD}(n) \) represent all the variables whose value might be modified at \( n \);
- \( \text{ND}(n) \) represent all the statements which are on a path from \( n \) to \( \text{RID}(n) \) excluding the endpoint and \( \text{RID}(n) \). \( \text{ND}(n) \) will be empty unless it has more than one immediate successor;
- \( \text{RIN}_c(n) \) be a set of variables related to the position of statement \( n \). Each variable in \( \text{RIN}_c(n) \) has potential effects on the values of variables in \( V \);
- \( \text{POS}(C) \) represent the statement position specified in the slicing criterion \( C \), i.e., \( \text{POS}(C) = p \);
- \( \text{VAR}(C) \) represent the variable set of slicing criterion \( C \), i.e., \( \text{VAR}(C) = V \).

For a slicing criterion \( C=<p, V> \) and an arbitrary statement \( n \) in program \( P \), the algorithm of computing that slicing criterion \( C \) is

\[
\text{RIN}_c^0(n) = \{ V | n = p \} \cup \{ \text{USE}(n) | \text{MOD}(n) \cap \text{RIN}_c^0(\text{IMS}(n)) \neq \phi \}
\]

\[ \cup \{ \text{RIN}_c^0(\text{IMS}(n)) - \text{MOD}(n) \} \quad (7.5) \]

\[
S_c^0 = \{ n | \text{MOD}(n) \cap \text{RIN}_c^0(\text{IMS}(n)) \neq \phi \} \quad (7.6) \]

\[
B_c^0 = \{ b | \text{ND}(b) \cap S_c^0 \neq \phi \} \quad (7.7) \]

\[
\text{RIN}_c^{i+1}(n) = \text{RIN}_c^i(n) \cup \text{RIN}_{BC}(b)^0(n) \quad (7.8) \]

\[
S_c^{i+1} = \{ n | \text{MOD}(n) \cap \text{RIN}_c^{i+1}(\text{IMS}(n)) \neq \phi \text{ or } n \in B_c^i \} \quad (7.9) \]

\[
B_c^{i+1} = \{ b | \text{ND}(b) \cap S_c^i \neq \phi \} \quad (7.10) \]

where \( i \geq 0 \).

The condition of stopping the iterations is

\[ \forall n, n \in N \Rightarrow \text{RIN}_c^{i+1}(n) = \text{RIN}_c^i(n) \]

OR

\[ S_c^{i+1} = S_c^i \quad (7.11) \]
The whole slice of the program $P$ for a slicing criterion $C$ is

$$S_c^{i+1} = S_c^i$$

where $i \geq 1$.

<table>
<thead>
<tr>
<th>Original Program</th>
<th>Sliced Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ACCEPT SALARY.</td>
<td>1 ACCEPT SALARY.</td>
</tr>
<tr>
<td>2 ACCEPT MONTHS.</td>
<td>2 ACCEPT MONTHS.</td>
</tr>
<tr>
<td>3 MOVE 0 TO FLAG1.</td>
<td>4 MOVE 0 TO MONEY.</td>
</tr>
<tr>
<td>4 MOVE 0 TO MONEY.</td>
<td>5 IF (SALARY&lt;=0) OR (MONTH&lt;=0)</td>
</tr>
<tr>
<td>5 IF (SALARY&lt;=0) OR (MONTH&lt;=0)</td>
<td>6 DISPLAY &quot;INVALID INPUT&quot;.</td>
</tr>
<tr>
<td>6 GO TO 001000-EXIT.</td>
<td>7 GO TO 001000-EXIT.</td>
</tr>
<tr>
<td>7 ELSE</td>
<td>8 ELSE</td>
</tr>
<tr>
<td>8 COMPUTE MONEY=SALARY*MONTHS</td>
<td>9 COMPUTE MONEY=SALARY*MONTHS</td>
</tr>
<tr>
<td>9 ENDIF</td>
<td>10 ENDIF</td>
</tr>
<tr>
<td>10 MOVE MONEY TO FLAG1</td>
<td>11 001000-EXIT.</td>
</tr>
<tr>
<td>11 001000-EXIT.</td>
<td>12 001000-EXIT.</td>
</tr>
<tr>
<td>12 EXIT.</td>
<td>13 EXIT.</td>
</tr>
</tbody>
</table>

Figure 7.1: An Example of A Slice Criterion (13, \{money\})

Figure 7.2: Control Flow Graph of Example Program

<table>
<thead>
<tr>
<th>DOM(n)</th>
<th>RDOM(n)</th>
<th>ID(n)</th>
<th>RID(n)</th>
<th>IMS(n)</th>
<th>ND(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>105</td>
</tr>
</tbody>
</table>
Chapter 7. Extracting UML Diagrams from COBOL Legacy Systems

| DOM(1) = Φ | RDOM(1) = {2,3,4,5,12} | ID(1) = Φ | RID(1) = 2 | IMS(1) = {2} | ND(1) = Φ |
| DOM(2) = {1} | RDOM(2) = {3,4,5,12} | ID(2) = 1 | RID(2) = 3 | IMS(2) = {3} | ND(2) = Φ |
| DOM(3) = {1,2} | RDOM(3) = {4,5,12} | ID(3) = 2 | RID(3) = 4 | IMS(3) = {4} | ND(3) = Φ |
| DOM(4) = {1,2,3} | RDOM(4) = {5,12} | ID(4) = 3 | RID(4) = 5 | IMS(4) = {5} | ND(4) = Φ |
| DOM(5) = {1,2,3,4} | RDOM(5) = {12} | ID(5) = 4 | RID(5) = 11 | IMS(5) = {6,9} | ND(5) = {6,7,9,10} |
| DOM(6) = {1,2,3,4,5} | RDOM(6) = {7,12} | ID(6) = 5 | RID(6) = 7 | IMS(6) = {7} | ND(6) = Φ |
| DOM(7) = {1,2,3,4,5,6} | RDOM(7) = {12} | ID(7) = 6 | RID(7) = 12 | IMS(7) = {12} | ND(7) = Φ |
| DOM(8) = Φ | RDOM(8) = {1,11} | ID(8) = 7 | RID(8) = 11 | IMS(8) = {11} | ND(8) = Φ |
| DOM(9) = {1,2,3,4,5} | RDOM(9) = {11,12} | ID(9) = 8 | RID(9) = 11 | IMS(9) = {11} | ND(9) = Φ |
| DOM(10) = {1,2,3,4,5,9} | RDOM(10) = {12} | ID(10) = 9 | RID(10) = 12 | IMS(10) = {12} | ND(10) = {7,11} |
| DOM(11) = {1,2,3,4,5} | RDOM(11) = Φ | ID(11) = 10 | RID(11) = 12 | IMS(11) = {12} | ND(11) = Φ |
| DOM(12) = {1,2,3,4} | RDOM(12) = Φ | ID(12) = 11 | RID(12) = 12 | IMS(12) = Φ | ND(12) = Φ |

Figure 7.3: Definition Examples of an Original Program

| USE(n) | MOD(n) | RINc°(n) |
| USE(1) = Φ{SALARY} | MOD(1) = {SALARY} | RINc°(1) = Φ |
| USE(2) = Φ{MONTHS} | MOD(2) = {MONTHS} | RINc°(2) = {SALARY} |
| USE(3) = Φ | MOD(3) = {FLAG1} | RINc°(3) = {SALARY, MONTHS} |
| USE(4) = Φ | MOD(4) = {MONEY} | RINc°(4) = {SALARY, MONTHS} |
| USE(5) = {SALARY, MONTHS} | MOD(5) = Φ | RINc°(5) = {SALARY, MONTHS, MONEY} |
| USE(6) = Φ | MOD(6) = Φ | RINc°(6) = {MONEY} |
| USE(9) = {SALARY, MONTHS} | MOD(9) = {MONEY} | RINc°(9) = {SALARY, MONTHS} |
| USE(11) = {MONEY} | MOD(11) = {FLAG1} | RINc°(11) = {MONEY} |
| USE(12) = Φ | MOD(12) = Φ | RINc°(12) = {MONEY} |

Figure 7.4: Computation of RINc°(n) of Example Program

| MOD(n) ∩ RINc°(IMS(n)) | RINc°(n) |
| MOD(1) ∩ RINc°(IMS(1)) = {SALARY} | RINc°(1) = Φ |
| MOD(2) ∩ RINc°(IMS(2)) = {MONTHS} | RINc°(2) = {SALARY} |
| MOD(3) ∩ RINc°(IMS(3)) = Φ | RINc°(3) = {SALARY, MONTHS} |
| MOD(4) ∩ RINc°(IMS(4)) = {MONEY} | RINc°(4) = {SALARY, MONTHS} |
| MOD(5) ∩ RINc°(IMS(5)) = Φ | RINc°(5) = {SALARY, MONTHS, MONEY} |
| MOD(6) ∩ RINc°(IMS(6)) = Φ | RINc°(6) = {MONEY} |
| MOD(9) ∩ RINc°(IMS(9)) = {MONEY} | RINc°(9) = {SALARY, MONTHS} |
| MOD(11) ∩ RINc°(IMS(11)) = Φ | RINc°(11) = {MONEY} |
| MOD(12) ∩ RINc°(IMS(12)) = Φ | RINc°(12) = {MONEY} |

S° = {1, 2, 4, 9}  S° = {1, 2, 4, 5, 7, 9, 12}

Figure 7.5: Computation of S° and S°(n) of Example Program
Chapter 7. Extracting UML Diagrams from COBOL Legacy Systems

As an example, Figure 7.1 shows the original COBOL program and its slice criterion \( C=\{13, \{\text{money}\}\} \). The control flow graph for that example program is shown in Figure 7.2. With respect to the slicing approach of Jiang, Figure 7.3 summarises \( \text{DOM}(n) \), \( \text{RDOM}(n) \), \( \text{ID}(n) \), \( \text{RID}(n) \), \( \text{IMS}(n) \), and \( \text{ND}(n) \). \( \text{USE}(n) \), \( \text{MOD}(n) \), and \( \text{RIN}_s(n) \) are computed by the algorithm presented in Figure 7.4. From the information in the figures and the definitions of the programming slices, the first level slice of that example program in Figure 7.5 is obtained, \( S_c^0=\{1, 2, 4, 9\} \). Based on the iterating conditions of program slicing, the slice criterion of that example program \( C=\{13, \{\text{money}\}\} \) is acquired, \( S_c^1=\{1, 2, 4, 5, 7, 9, 12\} \).

The original code contains a "GO TO" line in application programs written in the COBOL language that makes the source code become spaghetti code. In order to make the slice program sufficient, the statements including GOTO and corresponding labels are contained in the slice [17, 131, 132].

All the variables in the slice \( S_c \) with respect to the slice criterion \( C=(p, V) \) are composed of the relevant object set, the kernel of which is the variable \( V \).

According to Weiser’s introduction, slices are computed by computing consecutive sets of transitorily relevant statements based on data flow and control flow dependencies.

Program slicing consists of the computed part of a program that affect the values computed at some point of interest. It contains static and dynamic slicing. Dynamic slicing is defined as the kind of isolation that may relate to a specific input to the program, while static slicing relates to all possible inputs.

A number of hybrid applications using static and dynamic slicing methods have been suggested for the solutions of slicing programs containing procedures, unstructured control flow, composite data types and pointers, and concurrency, in terms of accuracy and efficiency. Program slicing is applied in debugging and program analysis, program differencing and integration, software maintenance, testing, tuning compilers, and other applications.

Program slicing is helpful in determining whether a change at some point in a program will affect the behaviour of other parts of the program in software maintenance. Static slicing decomposes a program into a set of components, and shows how each of those
components captures part of the original program’s behaviour, and how changes in a component can be merged back into the complete program in a semantically consistent way.

Because COBOL legacy systems involve data flow and control flow dependencies, they are suitable for the use of program slicing techniques in modelling them.

**Rule 4: Layering class diagrams into three.**

From the static point of view, the class diagram is treated as the king of all the diagrams in UML [83, 103]. The objects and classes of source code are identified and their relationships and attributes extracted, and legacy code is represented with UML class diagrams.

The structure of legacy code is complicated and it is displayed in different layers, which are represented with three different kinds of class diagram.

The class diagram which presents the leaf program elements in source code is termed the leaf class diagram, indicated as LEAF-PROCEDURE-NAME.

The class diagram which presents the node program elements in source code is termed the node class diagram, indicated as NODE-PROCEDURE-NAME.

The class diagram which presents the root program elements in source code is termed the root class diagram, indicated as ROOT-PROCEDURE-NAME.

**Rule 5: Starting from leaf program elements.**

Each program element is a single unit in a procedure-based model of a COBOL legacy system and is regarded as the essential model entity.

The root program element is the combination factory where its assemblies or parts are produced, although those assemblies or parts are the procedures. The root program is invoked directly by the user or some external process, otherwise there would be no way to execute it. By itself, the root program element is not a good candidate for the starting point of understanding the legacy system, since it calls other program elements and those program elements are unknown. The execution control goes back and forth among the program elements with the exception of the root program element in the legacy system. If the leaf program elements are not comprehended, it is difficult to understand
Node program elements are even more difficult to migrate than root program elements. They share the difficulty of root program elements, but also require that they must be created from the legacy system so that the remainder of the legacy system can continue to function in the same manner, and those node program elements can then be used by other program elements. Meanwhile, they include the difficulty of the leaf program elements that they call, and the flow of the execution control goes up and down to make the executing procedures more "spaghetti-like". The node program elements are called by the root program elements and call the leaf program elements. They are the worst candidates for comprehending the legacy system.

Isolated program elements can be migrated easily. These elements could be used in any given increment, since converting them does not increase or decrease the number of elements that need to be developed from the legacy system.

Leaf program elements are the best candidates for the starting point of comprehending the legacy system. They do not call back to legacy source code, and although they require development, it is possible to minimise the number of these elements by transferring an entire subsystem in a single iteration.

It is obvious that the root program element calls node or leaf program elements, leaf program elements are always called by root or node program elements, and isolated program elements neither call nor are called by any other program elements. Therefore, the leaf program elements are the first to be understood at the structural level.

**Rule 6: Generating pseudo classes.**

For a leaf program element P, PV is its variable set and POP is its operation set. If

\[
PV = \emptyset
\]  

(7.13)

then P is termed an empty program element. That empty program element is regarded as a class, indicated as CLASS Procedure-Name-Empty. The operations in the empty program element are transferred into the attributes and operations of that class.

For a leaf program element P, its slicing criterion \( C_i = \langle p, V_i \rangle \), \( 1 \leq i \leq n \), and the corresponding slicing program \( S_{ci} \), PC is the slicing criterion set \( PC = \{ C_i \} \), and PCV is
For the first slicing criterion $C_1 = \langle p, V_1 \rangle$, and its slice $S_{c1}$, $PCV(S_{c1})$ is composed of the variables of the slicing program $S_{c1}$. Let $V_1$ be the first pseudo class (Figure 7.6), and $PCV(S_{c1})$ is its attribute.

Let $V_2$ be the second variable, and

$$V_2 \in (PCV(P) - PCV(S_{c1})).$$

(7.14)

For the slicing criterion $C_2 = \langle p, V_2 \rangle$, its slice $S_{c2}$ and the variable set $PCV(S_{c2})$ are acquired. Let $V_2$ be the second pseudo class, and $PCV(S_{c2})$ is its attribute.

The iteration goes on until

$$(PCV(P) - \sum PCV(S_{ci})) = \phi.$$  

(7.15)

Then all the pseudo classes of all the leaf program elements of program $P$ are acquired.

Assume that $P$ is a procedure being sliced and $Q$ is a procedure which is called at statement $i$ in $P$. The algorithm of interprocedural slicing $CC$ extended from $P$ to $Q$ is:

$$CC = \langle n_1^Q, ROUT(i)_{f \rightarrow A} \cap SCOPE_Q \rangle$$

(7.16)

where $n_1^Q$ is the last statement of $Q$, $f \rightarrow A$ which means that the actual parameters will be replaced by formal parameters. $SCOPE_Q$ represents all variables which are accessible in procedure $Q$.

$$ROUT(i) = \bigcup RIN_C(j), \text{ where } j \in IMS(i).$$

(7.17)

**Rule 7: Generating real classes.**

Pseudo class $V_{ji}$ is slicing-dependent on $PCV(S_{cji})$. Each pseudo class is one group (Figure 7.7). Each group is related closely to each other for the purpose of describing the common core. Each group has one nucleus. Although that core is not always
obvious, every parameter and related operations are part of the specifications of that nucleus in that group. In every group, each parameter depicts one aspect of the characteristics of that nucleus, such as its name, its age, its weight, its height, or its ID; and every operation is the change, assessment, or detection of those aspects of the characteristics of that nucleus, such as increasing or decreasing its weight, confirming whether it is at that age, and determining whether it has another name.

![Figure 7.7: The i^{th} Pseudo Class V_{ji} of the j^{th} Layer Procedure](image)

All the parameters and operations in that group describe the attributes of that nucleus, the operations on that nucleus, and the relationships with other groups.

It is important to note that the name of the class, its attributes, and its operations are domain-related. The domain is the place in which the problem is allocated. All the definitions and extractions must be based on the domain knowledge [135].

Those parameters that are in the same group are served as its attributes because they are relevant to each other. They change the main data structure and are updated in terms of the control flow, and any operation that those parameters perform changes the state of the class attributes.

It is necessary to check the validity of the classes and corresponding operations and attributes. If a class is contained within another class, the former class is redundant for the latter.

For class $V_{js}$, if

$$\text{if } (\exists \ t, \ t \neq s) \Rightarrow (PCV(S_{ej}) \subseteq PCV(S_{cjs}))$$

then $V_{js}$ is termed an otiose class. If an otiose class is not a leaf class, it is deleted. Then the real classes are generated.

The relationship between the slicing criterion $C_{i} = \langle p, V_{i} \rangle$, $1 \leq i \leq n$ and $C_{j} = \langle p, V_{j} \rangle$, $1 \leq j \leq n$
Chapter 7. Extracting UML Diagrams from COBOL Legacy Systems

is indicated as \( RC(C_i, C_j) = PCV(S_{ci}) \cap PCV(S_{cj}) \). The relationship array corresponds to the presentation of the relationship between the slicing criteria \( C_i \) and \( C_j \), indicated as \( RAC(i, j) \).

\[
\begin{align*}
\text{If} \quad & RC(C_i, C_j) = \emptyset \\
\text{Then} \quad & RAC(i, j) = 0 \\
\text{Else} \quad & RAC(i, j) = 1 \\
\end{align*}
\]

(7.19)

The algorithm for computing the relationship array is

\[
\begin{align*}
\text{For } j &= 1 \text{ to } (n-1) \\
& \quad \text{For } i = (j+1) \text{ to } n \\
& \quad \quad \text{If } RC(C_i, C_j) = \emptyset \\
& \quad \quad \quad \text{then } RAC(i, j) = 0 \\
& \quad \quad \quad \text{Else } RAC(i, j) = 1 \\
& \quad \quad \text{End-If} \\
& \quad \quad RAC(j, i) = RAC(i, j) \\
& \quad \text{End-For} \\
& \quad RAC(j, j) = 0 \\
& \text{End-For} \\
\end{align*}
\]

(7.20)

Rule 8: Presenting systematic call classes and the systematic manager class.

Systematic calls are the operations that directly obtain functions from the resource of the operational environment in COBOL programming. They are represented as FUNCTION or CALL statements in original code. They are specific in realising the main functionality of legacy systems because they represent often used and already tested functions. Therefore, systematic calls are regarded as an individual class.

The systematic call class is composed of the systematic calls in selected original code. For example, part of the original code contains a systematic call "w$font" that uses four presented variables (Figure 7.8). It is represented in the systematic call class in Figure 7.9.

When a systematic call is used in a legacy system, it sometimes utilises the undefined variables in this program. In the following paragraph, the variable “aa3” is not defined in this procedure, but it is defined in another procedure (Figure 7.10). The variable “aa3” should be presented as an element in some other class.
Chapter 7. Extracting UML Diagrams from COBOL Legacy Systems

......
initialise wfont-data.
move "Times New Roman" to wfont-name.
move 12 to wfont-size.
move 1 to wfont-char-set wfont-family.
move 0 to wfont-bold-state.
call "w$font" using wfont-get-font, TNR-twelve-font, wfont-data.
......

Figure 7.8: Part Code Containing a Systematic Call

<table>
<thead>
<tr>
<th>Systematic call</th>
</tr>
</thead>
<tbody>
<tr>
<td>w$font: Systematic call</td>
</tr>
</tbody>
</table>

Figure 7.9: Example of Systematic Call Class Presentation

......
move 3 to aa1.
move 3 to aa1.
call "x$compute" using aa1, aa2, aa3.
......

Figure 7.10: Part Code Containing a Systematic Call

In complicated software systems, especially in early legacy systems written by programmers lacking experience, the variables in the original code are confused and cramped when defined and used. Unstructured programming languages give rise to repetition and redundancy of code statements and variables. So the classes produced from the legacy system will be numerous.

For example, the program presented below has three classes according to the method introduced above: day-hour, month-day, and year-month.

In order to illuminate the undefined variables and cope with redundancy and repetition, a new class termed Systematic Manager is defined, which is used to collect variables from the sliced set.

The systematic manager class is composed of the variables undefined in the procedure or collected from selected original code.
Chapter 7. Extracting UML Diagrams from COBOL Legacy Systems

Thus the systematic manager class of the example in Figure 7.11 is presented in Figure 7.12.

```cobol
Procedure division
   E1
   move 24 to day-hour.
   move 31 to month-day.
   move 12 to year-month.
   write day-hour.
   write month-day.
   write year-month.
   exit program
```

Figure 7.11: Part of Code Example

<table>
<thead>
<tr>
<th>Systematic manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>day-hour: int</td>
</tr>
<tr>
<td>month-day: int</td>
</tr>
<tr>
<td>year-month: int</td>
</tr>
</tbody>
</table>

Figure 7.12: Example of Systematic Manager Class

**Rule 9: Defining relationships between classes.**

An association shows a relationship between two or more classes. Associations have several properties:

- A name that is used to describe the association between two classes. Association names are optional and need not be globally unique.
- A role at each end that identifies the function of each class with respect to the associations.
- A cardinality at each end that identifies the possible number of instances.

Initially, the associations between classes are the most important because they reveal more information about the application domain. Every association should be named and roles assigned to each end. The associations between the classes are also domain-related.
It is necessary to model generalisation relationships between classes. Generalisation is used to eliminate redundancy from the analysis model. If two or more classes share attributes or behaviour, the similarities are consolidated into a superclass.

**Rule 10: Realising class diagrams.**

After acquiring all the classes of leaf program elements, the node program elements of program P need to be analysed. Because a leaf program element is a functional module and it is called by the node program elements, it is defined as a class in analysing node program elements that call leaf program elements.

For a leaf program element P, the leaf class is the class with respect to that leaf program element, when analysing the procedures and the functions calling it, indicated as CLASS LEAF-PROCEDURE-NAME (Figure 7.13). The class diagram which presents the node program elements in source code is regarded as a class in node and root class diagrams, and it is termed a node class, indicated as NODE-PROCEDURE-NAME (Figure 7.14).

![Leaf Class](Figure 7.13)

<table>
<thead>
<tr>
<th>Leaf Procedure-Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute: PV(P)</td>
</tr>
</tbody>
</table>

![Node Class](Figure 7.14)

<table>
<thead>
<tr>
<th>Node Procedure-Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute: PV(P)</td>
</tr>
</tbody>
</table>

In a node program element, its contained leaf and node program elements are presented with leaf and node classes. The other classes are realised as before. All node class diagrams are then obtained according to the increase of the procedure layers PLQ.

Assuming that the source code has the procedure layer \( j = n_0 \), \( C_{ji} \) represents the \( i^{th} \) slicing of the \( j^{th} \) layer, and \( P_j \) is the procedure whose procedure layer is \( j \). The algorithm
computing the pseudo classes (presented in Figure 7.7), which are not empty program elements, is similarly:

For \( j := 1 \) to \( n_0 \) DO
\[
C_j := p_j, V_{ji};
\]
\[
\text{ATTRIBUTE (CLASS } V_{ji} \text{)} = PCV(S_{cj});
\]
i := 1;
While \((PCV(P_j) - \sum PCV(S_{cj})) \neq \emptyset\) DO
\[
i := i + 1;
\]
\[
C_i := p_j, V_{ji};
\]
\[
\text{ATTRIBUTE (CLASS } V_{ji} \text{)} = PCV(S_{cj});
\]
End-While.
End-For

(7.21)

After all the leaf and node class diagrams are produced, the root class diagram is realised under the guidance above.

**Rule 11: Acquiring class diagrams of four models.**

The class diagram of a linear procedure-based model is described in Figure 7.15.

![Class Diagram of a Linear Procedure-Based Model](image)

The class diagram of a branch procedure-based model is described in Figure 7.16. The class diagram of a joint procedure-based model is described in Figure 7.17. The class diagram of synthetic procedure-based model is described in Figure 7.18.
Chapter 7. Extracting UML Diagrams from COBOL Legacy Systems

Figure 7.16: Class Diagram of a Branch Procedure-Based Model

Figure 7.17: Class Diagram of a Joint Procedure-Based Model

Figure 7.18: Class Diagram of a Synthetic Procedure-Based Model
Comparing the linear procedure-based model of a COBOL legacy system in Figure 4.6 with the synthetic procedure-based model in Figure 4.9 the former is simple in the extreme. Similarly, the class diagram of a COBOL legacy system whose procedure-based model is linear is considerably simpler than the synthetic one based on the condition (4.17). Suppose that the COBOL legacy system is satisfied with the condition (4.11), and the procedure-based model of that COBOL legacy system is linear as described in Figure 4.6. Then its class diagram is directly obtained as the description in Figure 7.15. The same applies to class diagrams of other procedure-based models of COBOL legacy systems. Therefore, the procedure-based model of COBOL legacy systems, in particular, diminishes the complexity of modelling and increases the efficiency of understanding. Obvious consequences are saved time and reduced costs.

7.3.2 Composite Structure Diagrams

**Rule 12:** Collecting variables of a class for composite structure diagrams.

A composite structure diagram in UML 2.0 depicts the internal structure of a classifier (such as a class, component, or use case), including the interaction points of the classifier to other parts of the system. Composite structure diagrams, which are new to UML 2.0, focus on instances and their internal structure, providing examples of how the static architecture will achieve a requirement.

Composite structure diagrams are composed of parts (classes with a defined role in the context of the enclosing composite), and connections (associations with limitations on potential links in the context of the enclosing composite). This diagram is most often used to show hidden internal details of a class, an object, or a component.

A composite structure diagram describes the relationships between elements that work together within a classifier. It shows parts and connectors. The parts are not necessarily classes in the model and they do not represent particular instances, but they may be roles that classifiers can play. The parts are shown in a similar manner to objects. For example, a message as the class has a name, length, id, user, location and content. A composite structure diagram is presented as Figure 7.19.

A composite structure diagram is sometimes used to show the runtime architectures of any kind of classifier.
Slicing criterion $C_i=<p, V_i>$ and its slice $S_{c1}$ in program $P$, $PCV(S_{c1})$ is composed of the variables of the slicing program $S_{c1}$. Assume that $V_1$ is a class, and $PCV(S_{c1})$ is its attributes. The composition of Class $V_1$ is the members $pcvv_k$ of the set $PCV(S_{c1})$, which is variable $pcvv_k$ of $PCV(S_{c1})$, $1 \leq k \leq n$. Then the presentation of $V_1$ is as in Figure 7.20.

The UML composite structure diagram is used to present the structure of the class and some artifacts, especially the leaf program elements that are termed the leaf class.

The variables that describe the leaf class in leaf program elements perform the function that is represented by the name of leaf class.

Assume that $x_i$ is a variable in leaf program element $LP$. Then for any variable $x_i$ in leaf program element $LP$,

$$\forall x_i, x_i \in LP \Rightarrow (x_i \in LVS(LP))$$  \hspace{1cm} (7.22)

$LVS()$ is termed the leaf variable set.

The leaf variable set is defined as the set of all the variables in a leaf program element, indicated as $LVS()$. It forms the collection of the parts of leaf class $LEAF-LP$ in a UML composite structure diagram. For example, the leaf COBOL program $DATE-EDIT$ below has the leaf variable set, $LVS(DATE-EDIT)=$\{DW-WORK-YEAR, DW-WORK-YYYY, DW-WORK-MONTH, DW-WORK-MM, DW-WORK-DAY, DW-WORK-DD, DW-DAYS-IN-MONTH(2), DW-WORK1, DW-WORK2\} (Figure 7.20).
Rule 13: Connecting the parts.

In composite structure diagrams, ports define the interaction point between a classifier and its environment or between a classifier and its internal parts. A port specifies the services that a classifier provides to and requires from its environment.

Collaborations and collaboration occurrences are also modelled in composite structure diagrams. A collaboration describes the roles and attributes that define a specific behaviour of the classifier. A collaboration occurrence represents a particular use of the collaboration to explain the relationships between the properties of a classifier. To identify the roles of the parts in the collaboration occurrence, a collaboration occurrence is attached to a collaboration and then the collaboration occurrence is added to a composite structure diagram.

Usually the connectors between the parts of the class in a composite structure diagram are association or dependency. They show that one part corresponds to one or more parts of the leaf class, one or more parts are dependent or independent on one special part, etc.
The close relationships between the variables in a leaf program element are the bases of the connectors. They are

- giving a value to
- getting a value from
- the conditional dependency
- the presupposition and the result
- the results and attendants of the computation.

**Rule 14: Realising composite structure diagrams.**

A composite structure diagram is similar to a class diagram, but it depicts individual parts instead of whole classes. A UML composite structure diagram depicts the internal structure of structured classifiers by using parts, ports, and connectors.

![Figure 7.22: Composite Structure Diagram of DateEdit](image)

After the parts and connectors of the leaf class are defined, it is easy to realise the corresponding composite structure diagram with rectangles and relationships between the parts of the leaf class. An example of a composite structure diagram is shown in Figure 7.22 based on the leaf program in Figure 7.21.

### 7.3.3 Sequence Diagrams

**Rule 15: Layering sequence diagrams.**

Description of the dynamic aspects of legacy code is complicated. The research in this section concentrates on the approach to modelling dynamic aspects of legacy code with UML sequence diagrams.

Firstly, objects in the source code are acquired through slicing the legacy COBOL code.
Secondly, objects from outside the source code are acquired. It is necessary to achieve the interactors that directly utilise that legacy code. The persons who utilise the legacy code, the hardware that executes it, and other systems that interact with it, are served as the interactors.

Thirdly, the operations of those objects are collected through program slicing, based on the timing and sequence of the source code.

Finally, the sequence diagrams of the legacy COBOL code are realised.

The key to modelling the dynamic aspects of legacy code with sequence diagrams is to achieve the objects of the legacy code.

The operations of legacy code are classified into different layers, especially in legacy code containing node and leaf program elements. Those operations that are similar and work sequentially and together are collected and presented as a message. In legacy code, some kinds of complex computation, especially many mathematical formulas, are represented by the messages in that application domain [104]. Meanwhile, it is fundamental to order the messages of legacy code in sequence diagrams according to the time and sequence they are executed by. The objects from which and to which the messages are sent are recorded. It is important to concentrate on the critical operations in order to refine the messages of legacy code. Not all the operations are treated as messages or one message only refers to one operation [96, 106].

As with class diagrams, three different kinds of sequence diagram are defined, based on the structure of the legacy code in different layers.

The sequence diagram which presents the leaf program elements in the source code is termed the leaf sequence diagram, indicated as LEAF-PROCEDURE-NAME.

The sequence diagram which presents the node program elements in the source code is termed the node sequence diagram, indicated as NODE-PROCEDURE-NAME.

The sequence diagram which presents the root program elements in the source code is termed the root sequence diagram, indicated ROOT-PROCEDURE-NAME.

The analysis of legacy COBOL systems with UML sequence diagrams starts from leaf program elements.
Chapter 7. Extracting UML Diagrams from COBOL Legacy Systems

**Rule 16: Defining leaf and node objects.**

In analysing legacy COBOL systems, there are two objects that are especially important, one of which is Leaf-Object, the other being Node-Object.

The object that represents the leaf program elements in node sequence diagrams and root sequence diagrams that call this leaf program element is termed the leaf object, indicated as LEAF-PROCEDURE-NAME.

The object that represents the node program elements in root sequence diagrams and note sequence diagrams that call this node program element is termed the node object, indicated as NODE-PROCEDURE-NAME.

**Rule 17: Slicing the program.**

According to the description in Rule 3 in Section 7.3.1, for program P, VAR(P) is the set of parameters of program P. Then it is sliced.

**Rule 18: Acquiring objects in source code.**

Let x be an arbitrary parameter of P, x ∈ VAR(P). Its program slicing is C(x) = <n, x>. Every element x of VAR(P) is a pseudo object. VAR(P) is a pseudo object set of P.

Let x₁ and x₂ be two parameters of P, x₁ ∈ VAR(P) and x₂ ∈ VAR(P). For the slicing C(x₁) = <n, x₁> and C(x₂) = <n, x₂>, if C(x₁) ⊆ C(x₂), then VAR(P) = VAR(P)-{ x₁ }.

Check every parameter of VAR(P), and then the remaining VAR(P) is the object set.

Let xᵢ be arbitrary parameters of P, 1 ≤ i ≤ k, xᵢ ∈ VAR(P), and VAR(P) has k parameters. Then the object of P is computed through:

```
For i=1 to (k-1)
    For j=(i+1) to k
        If C(xᵢ) ⊆ C(xⱼ) then
            VAR(P) = VAR(P)-{ xᵢ }
        End-if
    End-For
End-For
```

(7.23)
For an arbitrary object $x_i \in \text{VAR}(P)$, $1 \leq i \leq m$, its slicing is $C(x_i) = \langle n, x_i \rangle$.

**Rule 19: Finding the interactors.**

It is important to identify who is going to be using the legacy code directly. This should be done from outside the legacy code, based on domain knowledge. The matter is deeply involved with human interaction and closely related to domain knowledge. The candidates for actors include the people who interact with the code, the hardware that is external from the code, and the other systems that have interaction with the code.

Software interacts with people and other systems in the real world. By “interactors of the code” is meant the users of it and other systems that interact with it. The interactors have relationships with the legacy code. The users of the code will perform special tasks with it and the code can exchange information with other software, sending data to or receiving data from other software.

When the code is executed, it may exchange messages with other systems. It may send data to and receive data from other systems, send signals to control other systems, and receive signals to be controlled or be executed by other hardware systems. All these systems are regarded as interactors of the legacy code.

Different interactors look at legacy code from different points of view. Each interactor is interested in a special aspect of the legacy code. All the interactors are candidates to be actors of the legacy code.

The stakeholders are closely related to the software system. They are important to the analysis of legacy systems. End users, analysts, developers, system integrators, testers, technical writers and project managers are all different kinds of stakeholder—each brings a different agenda to a project, and each looks at the system in different ways at different times over the project’s life. System stakeholders are the people who are closely related to the system. They can be direct users, indirect users, managers of the users, senior managers, supporters who provide help, buyers who invest in the system, developers who work on other systems and interact with the system being developed, maintainers who will maintain the system, and system developers who will work on it.

Active stakeholder participation implies the need to have onsite access to people, typically users or their representatives, who have an authority and the ability to provide
information pertaining to the system, and to make pertinent and timely decisions regarding the requirements and prioritisation. System success requires a great deal of involvement by system stakeholders. The system's goals will be accomplished by the contributions of all the stakeholders.

Some of the stakeholders are regarded as the candidates of the system actors. This is vital to the identification of the objects of the legacy system and the realisation of the sequence diagrams. Because people are generally the most important candidates of the actors, it is important to take into account the stakeholders of the legacy software system. Some of the stakeholders can be regarded as actors with regard to the legacy software system. Because legacy code is critical for businesses and organisations, it is able to accomplish specified tasks, albeit ineffectively, inefficiently and riskily. The direct and indirect users, managers of the company, supporters who provide help, and maintainers of the legacy code, all are interactors.

It is essential to regard each of those interactors as an actor. If there are more than two actors, a person should be considered as the most important actor candidate. Because only people can utilise the software directly or indirectly for their own purposes, it would be better to take into account all the users, maintainers and managers of the company, or even the organisation of the business (Figure 7.23).

![Sequence Diagram for Printing Header]

**Figure 7.23: A Sequence Diagram for Printing Header**

It is important, when deciding who are the actors, to think over the importance of the candidates. It is correct that all candidates for actors are important to the legacy code. However, different candidates may be of different importance as regards the legacy code. The direct end users are the most important among all the candidates for actors of
the legacy code. In most cases, they would utilise the legacy code and determine when it is executed. The operational results are reported directly to them. Then they decide on what should be done next. Next may be the company managers, because they not only use the legacy code indirectly but also manage the direct end users of the legacy code in the company. Perhaps they direct a project of the business based on the output of the legacy code. Both the direct end users of the legacy code and the company managers use the legacy code for commercial purposes and in some cases their uses may be the same. Next may be the indirect users and the maintainers. However, in most cases they do not use the legacy code for business purposes.

Other interactors besides human beings are considered and chosen as actors, including the hardware that executes the legacy code and other systems that interact with it. This describes the interacting effect and process of how the legacy code is executed in the hardware and what messages are exchanged with other systems. Any systems that invoke the legacy code are regarded as actors.

**Rule 20: Getting the sequential array.**

When understanding the operations of legacy code, the preconditions and post-conditions of each operation are presented based on domain knowledge [87, 108]. Moreover, it is fundamental to order the operations of the legacy code in sequence diagrams according to the time and sequence they are executed by. The objects from which and to which the information is sent are recorded.

Let \( C(x) = <n, x> \) be a slicing of program P, \( s \) be an operation of \( C(x) \), \( m \) the sequential number, \( v_1 \) the input parameter set of \( s \), \( v_2 \) the output parameter set of \( s \), \( z_1 \) the preconditions of \( s \), \( z_2 \) the post-conditions of \( s \). The sequential array \( SA(C(x)) \) is

\[
SA(C(x)) (m) = (m, v_1, v_2, s, z_1, z_2) \tag{7.24}
\]

**Rule 21: Presenting the message array.**

The sequential array \( SA(C(x)) \) is basic for knowledge theory from the point of view of domain knowledge, and is dealt with in the messages which will form the new array \( MA(C(x)) \).

Assume that \( C(x) = <n, x> \) is a slicing of program P. Let \( mes \) be a message of \( C(x) \), k
the sequential number, \( v_1 \) the input parameter set of mes, \( v_2 \) the output parameter set of mes, \( z_1 \) the preconditions of mes, \( z_2 \) the post-conditions of mes. The message array \( MA(C(x)) \) is

\[
MA(C(x)) (k) = (k, v_1, v_2, mes, z_1, z_2)
\] (7.25)

It is important to use the domain knowledge in the message presentation of P [108]. Those operations that are similar and work sequentially and together are collected and presented as one message [87]. For example, the three operations of sending the value of the day, the month and the year to the date are extracted into one message that is termed “sending the date”.

In legacy code, some kinds of complex computation, especially many mathematical formulas, are represented by messages in the appropriate application area. For instance, the formula

\[ S = \text{Length1} \times \text{Length2} \]

is achieved as the message computing the rectangle area.

It is important to concentrate on the critical operations in order to refine the messages of legacy code. Not all operations are treated as messages, and, especially, a single operation is not regarded as a single message. The messages are based on domain knowledge (Figure 7.24 and 7.25).
Chapter 7. Extracting UML Diagrams from COBOL Legacy Systems

WRITE PRINT-REC AFTER
ADVANCING 1.
ADD 1 TO W00-LINE-COUNT.
MOVE W01-HEADER-6 TO PRINT-DATA.
WRITE PRINT-REC AFTER
ADVANCING 1.
ADD 1 TO W00-LINE-COUNT.
MOVE SPACES TO PRINT-DATA.
WRITE PRINT-REC AFTER
ADVANCING 1.
ADD 1 TO W00-LINE-COUNT.
PRINT-HEADER-2-END.
EXIT.

Figure 7.24: A Program Example

Figure 7.25: A Sequence Diagram of Printing Rest Header

Rule 22: Realising sequence diagrams.

A sequence diagram is an interaction diagram that details how operations are carried out – what messages are sent and when (Figure 7.26). Sequence diagrams are organised according to time. Normally time proceeds down the page.

A sequence diagram has two dimensions:

(1) the vertical dimension represents time.

(2) the horizontal dimension represents object interaction.

The vertical line is termed the object's lifeline. The lifeline represents the object's life during the interaction. A message is represented by an arrow between the lifelines of the sender and the receiver objects. A message is shown as a horizontal solid arrow from the lifeline of the sender to the lifeline of the receiver. The arrow is labelled with the
name of the operation to be invoked or the name of the signal. Its argument values or argument expressions may be presented as well. The arrow may also be labelled with a sequence number. Optionally, a message can be prefixed with an * (iteration maker), which shows that the message is sent many times.

Sequence diagrams are used to demonstrate the flow of control for a certain part of a program. It shows how objects in the system interact, based on messages sent and returned. Sequence diagrams show the timeflow of messages and present a detailed view of relationships between objects, and of the passage of messages.

Layering is a common approach to systems to be organised. As a result, it makes sense to layer the sequence diagrams of legacy code in a similar manner. That is done based on the layers of the program calls in the legacy code. The root program element is regarded as the first and the most important sequence diagram. Other program elements are included in that diagram. The node program elements are presented before the leaf and isolated program elements.

The primary actor of the legacy system is allocated at the top left hand side of the sequence diagram. Other actors follow in order of time and importance. Actors that are the reactors of the legacy system are described at the top right hand side of the sequence diagrams and are treated as the entities that the legacy code interacts with.

The message name is justified and aligned with the arrowhead. The receiver of the message implements the corresponding operations and it makes sense that the message name is close to that of the classifier. The syntax of the implementation language of the legacy code is utilised in naming the messages. This improves understandability and readability.

In a sequence diagram, an object receives messages and invokes the operation of time ordering. Only if a message has been executed can the next message then be performed in the time dimension. So the time periods when messages are executed by objects are clearly shown on sequence diagrams (Figure 7.26).

Return values are common in legacy code. When an object finishes processing a message, control returns to the sender of the message. This marks the end of the activation corresponding to that message, and is marked by a dashed arrow going from
Chapter 7. Extracting UML Diagrams from COBOL Legacy Systems

the bottom of the activation rectangle back to the lifeline of the role that sent the message giving rise to the activation. Activations and return messages are optional in a sequence diagram. They are optionally indicated using a dashed arrow with a label indicating the return value. When they are referred to in the next part of the sequence diagram, it is necessary to model the return values. Otherwise, they are ignored in sequence diagrams in order to make those diagrams of legacy code clearer and simpler.

The operations of a legacy code are classified into different layers, especially in legacy codes containing node and leaf program elements. They are described in the sequence diagrams of that legacy code.
Figure 7.26: An Example of a Sequence Diagram

**Rule 23:** Presenting the sequence diagrams of four models.
Chapter 7. Extracting UML Diagrams from COBOL Legacy Systems

The sequence diagram of a linear procedure-based model of a COBOL legacy system is presented in Figure 7.27, that of a branch model in Figure 7.28, and that of a joint model in Figure 7.29.

Figure 7.27: Sequence Diagram of a Linear Procedure-Based Model

Figure 7.28: Sequence Diagram of a Branch Procedure-Based Model

Figure 7.29: Sequence Diagram of a Joint Procedure-Based Model

The sequence diagram of a synthetic procedure-based model of a COBOL legacy
Chapter 7. Extracting UML Diagrams from COBOL Legacy Systems

system is presented in Figure 7.30.

Figure 7.30: Sequence Diagram of a Synthetic Procedure-Based Model

As mentioned at the end of Section 7.3.1, when comparing the sequence diagram of the linear procedure-based model in Figure 7.27 with the synthetic procedure-based model in Figure 7.30, the former is simple and direct, whilst the later is complex and confused. If a COBOL legacy system has the condition (4.11), which means that its procedure-based model is linear, then its sequence diagram is directly obtained from the model in Figure 4.6 as Figure 7.27. The same applies to the sequence diagrams of other procedure-based models of COBOL legacy systems. Therefore, the application of models raises the efficiency and improves the quality of modelling COBOL legacy systems.

7.3.4 Interaction Overview Diagrams

**Rule 24: Layering interaction overview diagrams.**

Interaction overview diagrams are new in UML 2.0. They overview the control flow within a system or business process (Figure 7.31). The nodes within a diagram are frames instead of the normal activities which can be seen on activity diagrams. There are two types of frame shown: interaction frames which depict any type of UML interaction diagram; and interaction occurrence frames which indicate an activity or operation to invoke. Decision points are shown as diamonds, exactly as on UML activity diagrams. There should be guards on all the exiting flows, labelling some of the guards when it is not obvious what is meant.
Chapter 7. Extracting UML Diagrams from COBOL Legacy Systems

The description of dynamic aspects of legacy code is complicated. The research in this chapter concentrates on modelling dynamic aspects of legacy code with UML interaction overview diagrams. The operations of the legacy code are classified into different layers, especially in legacy code containing node and leaf program elements.

The interaction overview diagram which presents the leaf program elements in source code is termed the leaf interaction overview diagram, indicated as LEAF-PROCEDURE-NAME.

The interaction overview diagram which presents the node program elements in source code is termed the node interaction overview diagram, indicated as NODE-PROCEDURE-NAME.

The interaction overview diagram which presents the root program elements in source code is termed the root interaction overview diagram, indicated as ROOT-PROCEDURE-NAME.

Figure 7.31: An Example of a Interaction Overview Diagram based on Figure 7.1
Chapter 7. Extracting UML Diagrams from COBOL Legacy Systems

code is termed the root interaction overview diagram, indicated as ROOT-PROCEDURE-NAME.

The analysis of legacy COBOL systems with UML interaction overview diagrams starts from leaf program elements.

**Rule 25: Defining the fork node.**

Let $s$ be a statement of program $P$. Node $s$ in the corresponding program control flow graph is termed the fork node if and only if

$$ND(s) \neq \emptyset$$  \hspace{1cm} (7.26)

In Section 4.3.1, basic COBOL operations are presented. Group Two and Group Three, which are conditional operations and control flow operations respectively, contain the fork nodes [88]. The fork nodes of COBOL code are composed of the set $FN(P)$, where

$$FN(P)= \{IF, IF...ELSE..., EVALUATE, PERFORM...UNTIL (BY)..., GOTO...LABEL...\}$$

Statements IF and IF...ELSE... are fork nodes of COBOL. They are normal to enabling the realisation of the choices of the execution (Figure 7.32).

![Figure 7.32: Fork Node IF...ELSE...based on Figure 7.2](image)

The IF statement is used in Figure 7.33, and its control flow graph of fork node IF is Figure 7.34.

![Figure 7.33: Fork Node IF](image)
The IF...ELSE... statement is used as in Figure 7.35. The control flow graph of fork node IF...ELSE... in Figure 7.35 is Figure 7.36.
Statement EVALUATE is the fork node in COBOL code. It is similar to the fork node IF...ELSE... (Figure 7.37).

Statement GOTO...LABEL is the fork node in COBOL source code. A typical usage of the GOTO statement is in Figure 7.38, and its control flow graph is given in Figure 7.39.

Figure 7.36: Control Flow Graph of Fork Node IF...ELSE...

Figure 7.37: Control Flow Graph of Fork Node EVALUATE

Figure 7.38: Fork Node GOTO
Rule 26: Defining joint node and action node.

Let $s$ be a statement of program $P$. Node $s$ in the corresponding program control flow graph is termed the joint node if and only if

$$s \in JN(P) \quad (7.27)$$

where

$$JN(P) = \{\text{END-IF, END-EVALUATE, EXIT, STOP RUN}\}$$

Let $s$ be a statement of program $P$. Node $s$ in the corresponding program control flow graph is termed the action node if and only if

$$s \in AN(P) \quad (7.28)$$

where

$$AN(P) = \text{VERB(SS)} \cup \text{VERB(SCD)} \cup \text{VERB(SCT)} - FN(P) - JN(P)$$

VERB(SS), VERB(SCD), and VERB(SCT) are defined in Section 4.3.1, and FN(P) and JN(P) in this section. That is, AN(P)={ ACCEPT, DISPLAY, MOVE, INITIALISE, SET, ADD, COMPUTE, DIVISION, MULTIPLY, SUBTRACT, STRING, UNSTRING, INSPECT, INITIATE, GENERATE, TERMINATE, SUPPRESS, USE BEFORE REPORTING, IN, OF, FUNCTION, DELETE, OPEN, CLOSE, READ, WRITE, REWRITE, COPY, RELEASE, SORT, RETURN, START, USE...AFTER..., USE FOR DEBUGGING, SELECT , CONTINUE, SEARCH, CALL, PERFORM,
Chapter 7. Extracting UML Diagrams from COBOL Legacy Systems

Rule 27: Layering nodes.

Let $s$ be a node in the control flow graph of program $P$. The node layer is the number of the node that represents its stratum in the program. It is indicated as $NL(s)$.

Table 7.1: One Example of Node Layer $NL()$

<table>
<thead>
<tr>
<th>Node Number</th>
<th>Source Code</th>
<th>$NL()$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PROCEDURE DIVISION.</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>OPEN OUTPUT StudentFile</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>DISPLAY &quot;Enter student details&quot;</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>PERFORM GetStudentDetails</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>PERFORM UNTIL StudentDetails = SPACES</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>WRITE StudentDetails</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>PERFORM GetStudentDetails</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>END-PERFORM</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>CLOSE StudentFile</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>STOP RUN</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>GetStudentDetails.</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>DISPLAY &quot;Enter Id, Name, Gender&quot;</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>ACCEPT StudentDetails</td>
<td>1</td>
</tr>
</tbody>
</table>

For example, in Table 7.1, the node layer of Node 1 is 0, $NL(1)=0$. That is, the node layer of the procedure name is 0. Then, the node that is executed after the procedure name has the node layer 1, $NL(2)=1$. If the next node is the fork, the first statement in the fork has node layer 2. Similarly, for the statement that in the PERFORM statement has node layer 2, $NL(6)=2$. And so on.

Rule 28: Presenting the control structure.

Let $s$ be a node of program $P$, $v$ be a variable of $P$, $FNS$ be the set of fork nodes of program $P$, so that

$$FNS(s) = \{ s | (s \in P) \text{ AND } (s \in FN(P)) \}$$  \hspace{1cm} (7.29)

Let $FNVS$ be the set of the variables of $FNS$, so that

$$FNVS(v) = \{ v | (v \in P) \text{ AND } (v \in FNS(s)) \}. \hspace{1cm} (7.30)$$

Assume that $\forall v_k \in FNVS(s), 1 \leq k \leq T, v$ is program sliced and its slicing criterion $C_k = \leq n,$
Chapter 7. Extracting UML Diagrams from COBOL Legacy Systems

\( v_k \). Every \( C_k \) has a number of nodes. Finally, the set \( CSS() \) of all sliced nodes is generated,

\[
CSS(s) = \sum C_k \tag{7.31}
\]

The control structure of program \( P \) \( CS(P) \) is formed by the nodes in \( CSS() \) and the fork nodes, that is,

\[
CS(P) = CSS(P) \cup FN(P). \tag{7.32}
\]

Let \( PL_i \) be a leaf program element of original COBOL code, its control structure \( CS(PL_i), 1 \leq i \leq t \). The basic control structure \( BCS(PL_i) \) is

\[
BCS(PL_i) = CS(PL_i) \cup JN(PL_i) = FN(PL_i) \cup JN(PL_i) \cup CSS(PL_i) \tag{7.33}
\]

In order to realise the interaction overview diagram, the analysis of program \( P \) starts from the leaf interaction overview diagram with the basic control structure \( BCS(PL_i) \) and the node layers \( NL() \).

**Rule 29: Realising interaction overview diagrams.**

Let \( s \) be an arbitrary node of the interaction overview diagram of program \( P \). The algorithm of the realisation of the interaction overview diagram is

For \( j := 0 \) to \( \text{MAX}(NL(P)) \) DO

While \( (BCS_j(P) \neq \emptyset) \) DO

If \( (s \in BCS_j(P)) \) AND \( (NL(s) = j) \) Then

\[
BCS_j(P) := BCS_j(P) \setminus \{s\}
\]

End-If

End-While

While \( (AN_j(P) \neq \emptyset) \) DO

If \( (s \in AN_j(P)) \) AND \( (NL(s) = j) \) Then

\[
AN_j(P) := AN_j(P) \setminus \{s\}
\]

End-If

End-While

End-For \( \tag{7.34} \)

The interaction overview diagram of leaf program element \( PL_i \) is composed of basic
control structure BCS(PL_i) and action nodes AN(PL_i):

- Start point and end point. Every UML interaction overview diagram has a start point. The start point is allocated at the top-left corner of the interaction overview diagram and the end point is at the bottom-right corner. Both of them are modelled as a filled-in circle with a border around it.

- Fork nodes FN(PL_i) and joint nodes JN(PL_i). It is possible that activities can occur in parallel that are regarded as forks and joins. A fork should have a corresponding joint. Each fork has one entry transition and each join has one exit transition.

- Sliced nodes CSS(PL_i). Sliced nodes are arranged in the interaction overview diagram based on the sequence of the nodes.

- Action nodes AN(PL_i). An activity on a UML interaction overview diagram typically represents the invocation of an operation, a step in a business process, or an entire business process. When an activity, except for the starting or ending point, has a transition into it but none out, or has a transition out but none into it, it is possible that one or more transitions have been missed.

In a UML interaction overview diagram, a decision point is modelled as a diamond. The guards (depicted using the format "[description]", which is a condition that must be true in order to traverse a transition), on the transitions leaving decision points, help to describe the decision points. Each transition leaving a decision point must have a guard. The guards on the transitions leaving a decision point, or an activity, must be considered alongside each another, and should not overlap. The guards on decision points form a complete set. It is necessary that exit transition guards and activity invariants must form a complete set.

Let PN_j be a node program element of original COBOL code, its containing leaf program elements PL_i, 1 ≤ i ≤ t. PL_i is termed the procedure node of node program element PN_j, which is indicated as

\[
\text{PL}_i \in \text{PCN}(\text{PN}_j)
\]  

(7.35)

The leaf program elements PL_i in a node program element PN_j are regarded as the action nodes of PN_j. Therefore, the drawing of the node interaction overview diagram is the same as the drawing of the leaf program element.
Interaction overview diagrams use the right level of detail in order to describe the system's functionality. A model is a communication device, so it requires an adequate level of detail to address the problem to be solved. Clarity and brevity are important to avoid visual overload, and a model should present key features of the control flows.

It is important to limit the level of complexity of each interaction overview diagram. If there are more than three possible paths (alternate or exceptional), it is necessary to use additional interaction overview diagrams to promote understanding. It is also necessary to use additional interaction overview diagrams if the processing requires specific data elements.

After all the node interaction overview diagrams have been acquired, the root interaction overview diagram is realised, based on the above.

7.3.5 Application of COBOL Rules

In some cases, when a program element of a COBOL legacy system has only a few variables, or even none at all, it is not necessary to present its class diagram and composite structure diagram. If a program element is short in length, such as only thirty lines of statements, it is not necessary to obtain its sequence diagram and interaction overview diagram. Or, if a program has more than a hundred statements, but its structure is only repetition of an "IF...ELSE..." statement, its interaction overview diagram is not necessary.

So, when a program element has one of the following characteristics, it is unnecessary to follow those rules:

**C1:** The number of variables is not more than ten.
**C2:** The number of statements is not more than fifty.
**C3:** The program structure is composed of the repetition of a fixed combination of fewer than five kinds of statement in Group One in Section 4.3.
**C4:** The program structure is composed of the repetition of a condition or loop statement in Group Two or Group Three in Section 4.3.
**C5:** A generated class contains not more than five variables.

As mentioned above, Rules 1 to 11 describe class diagrams. The COBOL legacy system is input, restructured and sliced. When condition C1 occurs in Figure 7.40, it means that
this program element has no more than ten variables and it is unnecessary to model with class and composite structure diagrams.

Rules 12 to 14 describe composite structure diagrams. The variables in a class are collected. When condition C5 occurs, it means that a generated class contains not more than five variables. It is not necessary to model this class with composite structure diagrams.

Rules 15 to 24 describe sequence diagrams and Rules 25 to 29 describe interaction overview diagrams. A program element is sliced. When condition C2 occurs, it means that this program is shorter than fifty statements in length and it is not necessary to model with sequence and interaction overview diagrams. If a program is longer than fifty statements but it is composed of the repetition of a condition or loop statement, then the structure of this program element is simple and it is not necessary to model it with interaction overview diagrams. After presenting sequence diagrams, when condition C3 occurs, it means that a program element is composed of the repetition of the fixed conditions of fewer than five kinds of statement in Group One. Its structure is simple and it is easy to understand. It is not necessary to model it with interaction overview diagrams.

Consequently, when conditions C1 to C5 occur, the complexity of modelling COBOL legacy systems is greatly reduced at that point.
7.4 Summary

In this chapter, class diagrams and composite structure diagrams are used to describe the static parts of a legacy system and its internal structure. Sequence diagrams and interaction overview diagrams are used to model the dynamic parts of a legacy COBOL system. Because of the presentation of the most important characteristics, or another form of sequence diagram, or pleonasm and redundancy in modelling the functionality
of COBOL code, or the focus on business proceedings, other UML diagrams are not used.

Following the 29 rules, those four UML diagrams are obtained. At the end, application of rules is presented. Normally, the modelling of a COBOL legacy system should follow these rules, but in some cases it is different. Five conditions are described.
Chapter 8

Extracting UML Diagrams from HTML Legacy Systems

8.1 Introduction

The information in every corner of the world is linked through web, and at the beginning the web was designed with HTML. The desired web page is realised through the correct usage of the tags by the translation of web browsers. And the tables, forms, images, and sounds appear in web pages in the designed format [52].

HTML is the presentation of web and data format, and it does not focus on data flow or control flow dependences. It has no behavioural functionality and is not sliced by program slicing techniques.

A HTML legacy system is modelled with using not all the UML diagrams. Because of no behavioural functionality, or the presentation of the most important characteristics, based on development/environment-specific model, class diagram, composite structure diagram, component diagram, and deployment diagram are used to describe the legacy HTML system, while other UML diagrams are not used.

8.2 Using Four UML Diagrams to Model HTML Legacy Systems

8.2.1 Not Having Dynamic Modelling

Because HTML uses cascading stylesheets to control the presentation of web pages, and basic special effects and interaction are provided by JavaScript, which adds a lot of power to basic HTML, it has its own characteristics which are different from other
The essential task of HTML is the service of Internet, and this is its biggest difference with other programming languages. HTML does not take on the responsibilities of calculation, real-time control, judgments and immediate response, batch disposal, database data management, man-made intelligence, etc.

HTML is a great tool to present the data under the designed format, especially with tables and frames. The nature of HTML means that it is the basic in web application. It does not focus on the complicated control or the computation.

Therefore HTML legacy systems do not need to be modelled with UML dynamic diagrams.

8.2.2 Static Modelling

UML 2.0 uses six diagrams to model the static parts of systems. The object and composite structure diagrams are low-level modelling diagrams, whilst others are high-level.

Among the UML diagrams, the class diagram is the most fundamental in modelling the structure of legacy systems. As mentioned in Section 7.2.1, the class diagram should be used to model HTML legacy systems, whilst the object diagram is unnecessary to be used.

Although the composite structure diagram is low-level, it is the description of internal structure of a class with different purposes from object diagram. The tables, forms, images, and frames have the different and complex structures in web usage, and their internal structure should be presented in detail in order to be more easily understood. So the composite structure diagram is used in modelling the internal structure of the classes of a HTML legacy system.

Because the most normal use for the package diagram is to organise use case diagrams and class diagrams although not limited to these, and HTML is a great tool to present the data under designed format, package diagram is not necessary to be used in modelling a HTML legacy system.
Chapter 8. Extracting UML Diagrams from HTML Legacy Systems

The component diagram shows the dependencies amongst software components, including the classifiers that specify them and the artifacts that implement them; these include source code files, binary code files, executable files, scripts and tables. The desired web page is realised through the correct usage of the tables, forms, images, and sounds in web pages in the designed format. So component diagram is used in modelling HTML legacy system.

The deployment diagram presents processors, devices, and the connections between them, which can comprise the run-time configuration of hardware nodes and the software components that run on those nodes. HTML uses the web servers and browsers to execute the web presentation and information exchange. Therefore the deployment diagram is used when modelling HTML legacy systems.

8.2.3 Using Four UML Diagrams to Model HTML Legacy Systems

Consequently, four UML diagrams: class, composite structure, component and deployment, are used to model HTML legacy systems.

8.3 Not Needing to Slice HTML Legacy Systems

According to the characteristics described in Section 5.2, the main characteristics of HTML legacy systems is the presentation of web and data format. They do not focus on data flow or control flow dependences. They are the great tool to present the data under the designed format, especially with tables and frames, and the common platform for the different data. They have few variables. Consequently, program slicing techniques are not suitable for analysing HTML legacy systems.

8.4 Modelling HTML Legacy Systems

8.4.1 Class Diagrams

Rule 1: Using four UML diagrams to model HTML legacy systems.
As mentioned in Section 5.4.3, there are three HTML development/environment-specific models: sequential, cyclical, and compositive link-based models.

Different models of HTML legacy systems have different structures, and different structures result in different structures of UML class diagrams when modelling legacy systems with UML. So there are three kinds of UML class diagrams corresponding to those three HTML development/environment-specific models.

UML composite structure diagram depicts the internal structure of a classifier (such as a class, component, or use case). It is most often used to show hidden internal details of a class. The three different HTML development/environment-specific models are composed of procedure and variable classes. The modelling of these two classes are the same, and not be influenced by the differences of the three HTML development/environment-specific models. So the applications of composite structure diagram to model the procedure and variable classes in the three different HTML development/environment-specific models are the same.

UML component diagram is used to describe the dependencies between various software components, including source code components, binary code components, and executable components.

A component represents a software entity in a system. Although HTML legacy system has three HTML development/environment-specific models, they have the same components. So the three different HTML development/environment-specific models are presented with the same component diagrams.

UML deployment diagram is the description of the processors, the devices, and the connections between the processors and the devices, and shows the hardware for the legacy HTML system, the software that is installed on that hardware, and the middleware used to connect the disparate machines to one another.

No matter how different the three HTML development/environment-specific models are, the processors, the devices, and the connections between the processors and the devices are the same. So the three different HTML development/environment-specific models have the same deployment diagrams.
Chapter 8. Extracting UML Diagrams from HTML Legacy Systems

The development/environment-specific models of HTML legacy system have three sequential, cyclical, and compositive link-based models. In order to simplify the research in the thesis and not lose the universality, the computing of development/environment-specific models of HTML legacy system will focus on compositive link-based model, the most complicated one among the three link-based models of HTML legacy systems. The other two models will not be computed. But the algorithms are similar.

Rule 2: Layering class diagrams.

In a HTML legacy system, one web page is regarded as the essential unit of the link-based model. The model entities are composed of the web systems. The structure of HTML legacy code is complicated and is displayed into different layers, which is represented with three different types of class diagrams.

The class diagram which presents the leaf web elements in source code is termed the leaf class diagram, indicated as LEAF-WEB-NAME.

The class diagram which presents the node web elements in source code is termed the node class diagram, indicated as NODE-WEB-NAME.

The class diagram which presents the root web element in source code is termed the root class diagram, indicated as ROOT-WEB-NAME.

One example of HTML code is in Figure 8.1, and its class diagram is presented in Figure 8.2.

```html
<HTML>
<HEAD>
<TITLE>Biography</TITLE>
</HEAD>
<BODY>
<H1><CENTER>The Jazz King’s Biography</CENTER></H1>
<TABLE BORDER="8" ALIGN="left">
  <TR>
    <TH>Record Name</TH>
    <TH>Year Produced</TH>
  </TR>
  <TR>
    <TD>Midnight Jazz</TD>
  </TR>
</TABLE>
</BODY>
</HTML>
```
The Jazz Kings burst onto the music scene in 1994 with their debut album entitled Midnight Jazz. Since then, the band has been gaining popularity through the United States. The band consists of five talented musicians who met in New Orleans in 1991. Since then, they have produced four albums and have toured throughout the United States and Europe. Although their albums are great, the band really must be seen live to be truly appreciated! The Jazz Kings play with a great deal of energy, and the audience is always dancing in unison by the end of the first song! Whether they are playing a small club, or a large stadium, the band is always at their best when on stage.

Rule 3: Starting from root web element.

The analysis of legacy HTML code starts from root web element. Because the head part describes the title and some introduction information, it is not included in the analysis model.

Leaf web elements are the end of the web link. Because of the single direction of the
web link from root web element to leaf web elements, every leaf web element does not contain the imported information. From leaf web elements, the link thread cannot be able to be discovered. Therefore, the leaf web elements are not good start point to analyse HTML code. Node web elements are difficult to find the linking clue comparing with root web element. Because they are imported by root web element and import leaf web elements, they share the difficulty of leaf web elements which are imported and increase the difficulty of root web element to track the link road. They are even worse to be the start point of analysing the HTML source code.

Root web element is the beginning of trailing web links. It is the starting point of the line of web pages. No matter how many web pages are behind the root web element, it is the essential unit to analyse the whole HTML code. It is the best start point of tracking the link.

Because one leaf web element is one functional module and it is linked in the node or root web elements, it is defined as one class in analysing node or root web elements that call the leaf web element.

For one leaf web element P, leaf web class is the class with respect to that leaf web element in analysing the webs linking it, indicated as CLASS LEAF-WEB-NAME.

The class which represents the node web element in source code is regarded as one class in node and root class diagrams which link that web, and it is termed node web class, indicated as NODE-WEB-NAME.

In one node web element, its contained leaf and node web elements are presented with leaf and node classes.

**Rule 4: Partitioning one web page into several blocks.**

The web page is used to exchange information through Internet. It is realised through the correct usage of the tags by the translation of the web browsers. It is composed of several parts, each one of which has the specific destination to describe the specific function, such as the title, the main description, the main data, the help and the additional information. And most of the presentation formats of data, which are tables, forms, images, sounds, photos and films, are displayed in the web pages. The data are presented with all manner of devices that display information on the Web: PCs with
Chapter 8. Extracting UML Diagrams from HTML Legacy Systems

graphics displays of varying resolution and color depths, cellular telephones, hand held devices, devices for speech for output and input, computers with high or low bandwidth, and so on. Different formats of data are presented with different description.

The web block is one part of one web page, which describes one specific function of that web page, indicated WB-NAME. One web page is one set of data that describes the main functionality. It is composed of several blocks. Those blocks are organised together with different functions to support the entire functionality of the web page. For example, one web page describes one definition UML. Then at least it has two web blocks: definition name UML and definition description, which are indicated as WB-UML and WB-UML-description.

The web blocks of one web page are former note, local header, local main, local footer, and additional note.

The former note is one web block in one web page that describes the top web, the main structure of the entire web system, the former web pages, the continuing web pages, etc. It is usually located at the upper web page. It is indicated as WBFN-NAME.

The local header is one web block in one web page that describes the main functions of the web page, the title of this web page, and the links to latter web pages. It is indicated as WBLH-NAME.

The local main is one web block in one web page that realises the main functions of the web page. It is indicated as WBLM-NAME.

The local footer is one web block in one web page that describes the explanation of the web realisation, the supplement of the web, the updating information, the contact styles, the information about the author, etc. It is usually located at the bottom of the web page. It is indicated as WBLF-NAME.

The additional note is one web block in one web page that provides the additional functions, such as helping, searching, statisticalising, recording, and advertising, and describes the links to latter web pages. It is usually located at the bottom or sides of the web page. It is indicated as WBAN-NAME.

In some cases, one web block has several parts. For example, local main of one web page can be divided into local main part 1, local main part 2, ..., and local main part n.
In the root web element, there are several blocks, and each block has different data descriptions.

**Rule 5: Presenting five operation classes.**

The HTML code has five groups operations: text, image, table, frame, and link. So the corresponding five kinds of classes are defined.

The text class of one HTML code PP is defined as the class that describes the text of the web element PP, indicated as Text Class. Its attributes include font, location, content, period, subscript, superscript, etc. Its operations include "to delete line", "to format", "to insert underline", "to quote", "to scroll", "to make the same style", etc (Figure 8.3).

<table>
<thead>
<tr>
<th>Class-Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text Class</td>
</tr>
<tr>
<td>Font: integer</td>
</tr>
<tr>
<td>Colour: string</td>
</tr>
<tr>
<td>Location: string</td>
</tr>
<tr>
<td>Content: string</td>
</tr>
<tr>
<td>to delete line</td>
</tr>
<tr>
<td>to insert underline</td>
</tr>
<tr>
<td>to format</td>
</tr>
<tr>
<td>to quote</td>
</tr>
<tr>
<td>to scroll</td>
</tr>
<tr>
<td>to make the same style</td>
</tr>
</tbody>
</table>

Figure 8.3: Text Class

<table>
<thead>
<tr>
<th>Class-Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Class</td>
</tr>
<tr>
<td>Location: string</td>
</tr>
<tr>
<td>Content: string</td>
</tr>
<tr>
<td>to create</td>
</tr>
<tr>
<td>to embed</td>
</tr>
<tr>
<td>to substitute with</td>
</tr>
<tr>
<td>to set spacer</td>
</tr>
</tbody>
</table>

Figure 8.4: Image Class
The image class of one HTML code PP is defined as the class that describes the image of the web element PP, indicated as Image Class. Its attributes include location, content, etc. Its operations include "to embed", "to substitute with", "to make the map", "to set spacer", "to create", etc (Figure 8.4).

The table class of one HTML code PP is defined as the class that describes the table of the web element PP, indicated as Table Class. Its attributes include location, table head, table body, table foot, caption, label, etc. Its operations include "to select", "to input", "to make button", "to make group", "to create text area", etc (Figure 8.5).

<table>
<thead>
<tr>
<th>Table Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class-Name</td>
</tr>
<tr>
<td>Location: string</td>
</tr>
<tr>
<td>Caption: string</td>
</tr>
<tr>
<td>Table Head: string</td>
</tr>
<tr>
<td>Table Body: string</td>
</tr>
<tr>
<td>Table Foot: string</td>
</tr>
<tr>
<td>Label: string</td>
</tr>
<tr>
<td>to select</td>
</tr>
<tr>
<td>to input</td>
</tr>
<tr>
<td>to make button</td>
</tr>
<tr>
<td>to make group</td>
</tr>
<tr>
<td>to create text area</td>
</tr>
<tr>
<td>to create volume</td>
</tr>
</tbody>
</table>

Figure 8.5: Table Class

The frame class of one HTML code PP is defined as the class that describes the frame of the web element PP, indicated as Frame Class. Its attributes include location, content, etc. Its operations include "to embed frame", "to substitute with", etc (Figure 8.6).

<table>
<thead>
<tr>
<th>Frame Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class-Name</td>
</tr>
<tr>
<td>Location: string</td>
</tr>
<tr>
<td>Content: string</td>
</tr>
<tr>
<td>to embed frame</td>
</tr>
</tbody>
</table>

155
Chapter 8. Extracting UML Diagrams from HTML Legacy Systems

The link class of one HTML code PP is defined as the class that describes the link of the web element PP, indicated as Link Class. It is the most important class among these classes because it leads to other web pages. Its attributes include location, base, content, layer, etc. Its operations include “to link”, etc (Figure 8.7).

Rule 6: Realising class diagram of one web block.

Each block of one web page contains all or part of those five classes, and each block is defined as one class diagram. See Figure 8.8.
Chapter 8. Extracting UML Diagrams from HTML Legacy Systems

Rule 7: Realising class diagram of one web element.

One web page is composed of one or more of the web blocks, and each web block is represented with one class diagram. Therefore class diagram of one web element is composed of one or more of those class diagrams.

Assume that the HTML source code PW has the layer \( j = n_0 \), \( n_j \) represents the maximum number of the \( s \)th web element of the \( j \)th layer, \( n_{jst} \) represents the maximum number of \( t \)th web block of the \( s \)th web element of the \( j \)th layer, \( PE_{jk} \) is the web element whose layer is \( j \), \( 1 \leq s \leq n_{js} \), whose web element is \( k \), \( 1 \leq k \leq n_{js} \), and \( PP_{jkl} \) is the web block whose layer is \( j \), \( 1 \leq s \leq n_{js} \), whose web element is \( k \), \( 1 \leq k \leq n_{js} \), \( 1 \leq l \leq n_{jst} \). The algorithm describing the class diagrams of web element of HTML code PW is:

\[
\text{For } j:=0 \text{ to } n_0 \text{ DO} \\
\quad \text{For } k:=1 \text{ to } n_{js} \text{ DO} \\
\quad \quad \text{For } l:=1 \text{ to } n_{jkl} \text{ DO} \\
\quad \quad \quad \text{TextClass}(PP_{jkl}); \quad \text{ImageClass}(PP_{jkl}); \quad \text{TableClass}(PP_{jkl}); \\
\quad \quad \quad \text{FrameClass}(PP_{jkl}); \quad \text{LinkClass}(PP_{jkl}); \quad \text{ClassDiagram}(PP_{jkl}); \\
\quad \quad \text{End-For} \quad \text{ClassDiagram}(PE_{jk}); \\
\quad \text{End-For} \quad \text{End-For} \\
\text{End-For} \quad \text{ClassDiagram}(PW); \quad \text{(8.1)}
\]

TextClass(), ImageClass(), TableClass(), FrameClass(), and LinkClass() are the functions to create the classes of source code, and ClassDiagram() is the function to draw the class diagram based on those classes.

When \( j=0 \), it is the class diagram of root web element, and others are the ones of leaf or node web elements.

Rule 8: Presenting class diagrams of HTML models.
Sequential link-based model is one link-based model in which the relationships between root web element and node web elements, node web elements, or node web elements and leaf web elements, are ordinal. The class diagram of sequential link-based model is described in Figure 8.9, 8.10, 8.11, and 8.12.

Figure 8.9: First Class Diagram of Sequential Link-Based Model

Figure 8.10: Second Class Diagram of Sequential Link-Based Model

Figure 8.11: Third Class Diagram of Sequential Link-Based Model
Chapter 8. Extracting UML Diagrams from HTML Legacy Systems

Figure 8.12: Fourth Class Diagram of Sequential Link-Based Model

Cyclical link-based model is one link-based model in which one or more relationships between root web element and node web elements, node web elements, or node web elements and leaf web elements, are loop. There is no ordinal relationship in this link-based model.

The class diagram of cyclical link-based model is described in Figure 8.13.

Figure 8.13: One Class Diagram of Cyclical Link-Based Model

Composite link-based model is one link-based model in which there exist two relationships of loop and ordinal at the same link-based model between root and node web elements, node web elements, or node web elements and leaf program elements.

The class diagram of compositive link-based model is described in Figure 8.14.
The class diagram of cyclical link-based model of HTML legacy system is simpler than composite link-based model. If one HTML legacy system has the condition (5.6) and it means that it has cyclical link-based model, then its class diagram is directly acquired from Figure 5.12 and Figure 8.13. So as to class diagrams of sequential link-based models of HTML legacy system.

Just like the description at the end of Section 7.3.1, the use of model for HTML legacy system improves the modelling efficiency.

### 8.4.2 Composite Structure Diagrams

*Rule 9: Presenting composite structure diagram of five classes.*

Composite structure diagram in UML 2.0 depicts the internal structure of a classifier (such as a class, component, or use case), including the interaction points of the classifier to other parts of the system. Composite structure diagrams, which are new to UML 2.0, focus on instances and their internal structure, providing examples of how the static architecture will achieve a requirement.

This diagram is most often used to show hidden internal details of a class, an object, or a component. It is composed of parts and connections.

The text class of HTML class diagram is composed of font, location, content, period,
Chapter 8. Extracting UML Diagrams from HTML Legacy Systems

subscript, superscript, deleted line, formatted text, inserted underline, quoted text, scrolled text, etc (Figure 8.15). The image class of HTML contains location, content, embedded image, text substituted with, made map, spacer, etc (Figure 8.16).

The table class includes location, table head, table body, table foot, caption, label, selection part, created form part, button, text area, created volume, etc (Figure 8.17). The frame class includes location, content, embedded frame, frame set, text substituted with, etc (Figure 8.18). The link class includes location, base, layer, linkURLs, etc (Figure 8.19).

![Composite Structure Diagram of Text Class of HTML](image1)

Figure 8.15: Composite Structure Diagram of Text Class of HTML

![Composite Structure Diagram of HTML Image Class](image2)

Figure 8.16: Composite Structure Diagram of HTML Image Class

![Composite Structure Diagram of Table Class](image3)

Figure 8.17: Composite Structure Diagram of Table Class

![Composite Structure Diagram of Frame Class of HTML](image4)

Figure 8.18: Composite Structure Diagram of Frame Class of HTML

161
Chapter 8. Extracting UML Diagrams from HTML Legacy Systems

<table>
<thead>
<tr>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
</tr>
<tr>
<td>Base</td>
</tr>
<tr>
<td>Layer</td>
</tr>
<tr>
<td>LinkURLs</td>
</tr>
</tbody>
</table>

Figure 8.19: Composite Structure Diagram of Link Class of HTML

Sometimes one class is composed of several equivalent parts, and they have the similar structure and the same functions. They are modelled with composite structure diagram in the same way.

Sometimes one part of one class is complicated and has many small pieces. In this case, the internal structure of that part of that class can be modelled with UML composite structure diagram as well.

8.4.3 Component Diagram

Rule 10: Presenting component diagram of HTML legacy system.

It is important to have a component diagram in modelling the legacy HTML system. A component diagram is defined by Object Management Group in 2001 as that shows the dependencies among software components, including the classifiers that specify them and the artifacts that implement them; such as source code files, binary code files, executable files, scripts and tables. A component diagram is used to describe the dependencies between various software components. A component represents a software entity in a system. A component is represented using a rectangular box, with two rectangles protruding from the left side. A Dependency is used to model the relationship between two components. The notation for a dependency relationship is a dotted arrow, pointing from a component to the component it depends on.

![Figure 8.20: One Component Diagram of Example In Figure 8.1](image-url)
Chapter 8. Extracting UML Diagrams from HTML Legacy Systems

A component diagram shows the dependencies among software components, including source code components, binary code components, and executable components. A software module may be represented as a component type. Some components exist at compile time, some exist at link time, and some exist at run time; some exist at more than one time. A compile-only component is one that is only meaningful at compile time; the run-time component in this case would be an executable program. A component diagram has only a type form, not an instance form. Legacy HTML system depicts the web page. It uses the wet browsers, the database, the application system, etc. The component diagram of example in Figure 8.1 is presented in Figure 8.20.

8.4.4 Deployment Diagram

Rule 11: Presenting deployment diagram of HTML legacy system.

A UML deployment diagram is the description of the processors, the devices, and the connections between the processors and the devices. It shows the hardware for the legacy HTML system, the software that is installed on that hardware, and the middleware used to connect the disparate machines to one another.

![Deployment Diagram](image)

Figure 8.21: Deployment Diagram of One Example In Figure 8.1
Chapter 8. Extracting UML Diagrams from HTML Legacy Systems

Legacy HTML system is the presentation of the web usage. It uses the web browsers, such as Internet Explorer and Netscape, which is allocated in the client, the database, such as FoxPro and Oracle, which is stored in the database server, and the application system, such as the search engine and email, which is deposited in the user client. The deployment diagram of example in Figure 8.1 is presented in Figure 8.21.

8.4.5 Application of HTML Rules

Based on the link-based model of HTML legacy system, four UML diagrams is able to be presented through the above eleven rules. All the discussion above is in the most complex situation. In some cases, one HTML legacy system may not link node web elements, and it only has leaf web elements; one web page may only contain parts of five local web blocks; one web page may only contain several paragraphs of text description; or even it is composed of only one picture. Then those eleven rules do not need to be followed one by one.

So, when one web legacy system has one of the following characters, it is unnecessary to follow those rules:

- **C1**: It is composed of one root element and not more than three leaf web elements.
- **C2**: One web element only has one block.
- **C3**: One web element is composed of several paragraphs of text.
- **C4**: One web element only has one image, such as one picture or one photo.
- **C5**: One of five blocks of one web element has only one class among text class, image class, image class, table class and frame class.

As mentioned in this chapter, Rule 1 to 8 describes the realisation of class diagrams of HTML legacy system. When C1 occurs, which means that this system only has one root web element and not more than three leaf web elements, its class diagram is unnecessary to be realised (Figure 8.22). When C2 occurs, which means that the web element being discussed only has one block, its class diagram does not need to be realised and Rule 4 to 6 are not used. When C3 happens, which means that the web element only contains text format, its class diagram and composite structure diagram are unnecessary to be realised and Rule 4 to 7 are not used. When C4 happens, which means that the web element is composed of one image, its class diagram and composite
structure diagram are unnecessary to be realised. When C5 occurs, which means that the
web element only has one block, its class diagram does not need to be realised.

Consequently, when conditions C1 to C5 occur, the complexity of modelling HTML
legacy system at that point is greatly reduced.

![HTML Rules Application](image)

Figure 8.22: HTML Rules Application
Chapter 8. Extracting UML Diagrams from HTML Legacy Systems

8.5 Summary

In this chapter, under the support of development/environment-specific model in Section 5.4, four UML static diagrams are used to describe HTML legacy system. Because of no behavioural functionality, other UML diagrams are not used.

Eleven rules are presented to realise those four UML diagrams. Based on link-based models, class diagram is obtained from HTML legacy system. The statements of in HTML code are composed of five groups. Three different kinds of web link layers are distinguished—root web element, leaf web element and node program element. Link-based model is indicated. It is composed of nodes and lines. The sequence of link-based model PG is upper-to-bottom. The web the first node represents imports the webs the next nodes represent. The sequence of the next nodes is the sequence being imported in the first web. In response to three web layers, three class diagram—leaf, node, and root class diagram, are introduced. The HTML code has five groups operations: text, image, table, frame, and link. So the corresponding five kinds of classes are defined. Based on one algorithm, class diagrams of HTML legacy system are realised. Composite structure diagrams are composed of parts of those five classes—text, image, table, frame, and link class. The component diagram and deployment diagram of legacy HTML system are presented as well.
Chapter 9
Extracting UML Diagrams from SQL Legacy Systems

9.1 Introduction

SQL is a full-featured relational database management system that offers a variety of administrative tools to ease the burdens of database development, maintenance and administration. T-SQL is Microsoft's proprietary extension to SQL and includes additional functionality designed to support Microsoft SQL server. It is common for large-scale database to use SQL to facilitate the database user and the administrator interactions. This language offers a flexible interface for databases of all shapes and sizes.

Because SQL has the operations of the data flow and control flow dependences, program slicing techniques are suitable for analysing SQL legacy system.

A SQL legacy system is modelled with not all the UML diagrams. Based on development/environment-specific model in Section 6.4, class diagram and composite structure diagram are used to describe the static part of SQL legacy system and internal structures of classes, and activity diagram describes the dynamic part of legacy system. Other UML diagrams are not used.

9.2 Using Three UML Diagrams to Model SQL Legacy Systems

9.2.1 Static Modelling

UML 2.0 uses six diagrams to model the static parts of systems. Because the class
diagram is the most fundamental in modelling the structure of legacy systems, so class
diagram is used to model SQL legacy systems. But object diagram is not used.

The composite structure diagram is low-level and is the description of the internal
structure of a class with different purposes to from the object diagram. It can be used to
model the complication of a class in detail. SQL is a full-featured relational database
management system that offers a variety of administrative tools to ease the burdens of
database development, maintenance and administration. This language offers a flexible
interface for databases of all shapes and sizes. Because every database has its own
specific structure, composite structure diagram should be used in modelling the internal
structure of a class of SQL legacy system.

The component diagram is used at a high level. After class diagram is used in modelling
a COBOL legacy system in high level, component diagram is pleonastic and redundant
in modelling SQL code. So the component diagram is not used.

The package diagram is adopted at too high a level in modelling a SQL legacy system,
after the class diagram is used, it is not necessary to use the package diagram.

The deployment diagram is the run-time configuration of hardware nodes and the
software components that run on those nodes. It is too high-level for analysing SQL
legacy systems from the static and structural points of view. Therefore, the deployment
diagram is not used when modelling SQL legacy systems.

9.2.2 Dynamic Modelling

UML 2.0 uses seven diagrams to model the dynamic parts of systems, timing diagram
and interaction overview diagram of which are new in UML 2.0.

A sequence diagram presents the interactions between objects that achieve a result at a
high level. It describes how groups of objects collaborate in presenting certain system
behaviours. Because SQL system mainly focuses on the databases and its main objects
are databases, and SQL system mainly creates or deletes the databases and it has few
messages between those databases like other objects in COBOL, BASIC and
FORTRAN languages, the sequence diagram that describes the objects and messages
between them is not used.
Chapter 9. Extracting UML Diagrams from SQL Legacy System

A communication diagram is used to model interactions between objects, and objects interact by invoking messages on each other. The class is the abstraction of the common characters of the object group. Most of the important characteristics of the object are reflected on the responding class. Composite structure diagram has been used to model the internal structure of the classes in SQL legacy system. It is not necessary to use the collaboration diagram.

An activity diagram describes a flowchart showing the flow of control from activity to activity at a high level. A SQL system has the control flow about the databases. So the activity diagram is used in modelling SQL legacy systems.

Use case diagram shows the relationships among actors and use cases within a system in terms of the UML specification at a high level. It is a set of use cases and actors and their relationships. A use case is a requirement that the users of the system want the system to do. Because the modling of SQL legacy system focuses on the modelling of the databases and it does not need to present the requirements of the users, the use case diagram is not used in modelling SQL legacy systems.

An interaction overview diagram shows the control flow within a system at a low level. SQL system has the control flow about the databases. However, SQL system mainly focuses on the databases and its main objects are databases, and SQL system mainly creates, deletes or changes the databases and it has several databases unlike so many objects in COBOL-like or BASIC-like languages. The control flow in SQL systems is modelled by activity diagram, and it is not necessary to use the interaction overview diagram to model SQL legacy systems.

A state machine diagram shows the sequence of states that an object goes through during its life cycle. Because a SQL system mainly focuses on the databases and its main objects are databases, and a SQL system mainly creates, deletes or changes the databases, the state machine diagram is not used in modelling SQL legacy system.

A timing diagram explores the behaviours of one or more objects throughout a given period of time. It depicts the change of the state or the condition of a classifier instance or role over time. Because a timing diagram is suitable for the description of embedded system, it is not used in modelling SQL legacy systems.
9.2.3 Modelling with Three UML Diagrams

Consequently, three UML diagrams: class, composite structure and activity, are used to model SQL legacy systems.

9.3 Modelling SQL Legacy Systems

9.3.1 Class Diagrams

Rule 1: Using three UML diagrams to model SQL legacy systems.

According to the description in Section 6.4.3, there are three SQL development/environment-specific models: association, generation, and composition database-based models.

Generation database-based model is a database graph in which the relationships between databases are all generations. Association database-based model is a database graph in which the relationships between databases are all associations. Composition database-based model is a database graph in which the relationships between databases are associations and generations existing at the same database graph. There are three states: two databases generate only one database; one database generates more than one database; two or more databases combine together to generate one or more databases.

Different models of programs in SQL legacy system have different structures, and different structures result in different UML class diagrams when modelling SQL legacy system with UML (Figure 9.1). So there are three types of UML class diagrams corresponding to those three SQL development/environment-specific models.

A composite structure diagram is the description of the internal structure of a class, component, or use case, including the interaction points of the classifier to other parts of a system. Although SQL legacy systems have three SQL development/environment-specific models, all of them are composed of the procedure and database classes. Both of those two types of classes are the same in those three different SQL development/environment-specific models. Thus those three models have the same composite structure diagrams.
A SQL legacy system has behavioural aspects, and it is modelled with the activity diagram. A activity diagram is an ongoing monatomic execution within a state machine. Activities ultimately result in some action, which is made up of executable atomic computations that result in a change in state of the system or the return of a value. Actions encompass calling another operation, sending a signal, creating or destroying an object, or some pure computation. A activity diagram models the sequential and possibly concurrent steps in a computational process. It also models the flow of an object as it moves from state to state at different points in the flow of control.

The three SQL development/environment-specific models present the different behavioural parts. Therefore three different activity diagrams are presented based on the three SQL development/environment-specific models.

The development/environment-specific models of SQL legacy systems are generation, association and composition database-based models. In order to simplify the research in this thesis and not lose the universality, the process described in this chapter of computing development/environment-specific models of SQL legacy systems will focus on composition database-based model, the most complicated one amongst the three database-based models of SQL legacy systems. The other two models of SQL legacy system will not be computed. But the algorithms are similar.

*Rule 2: Slicing SQL legacy system.*
Chapter 9. Extracting UML Diagrams from SQL Legacy System

As mentioned in Section 6.2, SQL is the common platform for the different relational database, leads to the common base for the usage and development of database, and provides the common interface of the different databases. The description of database is one important task of SQL. And at the same time, SQL has procedures, including many executable procedures. Those procedures provide the control and searching functions of the databases. SQL legacy system has some variables and executes the tasks of judging, looping, and computing.

Therefore, program slicing techniques are similar in analysing SQL legacy system as slicing COBOL legacy system.

**Rule 3: Layering class diagrams.**

The class diagram which presents the root program elements in source code is termed the root class diagram, indicated as ROOT-PROCEDURE-NAME. The class diagram which presents the node program elements in source code is termed the node class diagram, indicated as NODE-PROCEDURE-NAME. The class diagram which presents the leaf program elements in source code is termed the leaf class diagram, indicated as LEAF-PROCEDURE-NAME.

**Rule 4: Generating class diagrams.**

The databases that are created, changed, deleted and used to create the other databases are defined as the classes, which are indicated as DB-CLASS-name (Figure 9.2).

<table>
<thead>
<tr>
<th>Database Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB-CLASS-name</td>
</tr>
<tr>
<td>ID: int;</td>
</tr>
<tr>
<td>Volumes: int;</td>
</tr>
<tr>
<td>Rows: int;</td>
</tr>
<tr>
<td>To create();</td>
</tr>
<tr>
<td>To change();</td>
</tr>
<tr>
<td>To delete();</td>
</tr>
<tr>
<td>To use()</td>
</tr>
</tbody>
</table>

Figure 9.2: Database Class
Chapter 9. Extracting UML Diagrams from SQL Legacy System

The class that represents the leaf procedure element in source code is termed the leaf class, indicated as LEAF-CLASS-procedurename. The class that represents node procedure element in source code is termed the node class, indicated as NODE-CLASS-procedurename (Figure 9.3 and 9.4).

<table>
<thead>
<tr>
<th>Leaf Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEAF-CLASS-procedurename</td>
</tr>
<tr>
<td>ID: int;</td>
</tr>
<tr>
<td>layer: int;</td>
</tr>
<tr>
<td>To be used();</td>
</tr>
<tr>
<td>To use()</td>
</tr>
</tbody>
</table>

Figure 9.3: Leaf Class

<table>
<thead>
<tr>
<th>Node Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE-CLASS-procedurename</td>
</tr>
<tr>
<td>ID: int;</td>
</tr>
<tr>
<td>layer: int;</td>
</tr>
<tr>
<td>To be used();</td>
</tr>
<tr>
<td>To use()</td>
</tr>
</tbody>
</table>

Figure 9.4: Node Class

Based on the classes and their relationships, after all the leaf class diagrams are extracted from the leaf program elements, the node program elements start to be analysed. At last the root class diagram is realised.

**Rule 5: Presenting class diagrams of database-based models.**

A generation database-based model is one database graph in which the relationships between databases are all generations. The class diagram of a generation database-based model is described in Figure 9.5.

An association database-based model is one database graph in which the relationships between databases are all associations.

The class diagram of an association database-based model is described in Figure 9.6.
Chapter 9. Extracting UML Diagrams from SQL Legacy System

Figure 9.5: Generation Database-Based Model

Figure 9.6: Association Database-Based Model

Figure 9.7: First Example of Composition Database-Based Model
Chapter 9. Extracting UML Diagrams from SQL Legacy System

A composition database-based model is one database graph in which the relationships between databases are associations and generations existing at the same database graph. There are three states: two databases generate only one database; one database generates more than one database; two or more databases combine together to generate one or more databases. The class diagram of a composition database-based model is described in Figure 9.7, 9.8, 9.9, and 9.10.

Figure 9.8: Second Example of Composition Database-Based Model

Figure 9.9: Third Example of Composition Database-Based Model
As mentioned before, the class diagram of a composition database-based model of SQL legacy systems is more complex than those of generation and association database-based models. Based on the formula (6.1) to (6.7), it is easy to judge the model conditions. Therefore when one SQL legacy system has association or generation database-based model, it is of greatly less complexity than the one which has a composition database-based model. Consequently the model usage of SQL legacy systems is efficient.

9.3.2 Composite Structure Diagrams

Rule 6: Presenting composite structure diagrams of SQL legacy system.

A composite structure diagram describes the relationships between elements that work together within a classifier. It shows parts and connectors.

The composite structure diagram is used to show the runtime architectures of any kind of classifier. The database class of legacy SQL code is composed of name, table, size, etc (Figure 9.11). The procedure class is composed of name, used database, created database, deleted database, changed database, etc (Figure 9.12).
Chapter 9. Extracting UML Diagrams from SQL Legacy System

### Figure 9.11: Composite Structure Diagram of Database Class

<table>
<thead>
<tr>
<th>Database Name</th>
<th>Table</th>
<th>Size</th>
<th>Row</th>
<th>Volume</th>
<th>SubDatabase</th>
</tr>
</thead>
</table>

### Figure 9.12: Composite Structure Diagram of Procedure Class

<table>
<thead>
<tr>
<th>Procedure Name</th>
<th>UsedDatabase</th>
<th>CreatedDatabase</th>
<th>DeletedDatabase</th>
<th>ChangedDatabase</th>
</tr>
</thead>
</table>

9.3.3 Activity Diagrams

**Rule 7: Layering activity diagrams.**

The UML activity diagram describes the dynamic aspects of systems. It is essentially a flowchart, showing flow of control from activity to activity. An activity is an ongoing nonatomic execution within a state machine. Activities ultimately result in some action, which is made up of executable atomic computations that result in a change in state of the system or the return of a value. Actions encompass calling another operation, sending a signal, creating or destroying an object, or some pure computation. It models the sequential and possibly concurrent steps in a computational process. It also models the flow of an object as it moves from state to state at different points in the flow of control. Activity diagrams may stand alone to visualise, specify, construct, and document the dynamics of a society of objects, or they may be used to model the follow of control of an operation (Figure 9.13).

The operations of legacy code are classified into different layers, especially in legacy code containing node and leaf program elements. The analysis of legacy code starts from leaf program elements.

The activity diagram which presents the leaf program elements in SQL source code is termed the leaf activity diagram, indicated as LEAF-ACTIVITY-NAME.

The activity diagram which presents the node program elements in SQL source code is
Chapter 9. Extracting UML Diagrams from SQL Legacy System

termed the node activity diagram, indicated as NODE-ACTIVITY-NAME.

Figure 9.13: Activity Diagram of Getting Integer Part of Number

The activity diagram which presents the root program elements in SQL source code is termed the root activity diagram, indicated as ROOT-ACTIVITY-NAME.

**Rule 8: Presenting control graph.**

The activity group is the SQL code operations that execute the data and databases with the change, store, and display of information. It is indicated as ActionG(). It is the joint set of data group, procedure group, and safety group, which is

\[
\text{ActionG()} = \text{GroupD()} \cup \text{GroupP()} \cup \text{GroupS()} \tag{9.1}
\]

Any arbitrary element \( s, s \in \text{ActionG()} \), is termed one action of that legacy code.

Let \( s_i \) be one element of control group, \( s_i \in \text{GroupC()} \), and its control layer is indicated as \( \text{CL}(s_i) \). The first control operation in the legacy code \( s_i \) has the control layer 1,
and the parallel control operations have the same control layer 1; the first control operation \( s_i \) that is inserted in \( s_i \) has the control layer 2,

\[ CL(s_i) = 1, \]

and the parallel of \( s_j \) has control layer 2, so on.

If one procedure has no control operation, then that procedure is termed the empty-control procedure, indicated as PROCEDURE-C0-NAME.

The control graph is one graph that describes the control flow with the nodes \( s_i, s_i \in \text{GroupCO} \), which represent the control actions. Every node has one control action and unique control layer. The presentation of the empty-control procedure is defined as one node with the indication PROCEDURE-C0.

**Rule 9: Presenting activity array.**

Every node in the control graph of legacy code is termed program activity, indicated as \( AT \). Each activity \( AT_i \) represents the operations with the selection, judgment, and management under one or more conditions. The empty-control procedure is presented as the name with the indication PROCEDURE-C0.

Let \( n \) be the control layer of one procedure of legacy code, and \( t_i \) be the number of control action of \( i^{th} \) control layer, \( 0 < i < n \). The activity array is composed of all the program activities of that program based on the domain knowledge. It is indicated as \( AT(i, j) \).

**Rule 10: Realising activity diagrams.**

The leaf activity diagram of SQL legacy code is the graphical presentation of activity array of that leaf procedure with the presentation of the control conditions and the control graph. Let \( n \) be the control layer of one procedure of legacy code, \( t_i \) be the number of control action of \( i^{th} \) control layer, \( 0 < i < n \), and \( CD(i, j) \) be the control conditions. Then the algorithm of computing the leaf activity diagram is

\[
\text{For } i = 0 \text{ to } (n-1) \\
\text{For } j = 1 \text{ to } t_i
\]
Chapter 9. Extracting UML Diagrams from SQL Legacy System

After all the leaf activity diagrams are realised, the leaf program elements in node and root program elements are transformed into one activity. Some of node program elements become leaf program elements with the replacement of the activity. With the usage of the strategy (9.2), their activity diagrams are produced. And the loop continues until the root program element becomes the leaf. So all the activity diagrams are realised.

Let \( m \) be the procedure layer of legacy SQL code, \( u_i \) be the number of the procedures of the \( i^{th} \) layer, \( 0 < i < (m+1) \), \( AD(j, k) \) be the \( j^{th} \) procedure of the \( i^{th} \) layer. The algorithm of the realisation of activity diagrams of SQL legacy code is

\[
\text{For } i=0 \text{ to } (n-1) \\
\quad \text{For } j=1 \text{ to } u_i \\
\quad \quad AD((n-i), j) \\
\quad \text{End-For} \\
\text{End-For} \quad (9.3)
\]

The UML activity diagrams model the complex operation, business rules, business process, and software process [15, 49]. They are used to model the dynamic aspect of legacy system, which may involve the activity of any kind of abstraction in any view of a system's architecture. Activity diagram models a workflow and an operation of a system. An activity diagram is attached to any modelling element for the purpose of visualising, specifying, constructing, and documenting that element's behaviour[62].

It is possible that activities can occur in parallel that are regarded as forks and joins. A fork should have a corresponding join. One fork has one entry transition and one join has one exit transition. Activity diagrams allow for a great deal of freedom. They use the right level of detail to describe the system functionality.

It is important to limit the level of complexity of each activity diagram. If there are more than three possible paths (alternate or exceptional), it is necessary to use additional activity diagrams to promote understanding. It is also necessary to use
additional activity diagrams if the processing requires specific data elements.

Rule 11: Presenting activity diagrams of database-based models.

Association database-based model is one database graph in which the relationships between databases are all associations (Figure 9.14). Generation database-based model is one database graph in which the relationships between databases are all generations (Figure 9.15). Composition database-based model is one database graph in which the relationships between databases are associations and generations existing at the same database graph. There are at least three states: two databases generate only one database; one database generates more than one database; two or more databases combine together to generate one or more databases (Figure 9.16, 9.17, 9.18, and 9.19).
Chapter 9. Extracting UML Diagrams from SQL Legacy System

Figure 9.16: First Example of Composition Database-Based Model

Figure 9.17: Second Example of Composition Database-Based Model

Figure 9.18: Third Example of Composition Database-Based Model
Chapter 9. Extracting UML Diagrams from SQL Legacy System

As mentioned in Section 9.3.1, activity diagram of composition database-based model of SQL legacy system is more complex than generation and association database-based models. Under the help of the formula (6.1) to (6.7), after the judgement of the model conditions of SQL legacy system, when one SQL legacy system has association or generation database-based model, it is of greatly less complexity than the one which has composition database-based model. It saves the time and reduces the costs and the complexity. Therefore the model usage of SQL legacy system improves the efficiency of software evolution.

9.3.4 Application of SQL Rules

As mentioned in this chapter, through Rule 1 to 11, three UML diagrams are realised based on database-based model of SQL legacy system. If one SQL legacy system has one or two databases, has not or only has one procedure, or its one procedure has not more than thirty lines, some rules of those eleven rules may be passed over. So, when one SQL legacy system has one of the following characters, it is unnecessary to follow
Chapter 9. Extracting UML Diagrams from SQL Legacy System

those rules:

C1: It has only one database.
C2: The database has not more than four volumes.
C3: The procedure has not more than five variables.
C4: The procedure has not more than thirty lines.

SQL Rule 1 to 5 presents the realisation of class diagrams of SQL legacy system. When C1 occurs, which means that SQL legacy system only has one database, its class diagram is unnecessary to be realised and Rule 4 and 5 are not used (Figure 9.20).

When C2 occurs, which means that one database of SQL legacy system has not more than four volume, its composite structure diagram does not need to be realised and Rule 6 are not used. When C3 happens, which means that its procedure only contains no more than five variables, its activity diagram is unnecessary to be realised and Rule 7 to 11 are not used. When C4 happens, which means that its procedure has not more than thirty lines, its activity diagram is unnecessary to be realised and Rule 7 to 11 are not used.

Consequently, when conditions C1 to C4 occur, the complexity of modelling SQL legacy system at that point is greatly reduced.

Figure 9.20: SQL Rules Application
9.4 Summary

In this chapter, eleven rules are presented to discuss the evolution of SQL legacy system. Based on development/environment-specific model in Section 6.4, class diagram and composite structure diagram are used to describe the static part of SQL legacy system and its internal structure, and activity diagram describes the dynamic part of legacy system. Because of the presentation of the most important characteristics or pleonasm and redundancy, other UML diagrams are not used.

Because SQL also has the operations of the data flow and control flow dependences, program slicing techniques are suitable for analysing SQL legacy system.

Based on database-based models, class diagram is obtained from SQL legacy system. Four different kinds of program elements are distinguished: root program elements; leaf program elements; node program elements; and isolated program elements. Procedures are layered based on the calling relationships. They are classified into three groups: root, node, and leaf procedure elements. And class diagrams of SQL legacy system are divided into three root, node, and leaf class diagrams. The databases that are used in SQL code are regarded as the classes, and the procedures are defined as the node or leaf class. With the help of the procedure layers and graph, three class diagrams are produced. Composite structure diagrams are presented as well to describe the internal structure of the complex classes in generated class diagrams. Activity diagram shows the flow from activity to activity. One code operation that belongs to the group of exchanging information is defined as one action. The control operations that produce the judgment form the control graph. Those actions that execute one function are regarded as one activity, and the legacy SQL code produces the activity array. With the help of the control graph, the leaf activity diagram is produced with the nodes that represent the activity. Then, the node and root activity diagrams are realised.
Chapter 10

Tool and Experiments

10.1 Introduction

In this chapter, three cases are studied for domain-specific legacy systems in order to describe how the theoretical and technical aspects of the proposed method in the thesis were implemented. Those legacy systems are parsed at first, then the models are searched out, and based on the models UML diagrams are acquired. Those case studies also evaluate the proposed method in this thesis.

10.2 Tool Design

10.2.1 One Unified System----SEASAT

In order to demonstrate the evolution process, the proposed approach has been applied to domain-specific legacy systems. The models of COBOL legacy system correspond to the judgement conditions. In this thesis, one unified system is proposed, which is termed SEASAT (Software Evolution for domain-Specific legacy system). It opens one source code file, parses it, searches the models, computes the slices, and displays UML diagram. It is designed to demonstrate the proposed approach in this thesis.

SEASAT integrates all technical supports into a systematic method for software evolution of domain-specific legacy systems. Although the tool does not show the working processes step by step of 29 rules presented in Chapter 7, 11 rules in Chapter 8, and 11 rules in Chapter 9, it clearly describes the main thoughts of the proposed approach in this thesis.

10.2.2 Detail Design Description

The unified system SEASAT is mainly composed of six parts, which are named as
Chapter 10. Tool and Experiments

"OpenSourceFile", "ParseSourceCode", "SearchMode", "SliceCode", "DisplayUMLDiagram" and "Exit". Its main working flow is presented in Figure 10.1.

![Tool Architecture Diagram](image)

Figure 10.1: Tool Architecture

The first part of the unified system SEASAT is to open one source code file. One textual file containing source code is regarded as the input. The demonstration interface composes three text blanks. The text file is displayed in the first blank, and the displaying message is presented in the second blank. If the source file is successfully opened, one message which is "Source Code Displayed" is shown in the second blank. Otherwise, the system exits.

The second part is to parse the source code. The system SEASAT parses the input source code. In the second blank of the demonstration interface, all the candidates of the statements containing the procedures, links or databases in input legacy system are displayed. Then SEASAT computes the model entities and their relationships from those candidates and display them in the third blank of the demonstration interface. At last, it removes the redundancy of those model entities and shows the pure model entities of input legacy system in the first blank.

The third part is to search the model. Based on the model conditions of COBOL legacy systems (4.9) to (4.17) in Chapter 4, ones of HTML legacy systems (5.3) to (5.8) in Chapter 5, and ones of SQL legacy systems (6.1) to (6.7) in Chapter 6, respectively, the
system SEASAT acquires the model of input legacy system. In the new demonstration interface which contains one text blank and one figure blank, the model entities of input legacy system are shown in the text blank, and its model is presented in the figure blank.

The fourth part is to slice code. SEASAT slices the input source code based on the slicing techniques described in Section 6.3.1. The demonstration interface contains three blanks. SEASAT displays all the used variables in the analysed source code from the statements of input source code, and it removes the repetitions of those variables and shows all the variables in the second blank. Also SEASAT computes the systematic calls of input source code and displays them in third blank of the demonstration interface.

The fifth part is to display UML diagrams. SEASAT computes the pseudo and real classes and their relationships based on the program slicing results, and it stores them in the database. Then in the new demonstration interface, the class diagrams of input legacy system are displayed. Based on the rules in Chapter 7, 8, and 9, the selected UML diagrams of legacy system are shown.

The sixth part is to exit the system SEASAT.

Some figures display the main working interfaces using part source code as shown in Appendix A, such as tool architecture in Figure 10.1, one interface in Figure 10.2, and one working interface in Figure 10.5. In practice, because of the complexity of software evolution of entire domain-specific legacy systems, SEASAT does not attempt to be perfect. It mainly demonstrates the original ideas of proposed approach in this thesis.

As the fully implemented systems are highly complex, only parts of COBOL legacy system are presented here with the usage of tool.

10.3 Case Study of COBOL Legacy System

10.3.1 COBOL Legacy Code—Manager Application

One legacy software system named Manager Application in COBOL is presented and will be modelled with selected number of UML diagrams (Appendix A). This program
Chapter 10. Tool and Experiments

is designed to create the new manager file for the application that is performed once a week on Friday. It is executed on a Window client, and communicates with a local Unix server and a remote Window server. This legacy software is licensed SALES PRO TECHNOLOGIES, INC. Its name is dmFileProc. Its running environment is Window.

10.3.2 Parsing COBOL Legacy Code

According to the description of Section 4.4.1.1, in legacy COBOL system Manager Application System, the root program element is Initialisation section. The leaf program elements are initialise-file-copy-tables section, all-tasks-complete-process section, and Get-Download-File-Size section. Others are node program elements (see Table 10.1, 10.2, and 10.3). There is no isolated program element in legacy system Message Application (Figure 10.2).

Table 10.1: Table of Program Elements In Source Code

<table>
<thead>
<tr>
<th>Procedure Number</th>
<th>Procedure Name</th>
<th>Program Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initialisation</td>
<td>root</td>
</tr>
<tr>
<td>2</td>
<td>Initialise-File-Copy-Tables</td>
<td>leaf</td>
</tr>
<tr>
<td>3</td>
<td>Task-1-Process</td>
<td>node</td>
</tr>
<tr>
<td>4</td>
<td>Task-2-Process</td>
<td>node</td>
</tr>
<tr>
<td>5</td>
<td>Task-3-Process</td>
<td>node</td>
</tr>
<tr>
<td>6</td>
<td>Task-4-Process</td>
<td>node</td>
</tr>
<tr>
<td>7</td>
<td>Task-5-Process</td>
<td>node</td>
</tr>
<tr>
<td>8</td>
<td>Task-6-Process</td>
<td>node</td>
</tr>
<tr>
<td>9</td>
<td>Task-7-Process</td>
<td>node</td>
</tr>
<tr>
<td>10</td>
<td>All-Tasks-Complete-Process</td>
<td>leaf</td>
</tr>
<tr>
<td>11</td>
<td>Get-Download-File-Size</td>
<td>leaf</td>
</tr>
<tr>
<td>12</td>
<td>Setup-Unix-Server-File-Transfer-Session</td>
<td>node</td>
</tr>
<tr>
<td>13</td>
<td>Establish-Data-Port-Connection</td>
<td>node</td>
</tr>
<tr>
<td>14</td>
<td>Establish-Unix-Data-Port-Connection</td>
<td>node</td>
</tr>
</tbody>
</table>

Table 10.2: Table of Procedure Calling Relationships and Procedure Layers

<table>
<thead>
<tr>
<th>Procedure Number</th>
<th>Calling Procedures</th>
<th>Procedure Layer PL( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2, 3, 4, 5, 6, 7, 8, 9, 10</td>
<td>10</td>
</tr>
</tbody>
</table>
Chapter 10. Tool and Experiments

<table>
<thead>
<tr>
<th>Procedure Layer PL()</th>
<th>Procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2, 10, 11</td>
</tr>
<tr>
<td>1</td>
<td>3, 5, 6, 7, 9, 13, 14</td>
</tr>
<tr>
<td>2</td>
<td>8, 12</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 10.3: Procedure Layers With Procedures

Figure 10.2: Opening Source Code
Chapter 10. Tool and Experiments

Then as mentioned in Section 4.4, the set \( \text{SAP}\{ \} \) is the set of all procedures in COBOL legacy system Manager Application System,

\[
\text{SAP}\{} = \{ \text{PP}_i \mid \text{PP}_i \in \text{P} \} = \{ \text{Initialisation, Initialise-File-Copy-Tables, Task-1-Process, Task-2-Process, Task-3-Process, Task-4-Process, Task-5-Process, Task-6-Process, Task-7-Process, All-Tasks-Complete-Process, Get-Download-File-Size, Setup-Unix-Server-File-Transfer-Session, Establish-Data-Port-Connection, Establish-Unix-Data-Port-Connection } \},
\]

In order to simplify the presentation of the procedures, the names of the procedures are replaced with the number in Table 10.1, that is,

\[
\text{SAP}\{} = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 \}
\]

The set \( \text{SOOP}\{ \} \) is the set of the one to one procedures in COBOL legacy system Manager Application System,

\[
\text{SOOP}\{} = \{( \text{PP}_i, \text{PP}_j ) \mid (\text{PP}_i \gg \text{PP}_j) \land ((\forall k, k \neq j) \Rightarrow (\text{PP}_i \gg \gg \text{PP}_k)) \land ((\forall m, m \neq i) \Rightarrow (\text{PP}_m \gg \gg \text{PP}_j)) \}
\]

\[
= \emptyset
\]

Because

\[
\text{SOOP}\{} \neq \text{SAP}\{},
\]

so the procedure-based model of COBOL legacy system Manager Application System is not linear.

The set \( \text{SOMP}\{ \} \) is the set of the one to many procedures in COBOL legacy system Manager Application System,

\[
\text{SOMP}\{} = \{( \text{PP}_i, \text{PP}_j ) \mid (\text{PP}_i \gg \gg \text{PP}_j) \land ((\exists k, k \neq j) \Rightarrow (\text{PP}_i \gg \gg \text{PP}_k)) \land ((\forall m, m \neq i) \Rightarrow (\text{PP}_m \gg \gg \text{PP}_j)) \land ((\forall m, m \neq i) \Rightarrow (\text{PP}_m \gg \gg \text{PP}_k)) \}
\]

\[
= \{1, 4, 8, 12 \}
\]
Chapter 10. Tool and Experiments

Because

$$\text{SMOP} \neq \text{SAP},$$

so the procedure-based model of COBOL legacy system Manager Application System is not branch.

The set SMOP{} is the set of the many to one procedures in COBOL legacy system Manager Application System,

$$\text{SMOP} = \{ (PP_i, PP_j) | (PP_i \triangleright PP_j)$$

$$\text{AND } ((\exists k, k \neq i) \Rightarrow (PP_k \triangleright PP_j))$$

$$\text{AND } ((\forall m, m \neq j) \Rightarrow (PP_i \triangleright PP_m))$$

$$\text{AND } ((\forall m, m \neq j) \Rightarrow (PP_k \triangleright PP_m))\}$$

$$= \{2, 10\}$$

Because

$$\text{SMOP} \neq \text{SAP} \setminus \{1\},$$

so the procedure-based model of COBOL legacy system Manager Application System is not joint.

The set SSP{} is the set containing the many to many procedures in COBOL legacy system Manager Application System,

$$\text{SSP} = \{ PP_k | ((\exists i, j, k, i \neq j, j \neq k, i \neq k) \Rightarrow (PP_i \triangleright PP_j) \text{ AND } (PP_i \triangleright PP_k))$$

$$\text{AND } ((\exists r, s, r \neq s, s \neq k, r \neq k) \Rightarrow (PP_r \triangleright PP_k) \text{ AND } (PP_s \triangleright PP_k))\}$$

$$= \{3, 5, 6, 7, 9, 13, 14\}$$

Because

$$\text{SSP} \neq \emptyset$$

so the procedure-based model of COBOL legacy system Manager Application System is synthetic (Figure 10.3, 10.4, and 10.5).
10.3.3 Slicing Programs

Slicing programs in COBOL and SQL is the first step to extract classes from source code. Program slicing techniques are based on Weiser [131], and they are sufficient (Figure 10.6).
Chapter 10. Tool and Experiments

10.3.4 UML Class Diagrams

10.3.4.1 Producing Classes

Suppose that the slicing result of one leaf program element $P$ is $C_i = \langle p, V_i \rangle$, $1 \leq i \leq n$, the slicing criterion set $PC = \{ C_i \}$, and the set of slicing criterion variables $PCV(P) = \{ V_i \}$. The algorithm of computing the relationship array is executed and acquires the value $RAC(i, j)$.

10.3.4.2 Leaf and Node Classes

There is no leaf empty program element in legacy COBOL system Manager Application System, and there are three leaf classes that represent three leaf program elements and ten node classes that represent ten node program elements (see Table 10.4 and 10.5).

Table 10.4: Table of Leaf Classes

<table>
<thead>
<tr>
<th>Procedure Number</th>
<th>Leaf Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>LEAF-Initialise-File-Copy-Tables</td>
</tr>
<tr>
<td>10</td>
<td>LEAF-All-Tasks-Complete-Process</td>
</tr>
<tr>
<td>11</td>
<td>LEAF-Get-Download-File-Size</td>
</tr>
</tbody>
</table>
### Table 10.5: Table of Node Classes

<table>
<thead>
<tr>
<th>Procedure Number</th>
<th>Node Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>NODE-Task-1-Process</td>
</tr>
<tr>
<td>4</td>
<td>NODE-Task-2-Process</td>
</tr>
<tr>
<td>5</td>
<td>NODE-Task-3-Process</td>
</tr>
<tr>
<td>6</td>
<td>NODE-Task-4-Process</td>
</tr>
<tr>
<td>7</td>
<td>NODE-Task-5-Process</td>
</tr>
<tr>
<td>8</td>
<td>NODE-Task-6-Process</td>
</tr>
<tr>
<td>9</td>
<td>NODE-Task-7-Process</td>
</tr>
<tr>
<td>12</td>
<td>NODE-Setup-Unix-Server-File-Transfer-Session</td>
</tr>
<tr>
<td>13</td>
<td>NODE-Establish-Data-Port-Connection</td>
</tr>
<tr>
<td>14</td>
<td>NODE-Establish-Unix-Data-Port-Connection</td>
</tr>
</tbody>
</table>

#### Figure 10.7: Class Diagram of Part COBOL Code

#### 10.3.4.3 Starting from Leaf Program Elements

The three leaf procedures are regarded as three leaf classes. The analysis of modelling legacy system starts from leaf program elements.

Procedure Realising_Leaf_Class_Diagram

Begin

Call Parsing_COBOL()  
Call Slicing_Program()
***to delete the redundant slicing***

For \( t=1 \) to \( (n-1) \)

For \( s=(t+1) \) to \( n \)

\[
\text{If } (PCV(S_{cs}) \subseteq PCV(S_{ct})) \text{ AND } (POV(S_{cs}) \subseteq POV(S_{ct}))
\]

\[
\text{Then } PC(P)= PC(P)-\{C_{js}\}
\]

End-If

End-For

End-For

***to setup the relationships between classes***

For \( j:=1 \) to \( (n-1) \)

For \( i:=(j+1) \) to \( n \)

\[
\text{If } RC(C_{i}, C_{j})= \emptyset
\]

\[
\text{then } RAC(i, j)=0
\]

Else

\[
RAC(i, j)=1
\]

End-If

\[
RAC(j, i)= RAC(i, j)
\]

End-For

\[
RAC(j, j)=0
\]

End-For

***to realise leaf class diagrams***

Call Drawing_Class_Diagram

End-Procedure

The leaf procedure Initialise-File-Copy-Tables only has three parameters: usr-curtze-files, usr2-curtze2-files and copied-manager-files. Based on the program slicing, there are three individual slicings and they are three individual classes. It is too simple in practice for the analysis of leaf procedure Initialise-File-Copy-Tables.

Because the whole leaf procedure has been defined as one leaf class, it is suitable for modelling the leaf procedure with composite structure diagram.

The leaf procedure All-Tasks-Complete-Process is the same as leaf procedure Initialise-File-Copy-Tables. Although leaf procedure Get-Download-File-Size has many parameters, it is analysed with composite structure diagram in replace of class diagram as well.
Chapter 10. Tool and Experiments

10.3.4.4 Modelling Node Program Elements

In node program elements, the called leaf program elements are defined as leaf classes and the called node program elements are defined as node classes. The parameters in node program elements are sliced based on the program slicing, and some of them are collected as System Manager Class. The calls that belong to system form the Systematic Call Class. Then the node class diagrams are produced.

Suppose that systematic calls SYSC\(_i\), \(1 \leq i \leq m\), make use of the array SYCA\(_i\), SYCA\(_i\)= SYSC\(_i\), \(1 \leq i \leq m\), and the variables of SYCA\(_i\) form the variable set SYCAV\(_i\), SYCAV\(_i\)=\{variable(SYCA\(_i\))\}.

Procedure Realising Node Class Diagram

Begin

***to setup variable classes and their relationships***

Call Realising Leaf Class Diagram

***to setup relationships between variable and systematic call classes***

For j:=1 to n

For i:=1 to m

If SYCAV\(_i\) \cap PCV(S\(_j\))=\emptyset

then RAC(i, j)=0

Else RAC(i, j)=1

End-If

End-For

End-For

***to realise leaf class diagrams***

Call Drawing Class Diagram

End-Procedure

The node class diagrams of legacy COBOL system are produced in Figure 10.7 to Figure 10.17 corresponding to those ten node program elements.
Chapter 10. Tool and Experiments

Figure 10.8: Class Diagram NODE-Task-1-Process

Figure 10.9: Class Diagram NODE-Task-2-Process
Figure 10.13: Class Diagram NODE-Task-6-Process

Figure 10.14: Class Diagram NODE-Task-7-Process
Chapter 10. Tool and Experiments

Figure 10.15: Class Diagram NODE- Setup-Unix-Server-File-Transfer-Session

Figure 10.16: Class Diagram NODE- Establish-Data-Port-Connection

Figure 10.17: Class Diagram NODE- Establish-Unix-Data-Port-Connection
Chapter 10. Tool and Experiments

10.3.4.5 Modelling Root Program Element

The algorithm of producing root class diagram is similar with the one of producing node class diagram.

The root program element Initialisation contains leaf classes and note classes that are modelled before. It also contains parameters that form the class Systematic manager. At last, the root program element ROOT-Initialisation is produced (Figure 10.18).

Figure 10.18: Class Diagram Initialisation
There are three leaf program elements in legacy COBOL system Manager Application System (Table 10.6). Each leaf program element has LVS(), and the relationships between the variables in LVS() are defined in Section 7.3.2. Then those leaf classes are presented with UML composite structure diagrams (Figure 10.19, 10.20 and 10.21).

Table 10.6: Table of Leaf Program Element

<table>
<thead>
<tr>
<th>Procedure Number</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Initialise-File-Copy-Tables</td>
</tr>
<tr>
<td>10</td>
<td>All-Tasks-Complete-Process</td>
</tr>
<tr>
<td>11</td>
<td>Get-Download-File-Size</td>
</tr>
</tbody>
</table>

![Figure 10.19: Composite Structure Diagram Initialise-File-Copy-Tables](image1)

![Figure 10.20: Composite Structure Diagram All-Tasks-Complete-Process](image2)

![Figure 10.21: Composite Structure Diagram Get-Download-File-Size](image3)
10.3.6 UML Sequence Diagrams

10.3.6.1 Three Kinds of Sequence Diagrams

In this legacy system Manager Application System, there are three leaf program elements—Initiate-File-Copy-Tables, All-Tasks-Complete-Process, and Get-Download-File-Size. The root sequence diagram is ROOT-initialisation. There are ten node sequence diagrams (Table 10.7).

Table 10.7: Table of Node Program Elements

<table>
<thead>
<tr>
<th>Procedure Number</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>NODE-Task-1-Process</td>
</tr>
<tr>
<td>4</td>
<td>NODE-Task-2-Process</td>
</tr>
<tr>
<td>5</td>
<td>NODE-Task-3-Process</td>
</tr>
<tr>
<td>6</td>
<td>NODE-Task-4-Process</td>
</tr>
<tr>
<td>7</td>
<td>NODE-Task-5-Process</td>
</tr>
<tr>
<td>8</td>
<td>NODE-Task-6-Process</td>
</tr>
<tr>
<td>9</td>
<td>NODE-Task-7-Process</td>
</tr>
<tr>
<td>12</td>
<td>NODE-Setup-Unix-Server-File-Transfer-Session</td>
</tr>
<tr>
<td>13</td>
<td>NODE-Establish-Data-Port-Connection</td>
</tr>
<tr>
<td>14</td>
<td>NODE-Establish-Unix-Data-Port-Connection</td>
</tr>
</tbody>
</table>

10.3.6.2 Starting from Leaf Sequence Diagrams

Based on Section 7.3, the program of realising leaf sequence diagram is presented as:

Procedure Realising_Leaf_Sequence_Diagram

Begin

Call Parsing_COBOL()

Call Slicing_Program()

***to acquire objects***

For i=1 to (k-1)

For j=(i+1) to k

If C(x_i) ⊆ C(x_j) then

VAR(P) = VAR(P) - { x_i }

End-if

End-for

End-for

End

204
Sometimes one leaf program element is simple and easy to be understood with a few variables or in simple or repeated structure. Then it is unnecessary to present sequence diagrams of those kinds of leaf program elements. The three leaf program elements in Manager Application System are Initialise-File-Copy-Tables, All-Tasks-Complete-Process, and Get-Download-File-Size, and they are simple. Based on the conditions described in Section 7.3.5, they do not need to be modelled with sequence diagram.

10.3.6.3 Node Sequence Diagrams

The node program elements contain leaf and node program elements, which are presented as leaf objects and node objects. Suppose that the leaf objects of one node program element are the array LFO(i), 1≤i≤p, and the node objects are the array NDO(j), 1≤j≤q.

Procedure Realising_Node_Sequence_Diagram

Begin

***to acquire objects***

Call Realising_Leaf_Sequence_Diagram

***to acquire messages***

For i=1 to k

MA(C(x))(i) = (i, v1i, v2i, mesi, z1i, z2i)

End-For

For i=1 to p

MA(C(x))(i+k) = (i+k, v1(i+k), v2(i+k), mes(i+k), z1(i+k), z2(i+k))

End-For

End
Chapter 10. Tool and Experiments

For \( i = 1 \) to \( q \)

\[
MA(C(x))(i+k+p) = (i+k+p, v1(i+k+p), v2(i+k+p), mes(i+k+p), z1(i+k+p), z2(i+k+p))
\]

End-For

***drawing sequence diagram***

Call Drawing_Sequence_Diagram()

End

The ten node program elements in Manager Application System (Table 10.7) are modelled with sequence diagrams. The leaf program elements called in node program elements are regarded as the special objects, named with the beginning "LEAF-".

The parameters in every node program element are sliced. Based on Section 7.3, the objects are acquired. The outside interactor of this legacy COBOL system is defined as System Manager. Then the ten node sequence diagrams are produced (Figure 10.22 to 10.31).

![Sequence Diagram NODE-Task-1-Process](image_url)

Figure 10.22: Sequence Diagram NODE-Task-1-Process
Figure 10.23: Sequence Diagram NODE-Task-2-Process

Figure 10.24: Sequence Diagram NODE-Task-3-Process
Chapter 10. Tool and Experiments

Figure 10.25: Sequence Diagram NODE-Task-4-Process

Figure 10.26: Sequence Diagram NODE-Task-5-Process
Chapter 10. Tool and Experiments

System Manager

switch to "sales" ftp-directory
receive connection handle

switch to "Sales/Updates" ftp-directory
receive connection handle

[not-switched] close connection
receive connection handle

switch to "Sales/Updates/9998" ftp-directory
receive handle

[not-connected] close connection
receive connection handle

send transmission type "T"

[not successful] close connection
receive connection handle

send "PASV" command

[not successful] close connection
receive handle
establish data port connection

set "manager.9998" file path
receive handle
get file size
transmit file
receive transmission handle

[not successful] close connection
receive handle
set flag

[not successful] close connection

Figure 10.27: Sequence Diagram NODE-Task-6-Process
Chapter 10. Tool and Experiments

Figure 10.28: Sequence Diagram NODE-Task-7-Process

Figure 10.29: Sequence Diagram NODE-Setup-Unix-Server-File-Transfer-Session

Figure 10.30: Sequence Diagram NODE-Establish-Data-Port-Connection
Chapter 10. Tool and Experiments

10.3.6.4 Root Sequence Diagram

The computation of root program element is similar with the computation of node program element.

The root program element Initialisation in legacy COBOL system Manager Application System calls two leaf program elements and seven node program elements (Table 10.1 and 10.2). Those program elements produce two leaf objects and seven node objects. The outside interactor of this legacy COBOL system is defined as System Manager. Then the root sequence diagram is produced (Figure 10.32).

Figure 10.31: Sequence Diagram NODE-Establish-Unix-Data-Port-Connection

Figure 10.32: Sequence Diagram Initialisation
10.3.7 UML Interaction Overview Diagrams

Because LEAF-Initialise-File-Copy-Tables and LEAF-All-Tasks-Complete-Process have simple structures and LEAF-Get-Download-File-Size has repeated structure, they are not modelled with interaction overview diagram based on Section 7.3.5.

10.4 Case Study of HTML Legacy System

10.4.1 HTML Legacy System

One HTML legacy system, which is named as Weather - ABC News, is regarded as the modelling example with part UML diagrams (Appendix B). It is some web pages that presents the weather forecast through the Internet of News of Australian Broadcasting Corporation.

10.4.2 Parsing HTML Legacy Code

In HTML legacy system Weather - ABC News System, the root web element is ABC News, and the node web element is Weather-ABC News. The left are the leaf web elements (Table 10.8, 10.11, and 10.12).

<table>
<thead>
<tr>
<th>Web Number</th>
<th>Name</th>
<th>Web Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ABC-News</td>
<td>root</td>
</tr>
<tr>
<td>2</td>
<td>Weather-ABC News</td>
<td>node</td>
</tr>
<tr>
<td>3</td>
<td>Weather-New South Wales-ABC News</td>
<td>leaf</td>
</tr>
<tr>
<td>4</td>
<td>Weather-Victoria-ABC News</td>
<td>leaf</td>
</tr>
<tr>
<td>5</td>
<td>Weather-Queensland-ABC News</td>
<td>leaf</td>
</tr>
</tbody>
</table>

Table 10.9: Table of Web Link Relationships and Layers

<table>
<thead>
<tr>
<th>Web Number</th>
<th>Link</th>
<th>Web Layer WL( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3, 4, 5</td>
<td>1</td>
</tr>
</tbody>
</table>
Table 10.10: Web Layers

<table>
<thead>
<tr>
<th>Procedure Layer PL()</th>
<th>Procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3, 4, 5</td>
</tr>
</tbody>
</table>

Based on the description in Section 5.4.3, the set $SAW\{}$ is the set of all web elements in HTML legacy system Weather - ABC News System,

$$SAW\{} = \{ PP_i | PP_i \in P \}$$


$$= \{1, 2, 3, 4, 5\},$$

The set $SOW\{}$ is the set of the web elements whose relationships are ordinal in HTML legacy system Weather - ABC News System,

$$SOW\{} = \{ PP_k | (\forall k, k \neq 0, PP_k \in \{\text{DOM}(PP_0)\}) \Rightarrow (PP_0 \notin \{\text{DOM}(PP_k)\}) \}$$

$$= \emptyset$$

Because

$$SOW\{} \neq SAW\{} - \{ PP_0 \},$$

so the link-based model of HTML legacy system Weather - ABC News System is not sequential.

The set $SCW\{}$ is the set of the web elements whose relationships are loop in $P$ (SCW-Set of Cyclical Webs),

$$SCW\{} = \{ PP_j | ((\forall k, k \neq j) \text{ AND } (PP_k \in SCW\{}))$$

$$\Rightarrow (PP_k \in \{\text{IM}(PP_j)\}) \text{ AND } (PP_j \in \{\text{IM}(PP_k)\}) \})$$

$$= \{2, 3, 4, 5\}$$

Because
Chapter 10. Tool and Experiments

SCW{} \neq SAW{},
so the link-based model of HTML legacy system Weather - ABC News System is not
cyclical.
The set SPW{} is the set of the web elements whose relationships contain ordinal and
loop at the same link-based model in P (SPW-Set of comPositive Webs),

\[ SPW{} = \{ PP_k | (PP_k \in \{ \text{DOM}(PP_0) \}) \Rightarrow (PP_0 \notin \{ \text{DOM}(PP_k) \}) \}
\]

AND \((\exists m, m \neq k) \Rightarrow (PP_m \in \{ \text{IM}(PP_k) \}) \text{ AND } (PP_k \in \{ \text{IM}(PP_m) \})\},\]

Because

SPW{} \neq \emptyset,
so the link-based model of HTML legacy system Weather - ABC News System is
compositive (Figure 10.33).

![Figure 10.33: Compositive Link-Based Model of Weather-ABCNews System](image)

10.4.3 UML Class Diagrams

10.4.3.1 Three Layers of HTML Class Diagrams

In HTML legacy system Weather - ABC News System, the root class diagram is the
root program element ABC News, and node class diagram is the node program element
Weather-ABC News (Table 10.11). There are three leaf program elements (Table 10.12).
Chapter 10. Tool and Experiments

There is no isolated program element in legacy system Weather - ABC News System.

Table 10.11: Table of Node Classes

<table>
<thead>
<tr>
<th>Web Number</th>
<th>Node Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>NODE-Weather-ABC News</td>
</tr>
</tbody>
</table>

Table 10.12: Table of Leaf Classes

<table>
<thead>
<tr>
<th>Web Number</th>
<th>Leaf Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>LEAF-Weather-New South Wales-ABC News</td>
</tr>
<tr>
<td>4</td>
<td>LEAF-Weather-Victoria-ABC News</td>
</tr>
<tr>
<td>5</td>
<td>LEAF-Weather-Queensland-ABC News</td>
</tr>
</tbody>
</table>

10.4.3.2 Starting from Root Web Element

Differently from modelling legacy COBOL system, the analysis starts from root web element ROOT-ABC-News.

There are four blocks in root web elements: local header, local main, local footer and additional note. Each one is regarded as one class diagram according to Section 8.3 (Figure 10.34 to 10.37). Those four blocks then form the web, which is corresponding to the root class diagram (Figure 10.38 to 10.44).

Figure 10.34: Class Diagram Local Header of ROOT-ABC-News
Figure 10.37: Class Diagram Additional Note of ROOT-ABC-News

Figure 10.38: Class Diagram ROOT-ABC-News
Chapter 10. Tool and Experiments

Figure 10.39: Class Diagram Former Note of NODE-Weather-ABC News

Figure 10.40: Class Diagram Local Header of NODE-Weather-ABC News

Figure 10.41: Class Diagram Local Main of NODE-Weather-ABC News
Chapter 10. Tool and Experiments

Figure 10.42: Class Diagram Local Footer of NODE-Weather-ABC News

Figure 10.43: Class Diagram Additional Note of NODE-Weather-ABC News
Chapter 10. Tool and Experiments

10.4.3.3 Three Leaf Web Elements

In HTML legacy system Weather - ABC News, there are three leaf web elements: Weather-NewSouthWales-ABCNews, Weather-Victoria-ABCNews, and
Chapter 10. Tool and Experiments

Weather-Queensland-ABCNews (Table 10.8). They are linked from the note web element Weather-ABCNews.

Because those three leaf web elements have similar blocks as the note web element Weather-ABCNews, their blocks are not modelled with class diagram. Those three leaf web elements are modelled with class diagram on Section 8.5 (Figure 10.45 to 10.47).

Figure 10.45: Class Diagram LEAF-Weather-New South Wales-ABC News
Figure 10.46: Class Diagram LEAF-Weather-Victoria-ABC News
Chapter 10. Tool and Experiments

10.4.4 UML Composite Structure Diagrams

Composite structure diagram is the description of internal structure of class. The classes that are composed of several equivalent parts and have the similar structure and the
same functions are not modelled in this chapter. The five typical classes, which are text, image, table, framework, and link classes, are presented with UML composite structure diagrams in Chapter 8.

The complex classes of HTML legacy system Weather – ABCNews are complicated in the structures and have many different small pieces with different functions. Those classes are LINK-TopStories, LINK-NSWNews, LINK-NSW, LINK-Sections, LINK-NewsHome, TABLE-Weather, and InternationalCentre. They are described with UML composite structure diagram in Figure 10.48 to 10.54. The classes whose structures are simple and that have a few parameters are not modelled.

<table>
<thead>
<tr>
<th>LINK-TopStories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Just In</td>
</tr>
<tr>
<td>Most Popular</td>
</tr>
<tr>
<td>Costello says Rudd 'naked' on tax system understanding</td>
</tr>
<tr>
<td>Cyberspace driving 'liquid terrorism'</td>
</tr>
<tr>
<td>Police probe Hicks on range of topics</td>
</tr>
<tr>
<td>Organised crime a looming threat to Aust</td>
</tr>
<tr>
<td>Court hears mother felt 'violated' by twin conception</td>
</tr>
<tr>
<td>Vic rejects Federal plan to hold back Murray water</td>
</tr>
<tr>
<td>Thai crash investigators say alerts not all working</td>
</tr>
<tr>
<td>Israel proclaims Gaza Strip enemy entity</td>
</tr>
<tr>
<td>Media Watch's Attard to leave program</td>
</tr>
<tr>
<td>Thurston cleared of dangerous throw charge</td>
</tr>
</tbody>
</table>

Figure 10.48: Composite Structure Diagram of LINK-TopStories
Chapter 10. Tool and Experiments

![Composite Structure Diagram of LINK-NSWNews](image1)

Figure 10.49: Composite Structure Diagram of LINK-NSWNews

![Composite Structure Diagram of LINK-NSW](image2)

Figure 10.50: Composite Structure Diagram of LINK-NSW
Chapter 10. Tool and Experiments

Figure 10.51: Composite Structure Diagram of LINK-Section

Figure 10.52: Composite Structure Diagram of LINK-NewsHome
Chapter 10. Tool and Experiments

TABLE-Weather

| Capital City Forecast | Yesterday's Temperature | International Centres |

Figure 10.53: Composite Structure Diagram of TABLE-Weather

International Centres

| Centre | Forecast | Min | Max |

Figure 10.54: Composite Structure Diagram of InternationalCentres

10.4.5 UML Component Diagram

![UML Component Diagram](image)

Figure 10.55: Component Diagram of Weather-ABCNews
10.4.6 UML Deployment Diagram

Figure 10.56: Deployment Diagram of Weather-ABCNews

10.5 Case Study of SQL Legacy System

10.5.1 SQL Legacy System

One SQL legacy system, which is named as SimulatingINSERT, is regarded as the modelling example with part UML diagrams (Appendix C). It is one SQL example to create one statement that has the same function as the INSERT statement in SQL. The statement in this legacy system is presented as one procedure named as sp_generate_inserts. It is written by Narayana Vyas Kondreddi at http://vyaskn.tripod.com and tested on SQL Server 2000.
10.5.2 Parsing SQL Legacy Code

In SQL legacy system SimulatingINSERT code, the database used is Master. The root program element is Master, and the leaf program element is sp_generate_inserts (Table 10.13). Suppose that the procedure sp_generate_inserts uses the database sp_generate_inserts. It is obvious that the development/environment-specific model of SQL legacy system SimulatingINSERT code is generation database-based model (Figure 10.57).

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Master</td>
<td>root</td>
</tr>
<tr>
<td>2</td>
<td>sp_generate_inserts</td>
<td>leaf</td>
</tr>
</tbody>
</table>

Figure 10.57: Generation Database-Based Model of SimulatingINSERT

10.5.3 Slicing SQL Source Code

Based on the program slicing techniques, SQL source code is sliced.

10.5.4 UML Class Diagrams

10.5.4.1 Two Layers of Procedure Elements

In legacy SQL system Simulating INSERT system, the root program element is named as Master, and the leaf program element is the procedure sp_generate_inserts. There is no node and isolated program element in legacy system Simulating INSERT system.

10.5.4.2 Class Diagram

Legacy SQL system Simulating INSERT system has three classes, one root class ROOT-Master, and two leaf class LEAF-master.dbo.sp_MS_upd_sysobj_category and
LEAF-sp_generate_inserts based on Section 9.3. Then its class diagram is realised in Figure 10.58.

Figure 10.58: Class Diagram of Simulating INSERT System

10.5.5 UML Composite Structure Diagrams

In modelling SQL legacy system Simulating INSERT system, UML composite structure diagram is used to describe the internal structure of the leaf program element LEAF-sp_generate_inserts (Figure 10.59).

Figure 10.59: Composite Structure diagram of LEAF-sp_generate_inserts
Chapter 10. Tool and Experiments

10.5.6 UML Activity Diagrams

10.5.6.1 Three Layers of Activity Diagrams

In legacy SQL system Simulating INSERT system, the root program element Master result in the root activity diagram ROOT-Master, and leaf program element sp_generate_inserts results in the leaf activity diagram LEAF-sp-generate-inserts. There is no node and isolated program element in legacy system Simulating INSERT system so there is no node or isolated activity diagram.

10.5.6.2 Starting from Leaf Program Elements

The analysis of modelling legacy SQL system starts from leaf program element. Based on Section 9.3.3, control graph is obtained from SQL code Master, and the activity array AT(1, 1) is acquired.

\[
\begin{align*}
AT(1, 1, 1) &= \text{"check original data";} \\
AT(1, 1, 2) &= \text{"set target table";} \\
AT(1, 1, 3) &= \text{"get the first column’s ID";} \\
AT(1, 1, 4) &= \text{"[column_ID is not null] get column’s name and type";} \\
AT(1, 1, 5) &= \text{"delete extra characters";} \\
AT(1, 1, 6) &= \text{"form executing string".}
\end{align*}
\]

Then the activity diagram of LEAF-sp-generate-inserts is produced in Figure 10.60.

10.5.6.3 Leaf and Node Activities

The leaf activity in this legacy SQL system is LEAF-sp_generate_inserts activity. There is no node activity.

10.5.6.4 Root Activity Diagram

The activity array AT(0, 1) of ROOT-Master is acquired.

\[
\begin{align*}
AT(0, 1, 1) &= \text{"use master database";} \\
AT(0, 1, 2) &= \text{"[sp_generate_inserts exists] drop sp_generate_inserts";} \\
AT(0, 1, 3) &= \text{"turn system-category on";} \\
AT(0, 1, 4) &= \text{"create procedure sp_generate_inserts";} \\
AT(0, 1, 5) &= \text{"turn system-category off";} \\
AT(0, 1, 6) &= \text{"grant to sp_generate_inserts public".}
\end{align*}
\]
Chapter 10. Tool and Experiments

Then the activity diagram of ROOT-master is produced in Figure 10.61.

![Activity diagram of LEAF-sp_generate_inserts]

Figure 10.60: Activity diagram of LEAF-sp_generate_inserts
10.6 Summary

In this chapter, three cases are studied. One COBOL legacy system named Manager Application is presented to create the new manager file that is performed once a week on Friday. It is modelled with UML class diagram, composite structure diagram,
interaction overview diagram, and sequence diagram. One HTML legacy system named as Weather-ABCNews is regarded as the example to present the weather forecast through the Internet of News of Australian Broadcasting Corporation. It is modelled with UML class diagram, composite structure diagram, component diagram and deployment diagram. One SQL legacy system named as Simulating INSERT is regarded as the example to create one statement that has the same function as the INSERT statement in SQL. It is modelled with UML class diagram, composite structure diagram, and activity diagram. One tool demonstrates the proposed approach.
Chapter 11
Discussion and Conclusions

11.1 Comparison and Evaluation

Traditional studies obtained UML diagrams from legacy code through program slicing techniques, which have shortcomings. Formal methods lack a common platform and this leads to confusion and difficulties with the comprehension and reuse of original code. The problem with model-driven engineering is that it can lead to a lock-in in the abstractions and generator technology adopted when the project starts. Other methods also lack a common platform, which seriously impedes the reuse of legacy systems. Using all the UML diagrams results in redundancy.

The proposed approach overcomes these disadvantages of traditional studies by using development/environment-specific models. Modelling enables IT to be more efficient in reacting to business users' requests for new systems or changes to existing ones, and it is able to build applications once and use them many times. It satisfies the users by giving them immediate answers to questions as they arise. It enables IT to be nimbler and quicker in reacting to business changes, rather than having IT be a constraint on the business. Modelling has the ability to help break down the applications supporting various products into pieces that can be saved or discarded as duplicative.

The proposed approach is based on development/environment-specific models. They are different for different legacy systems. The proposed development/environment-specific model of COBOL legacy systems is based on the characteristics and operations of COBOL, and is the linear, branch, joint, and synthetic procedure-based model; the proposed development/environment-specific model of HTML legacy systems is the sequential, cyclical, and compositive link-based model; and the proposed development/environment-specific model of SQL legacy systems is based on the characteristics and operations of SQL, and is the association, generation and composition database-based model.
Chapter 11. Discussion and Conclusions

The traditional studies sliced legacy code with program slicing techniques based on Weiser's approach. However, the slices are computed by computing consecutive sets of transitivity relevant statements based on data flow and control flow dependencies. COBOL and SQL legacy systems involve data flow and control flow, and program slicing techniques are suitable for modelling them, whilst HTML legacy systems are the presentation of web and data formats, and do not focus on data flow or control flow dependencies, and program slicing techniques are not suitable for analysing HTML legacy systems.

The proposed approach presents the differences in applying program slicing techniques between COBOL and SQL legacy systems on the one hand, and HTML legacy systems on the other. The first two types can be sliced, but not the last one.

Traditional studies presented all the UML diagrams. As a theory, UML should be all that is needed to model real systems. It is necessary for UML to be sufficient because it is a universal modelling theory and it should have the capability of being suitable for different systems. But some of the UML diagrams are similar or even redundant in some ways and in some areas, especially in software systems. When using UML in the designated area, it should be edited so as to be suitable for the given modelling systems. It is not necessary to use all of its thirteen diagrams to model those legacy systems.

The proposed approach uses four UML diagrams, which are the class, composite structure, sequence and interaction overview diagrams, to model COBOL legacy systems; uses four UML diagrams, which are the class, composite structure, component and deployment diagrams, to model HTML legacy systems; and three UML diagrams, which are the class, composite structure and activity diagrams, to model SQL legacy systems.

11.2 Summary

The proposed approach in this thesis is an attempt to reduce some of the problems inherent with domain-specific legacy systems. It involves many issues including models, program slicing techniques and UML. Although formal methods, cognitive methods and model-driven engineering of software evolution have been around for some years, dedicated approaches using the integration of development/environment-specific
Chapter 11. Discussion and Conclusions

models of domain-specific legacy systems and some UML diagrams did not exist.

In order to deal with those existing shortcomings, development/environment-specific models are generated. The development/environment-specific model of COBOL legacy systems is based on the characteristics and operations of COBOL, and is a procedure-based model. It is a graph that describes the calling and being-called relationships of those procedures in COBOL legacy systems, and has four types. The development/environment-specific model of HTML legacy systems is a link-based model. It is a graph that describes the importing or being-imported relationships of those webs in HTML legacy systems. It has three types. The development/environment-specific model of SQL legacy systems is a database-based model. It is a graph that describes the database relationships in COBOL legacy systems, and has three types.

29 rules are used to model COBOL legacy systems with four UML diagrams. In Section 7.3.5, the rules application is presented. Eleven rules then model HTML legacy systems with four UML diagrams. Finally, another eleven rules model SQL legacy systems with three UML diagrams.

11.3 Significance of Contributions and Evaluation

The major contribution of the thesis is the presentation of development/environment-specific models of legacy systems and an approach towards the software evolution of domain-specific legacy systems with some UML diagrams. In concrete terms, the original contributions of this thesis are described as follows:

OC1: In Chapter 4, 5, and 6, development/environment-specific models of three domain-specific legacy systems, which are a procedure-based model of COBOL legacy systems, a link-based model of HTML legacy systems, and a database-based model of SQL legacy systems, are defined based on the characteristics and operations of these domain-specific legacy systems. They integrate formal methods and cognitive methods of software evolution and contain structural and operational information about working flow or executable functions.
OC2: In Chapter 7, four out of thirteen UML diagrams are used to model COBOL legacy systems; in Chapter 8, four UML diagrams are used to model HTML legacy systems; and in Chapter 9, three UML diagrams are used to model SQL legacy systems. It is made clear that only some of the set of UML diagrams are used to model domain-specific legacy systems, so as to eliminate the redundancy of UML application.

OC3: In Chapter 10, a system is developed to demonstrate the effectiveness of the proposed approach by applying and integrating evolution rules in model applications based on parsing and slicing domain-specific legacy systems.

OC4: In Chapter 7, 8, and 9, a set of rules is devoted to modelling domain-specific legacy systems with their application conditions.

11.4 Revisiting Criteria for Success

The approach presented in the thesis is based on the criteria for success in Chapter 1. The analysis of those criteria for success is performed as follows in order to judge the success of the research.

- For those domain-specific legacy systems, is it possible and necessary for them to be modelled with UML?

  UML is a graphical language for visualising, specifying, constructing, and documenting the artifacts of a software-intensive system. It is appropriate for modelling, ranging from enterprise information systems to distributed Web-based applications and even to hard real-time embedded systems. It is a very expressive language with the graphic presentation of diagrams addressing the views that need to be developed and the models extracted from systems. Because a legacy system is a computer software system, UML is able to model it. A legacy system comprises large, complicated, old, heavily modified, difficult to maintain and old-fashioned software that is still important to the organisation. In order to reuse and restructure the legacy system, it needs to be modelled using UML diagrams.

- Is it necessary for those three kinds of legacy system to be modelled with all of the UML diagrams?
Since UML was released, it has become regarded as the new modelling standard, because most of the currently existing methods have been integrated within UML. As a theory, UML is all that is needed to model real systems. It is the integration of many different methods in different views from different layers. It is necessary for UML to be sufficient because it is a universal modelling theory and it should have the capability of being suitable for different systems. It therefore contains a certain number of concepts, which are similar or even redundant in some ways and in some areas, especially in software systems. A large number of definitions are presented that are not needed in the specific area but only for theoretical reasons. Therefore, when using UML in the designated area, it should be edited so as to be suitable for the modelling systems in question. It is not necessary to use all of its thirteen diagrams to model those legacy systems.

In practice, for given modelling tasks in given modelling areas, UML is well placed to cope with the problems at hand. Comprehension of systems is complicated and it is influenced by different aspects of the systems that could relate to structure, behaviour, execution process, error detection, etc. Each of those requires a different understanding from a different point of view. Therefore, when modelling real legacy systems, the research in this thesis has identified the most appropriate modelling approach, using UML, for the given tasks. Only some of those thirteen UML diagrams were selected and utilised in order to model domain-specific legacy systems.

- If not necessary, how many UML diagrams are suitable for modelling COBOL legacy systems? What are they? Are they enough for the modelling task? Why?

The class and composite structure diagrams were used to model the static parts of COBOL legacy systems, and the sequence and interaction overview diagrams to model the dynamic parts.

The class diagram is the most fundamental of the UML diagrams for modelling the structure of legacy systems. Because of the popular acceptance of the definition CLASS, the class diagram is easy to understand in modelling a legacy system. So, the class diagram should be used to model legacy systems.

Although the composite structure diagram is low-level, its connotation is different
from that of the object diagram. It describes the internal structure of a class and for different purposes from those of the object diagram. It can be used to model the complication of a class in detail. The composite structure diagram should therefore be used when modelling the internal structure of classes of COBOL legacy systems.

The sequence diagram is a high-level UML diagram that models the dynamic aspects of modelled systems. Sequence diagrams present the interactions between objects when achieving a result. A sequence diagram describes how groups of objects collaborate in presenting certain system behaviours. Typically, a sequence diagram describes the detailed implementation of how a legacy system accomplishes its main tasks. A sequence diagram presents an interaction in terms of a set of messages sent between objects that all work together to provide a desired operational result, which is its main difference from a collaboration diagram which shows collaborations and associations between instances in a system. The sequence diagram is useful in modelling COBOL legacy systems. It is used in presenting the dynamic aspects of COBOL legacy systems.

The interaction overview diagram is a diagram which is new in UML 2.0. It overviews the control flow within a system or business process at a low level. It focuses on the overview of the flow of control of the interactions. It describes the interactions where messages and lifelines are hidden. Because the interaction overview diagram is low-level in describing the control flow within a legacy system, it is appropriate for modelling COBOL legacy systems, and therefore it is used in this thesis.

An object has the same characteristics as the corresponding class. Most of the important characteristics of the object are reflected in the corresponding class. Therefore, after the class diagram has been used in modelling a COBOL legacy system, it is unnecessary to use the object diagram in modelling.

The UML collaboration diagram is used to model interactions between objects, and objects interact by invoking messages on each other. The collaboration diagram is another form of sequence diagram, and the sequence diagram is used in modelling COBOL legacy systems. Because the composite structure diagram has been used
Chapter 11. Discussion and Conclusions

to model the internal structure of the classes in a COBOL legacy system, it is not necessary to use the collaboration diagram.

The component diagram could be used at a high level to analyse legacy COBOL systems. After the class diagram has been used in modelling a COBOL legacy system at a high level, the component diagram is pleonastic and redundant for modelling the functionality of COBOL code. The component diagram was therefore not used in this research.

Because a COBOL legacy system is a description of business rules and is complicated in practice, the package diagram is at too high a level to be useful in modelling COBOL legacy systems. After using the class diagram it is therefore not necessary to use the package diagram to model COBOL code.

A COBOL legacy system is modelled from the functional point of view. The deployment diagram is too high-level for analysing COBOL legacy systems from the static and structural points of view. Therefore, the deployment diagram is not used in modelling COBOL legacy systems.

The UML activity diagram describes the dynamic aspects of a system at a high level. It is essentially a flowchart, showing the flow of control from activity to activity. An activity is an ongoing nonatomic execution within a state machine. Activity diagrams may stand alone to visualise, specify, construct and document the dynamics of a society of objects, or they may be used to model the flow of control of an operation. Because the sequence diagram is used at a high level to describe the messages and objects of COBOL legacy systems, and presents the operations with messages and the message senders or receivers, the activity diagram does not need to be used when modelling COBOL legacy system.

The use case diagram is defined as a high-level diagram that shows the relationships amongst actors and use cases within a system in terms of the UML specification. It shows a set of use cases and actors and their relationships. Use case diagrams can be used to model the context and requirements of the system from the stakeholders' point of view. A use case is a requirement that the users of the system, termed the actors, want the system to do. A use case contains a special function that can be specified as a set of usage scenarios. As a user-centred
analysis technique, the purpose of a use case is to yield a result of measurable value to an actor in response to the initial request of that actor. Use cases can be utilised to model the requirements of the system from the stakeholders’ point of view. Because the modelling of COBOL legacy systems focuses on the modelling of the business rules in COBOL code, and does not need to determine the requirements of the users, the use case diagram is not used in modelling COBOL legacy systems.

Timing diagrams are used to show changes and their relationship to clock times. It provides a visual representation of objects changing state and interacting over time. Because the timing diagram is suitable for the description of embedded systems, it is not used in modelling COBOL legacy systems.

A state machine diagram describes the possible states of a single class and the events that cause state transitions. It shows the sequence of states that an object goes through during its life cycle in response to stimuli. It is useful for showing the life cycle of the class. Generally, it is attached to a class of objects with an interesting dynamic behaviour. When a transition in a statechart is triggered, the object leaves its current state, initiates the action(s) for that transition and enters a new state. Any internal or external event is broadcast to all states of all objects in the system. Because the objects in a COBOL legacy system are extracted from variables and they often have different values, the states of objects are changed so often that it is hard to describe them, and to do so would be meaningless. Therefore, the state machine diagram is not used in modelling COBOL legacy systems.

Consequently, four UML diagrams, which are the class, composite structure, sequence and interaction overview diagrams, are used to model COBOL legacy systems.

- How many UML diagrams are suitable for modelling HTML legacy systems? What are they? Why?

Four UML diagrams, which are the class, composite structure, component and deployment diagrams, are used to model HTML legacy systems.

The essential task of HTML is the service of the Internet, and this is its biggest restriction and difference from other programming languages. HTML does not take
on the responsibilities of calculation, real-time control, judgments and immediate response, batch disposal, database data management, man-made intelligence, etc. HTML is an excellent tool for presenting data under a designed format, especially with tables and frames. The nature of HTML means that it does not focus on complicated control or computation. Therefore, legacy HTML systems do not need to be modelled with the dynamic UML diagrams.

The class diagram is the most fundamental of the UML diagrams used in modelling the structure of legacy systems. As mentioned before, the class diagram should be used when modelling HTML legacy systems, but it is not necessary to use the object diagram.

The composite structure diagram is low-level, and it describes the internal structure of classes, having a different purpose from that of the object diagram. Tables, forms, images and frames have different and complex structures in web usage, and their internal structures should be presented in detail in order for them to be more easily understood. So, the composite structure diagram should be used in modelling the internal structure of classes in HTML legacy systems.

The component diagram shows the dependencies amongst software components, including the classifiers that specify them and the artifacts that implement them; such as source code files, binary code files, executable files, scripts and tables. The desired web page is realised through the correct use of tables, forms, images and sounds in web pages of the designed format. So the component diagram is used in modelling HTML legacy systems.

The UML deployment diagram presents the processors, devices and the connections between them, that can be the run-time configuration of hardware nodes and the software components that run on those nodes. HTML uses web servers and browsers to execute web presentation and information exchange. Therefore, the deployment diagram is used in modelling HTML legacy systems.

The collaboration diagram is another form of sequence diagram. HTML is an excellent tool for presenting data under the designed format, especially with tables and frames. It does not focus on complicated control or computation. So it is not necessary to use the collaboration diagram when modelling HTML legacy systems.
Because the most common use of package diagrams is to organise use case diagrams and class diagrams, although not limited to these, and HTML is an excellent tool for presenting data under the designed format, it is not necessary to use the package diagram when modelling HTML legacy systems.

Consequently, four UML diagrams, which are the class, composite structure, component and deployment diagrams, are used to model HTML legacy systems.

○ How many UML diagrams are suitable for modelling SQL legacy system? What are they? Why?

Three UML diagrams, which are the class, composite structure, and activity diagrams, are used to model SQL legacy systems.

Because the class diagram is the most fundamental of the UML diagrams in modelling the structure of legacy systems, it is used to model legacy SQL systems, but the object diagram is not used.

The composite structure diagram can be used to model the complications of a class in detail. SQL is a full-featured relational database management system that offers a variety of administrative tools to ease the burdens of database development, maintenance and administration. The language offers a flexible interface for databases of all shapes and sizes. Because every database has its own specific structure, the composite structure diagram should be used in modelling the internal structure of classes of SQL legacy systems.

The UML activity diagram describes a flowchart showing the flow of control from activity to activity at a high level. SQL systems involve the flow of control within databases. So the activity diagram is used in modelling SQL legacy systems.

The UML collaboration diagram is used to model interactions between objects, and objects interact by invoking messages on each other. The class is the abstraction of the common characteristics of the object group. Most of the important characteristics of the object are reflected in the corresponding class. Because the composite structure diagram has been used to model the internal structure of the classes in an SQL legacy system, it is not necessary to use the collaboration diagram as well.
Chapter 11. Discussion and Conclusions

The component diagram is at a high level. After the class diagram has been used in modelling a COBOL legacy system at a high level, the component diagram is pleonastic and redundant in modelling SQL code. So the component diagram is not used.

The package diagram is at too high a level for modelling SQL legacy systems. As the class diagram has been used, it is not necessary to use the package diagram as well.

The UML deployment diagram has to do with the run-time configuration of hardware nodes and the software components that run on those nodes. It is too high-level for analysing SQL legacy systems from the static and structural points of view. Therefore, the deployment diagram is not used for modelling SQL legacy systems.

Sequence diagrams present, at a high level, interactions between objects when achieving a result. A sequence diagram describes how groups of objects collaborate in presenting certain system behaviours. Because the main focus and objects of SQL systems are databases, which are created and deleted, there are few messages passing between those databases, as there would be with the COBOL, BASIC and FORTRAN languages. The sequence diagram, describing objects and messages between them, is therefore not used.

The use case diagram shows, at a high level, the relationships amongst actors and use cases within a system, in terms of the UML specification. It is a set of use cases and actors and their relationships. A use case is a requirement that the users of the system want the system to do. Because the modelling of an SQL legacy system focuses on the modelling of the databases, and it does not need to determine the requirements of the users, the use case diagram is not used in modelling SQL legacy systems.

The interaction overview diagram shows the control flow within a system at a low level. SQL systems are concerned with control flow within the databases. However, SQL systems mainly focus on databases, their main objects are databases, and they mainly create or delete databases, which is unlike the handling of objects in COBOL-like or BASIC-like languages. The control flow in SQL systems is
modelling the activity diagram, and it is not necessary to use the interaction overview diagram to model SQL legacy systems.

A state machine diagram shows the sequence of states that an object goes through during its life cycle. For the same reason as above, it is not used in modelling SQL legacy systems.

The timing diagram explores the behaviours of one or more objects throughout a given period of time. It depicts the change of the state or condition of a classifier instance or role over time. Because the timing diagram is suited for the description of embedded systems, it is not used in modelling SQL legacy systems.

Consequently, only three UML diagrams, which are the class, composite structure and activity diagrams, are used to model SQL legacy systems.

- Is it necessary for all legacy systems to be sliced? Why?

According to Weiser's introduction, slices are computed by computing consecutive sets of transitively relevant statements based on their data flow and control flow dependencies. Because COBOL and SQL legacy systems involve data flow and control flow dependencies, they are suitable for the use of program slicing techniques in modelling them.

However, the main characteristics of HTML legacy systems are the presentation of web and data formats. They offer common platforms for different data. They have few variables. Consequently, program slicing techniques are not suitable for analysing HTML legacy systems.

- Is it helpful to use models in software evolution?

Models enable IT to be more efficient in reacting to business users' requests for new systems or changes to existing ones, and make it more feasible to build an application once and use it many times, thus enabling faster reactions to business changes. They reduce the need to access the basic components of applications and promote the sharing, by building frameworks, of various pieces of functionality across applications. They can also help to break down the applications supporting various products into pieces that can be saved or discarded as duplicates, in order to integrate the users' products with existing products. The use of models
Chapter 11. Discussion and Conclusions

overcomes the disadvantages inherent in traditional studies of software evolution.

• Is it right to build development/environment-specific models of COBOL, HTML, and SQL legacy systems?

The development/environment-specific models of COBOL legacy systems correspond to the procedure relationships in COBOL legacy systems. The procedure relationship describes the calling or being-called relationship between the two procedures in a COBOL legacy system, and has four kinds: one to one, one to many, many to one, and many to many. Its model is based on the characteristics and operations of COBOL, and is a procedure-based model which is a graph that describes the calling and being-called relationships of those procedures in COBOL legacy systems. It has four kinds: linear, branch, joint, and synthetic procedure-based models.

The development/environment-specific models of HTML legacy systems correspond to the web relationships in HTML legacy systems. A web relationship depicts the linking or being-linked relationship between two web pages in an HTML legacy system. It has two kinds: ordinal and loop. Its model is a graph that describes the importing or imported relationships of those webs in an HTML legacy system. It has three kinds: sequential, cyclical, and compositive link-based models.

The development/environment-specific models of SQL legacy systems correspond to the database relationships in SQL legacy systems. The relationships between databases in SQL legacy systems are mainly of two types: generation and association. A models is based on the characteristics and operations of SQL, and is a database-based model which is a graph that describes the database relationships in a COBOL legacy system. Database-based models are of three types: association, generation and composition.

• Is the realisation of those UML diagrams obtained from COBOL legacy systems appropriate?

The experiment on an COBOL legacy code was performed in Section 10.3 and was successfully based on Chapter 4 and 7. Therefore, the realisation of those UML diagrams from COBOL legacy systems is appropriate. The definitions and
algorithms are correct. Those algorithms result in UML diagrams from legacy COBOL code.

- Is the realisation of those UML diagrams from HTML legacy systems appropriate?

After the experiment on an HTML legacy code was successfully performed in Section 10.4, based on Chapter 5 and 8, the answer to the question is positive. The definitions and algorithms are correct and the realisation of those UML diagrams is appropriate.

- Is the realisation of those UML diagrams from SQL legacy system appropriate?

As mentioned in Section 10.5, the experiment on an SQL legacy code was successfully based on Chapter 6 and 9. Therefore the answer to the question is positive.

11.5 Conclusion

A model is a description designed to show the structure or workings of a system. A model enables IT to be more efficient in reacting to business users' requests for new systems or changes to existing ones. A model makes it feasible to build an application once and use it many times.

A model creates an IT infrastructure that can quickly react to business changes, rather than IT being a constraint on the business. A model gives the users immediate answers to questions as they arise. A model makes IT more flexible and quicker to react to the changes affecting a business.

A model reduces applications to their basic components, so that parts of them can be reused or dropped by various parts of the company as needed. A model shares various pieces of functionality across applications and builds a framework. It is attractive to both IT professionals and business managers. A model is handy when engineers are trying to slice and dice functionality into atomic pieces of business functions.

A model has the ability to help break down the applications supporting various products into pieces that can be saved or discarded as duplicates in order to integrate the users' new products with their existing products, so that there will be no overlap.
Chapter 11. Discussion and Conclusions

The proposed approach is based on development/environment-specific models. They are different in different legacy systems. The proposed development/environment-specific model of COBOL legacy systems is based on the characteristics and operations of COBOL, and is a procedure-based model; the proposed development/environment-specific model of HTML legacy systems is a link-based model; and the proposed development/environment-specific model of SQL legacy systems is based on the characteristics and operations of SQL, and is a database-based model.

The proposed approach addresses the shortcomings of traditional studies. It presents the suitability of program slicing techniques to different legacy systems, and uses a selection of UML diagrams rather than all of them.

11.6 Future Work

11.6.1 Limitations

The proposed approach in this thesis has some limitations in the research area. In order to address those limitations, the directions of possible future work are outlined below.

The limitations of the approach proposed in this thesis include the following aspects:

(1) Domain knowledge is an important issue in software engineering, but the approach proposed in this thesis does not focus on domain knowledge.

(2) Besides the program slicing techniques outlined by Weiser, other slicing techniques are not considered or used in the proposed approach.

(3) An area for future research involves the extension of the approach described in this thesis into other fields, that is, there is a need to find out whether or not the approach presented in this thesis can be used to meet the modelling of legacy systems from other fields besides the software evolution of domain-specific legacy systems. In this context, real-time legacy systems in particular should be considered.

(4) The purpose of software evolution is to reuse legacy systems. The proposed approach does not demonstrate reuse.
11.6.2 Directions for Future Work

In order to address the limitations mentioned above, further research should be focused on the following aspects:

(1) Although software evolution of domain-specific legacy systems is related to domain knowledge, the proposed approach does not include all the domain knowledge of domain-specific legacy systems. Further research should discuss software evolution of domain knowledge of domain-specific legacy systems.

(2) Other slicing techniques, including FERMAT used in Wide Spectrum Language (WSL), are useful. Further research should present other slicing techniques, compare the slicing results, and judge their suitability for different legacy systems.

(3) Real-time systems are diversified and complicated, and different branches have distinct characteristics. Further research should therefore include real-time legacy systems.

(4) The next area for further research will be the forward engineering of domain-specific legacy systems from UML into software, in order to improve the development of the software application.
References


References


References


References


References


References


[86] Y. Li and H. Yang, "Code Understanding through Context Oriented Uncertainty


References


References


References


References


Appendix A: COBOL Legacy System

One COBOL legacy system named Manager Application is presented and was modelled with part UML diagrams. It is designed to create the new manager file for the application that is performed once a week on Friday. It is executed on a Window client, and communicates with a local Unix server and a remote Window server. This legacy software is licensed by SALES PRO TECHNOLOGIES, INC.
move 1 to task-1-act-visible
move 1 to task-1-incmp-visible
move 1 to task-1-mcmp-visible
move 1 to ok-status
display message box
"Unable to connect to the local server. The connection was refused. The port or IP address may be incorrect. Contact your support administrator.

title "District Manager File Process"
type mb-ok
icon mb-warning-icon
go to all-tasks-complete-process
end-if

initialise unix-receive-command-byte-hold
move 1 to unix-receive-end-length
perform until byte-hold > end-cmd-byte or unix-receive-end-length > max-end-length
move 1 to num-bytes-read
"CWD " delimited by size
string byte-hold delimited by size into unix-send-command with pointer

string byte-hold delimited by size into unix-receive-end-length
end-perform

move unix-receive-command to parse-receive-command.

if parse-command-code = "511"

initialise unix-send-command
move 1 to unix-send-end-length
string "PASS " delimited by size
unix-password delimited by size
x"OA" delimited by size
into unix-send-command with pointer

unix-send-end-length > max-cmd-length
move 1 to ok-status
display message box
"Unable to connect to the local server. The connection was refused. An incorrect user name or password was detected. Contact your support administrator.

title "District Manager File Process"
type mb-ok
icon mb-warning-icon
go to all-tasks-complete-process
end-if

initialise unix-receive-command-byte-hold
move 1 to unix-receive-end-length
perform until byte-hold > end-cmd-byte or unix-receive-end-length > max-end-length
move 1 to num-bytes-read
"CWD " delimited by size
string byte-hold delimited by size into unix-receive-command with pointer

string byte-hold delimited by size into unix-receive-end-length
end-perform

move unix-receive-command to parse-receive-command.

if parse-command-code = "210"

move spaces to accyunk
display message box
"Unable to connect to the local server. The connection was refused. An incorrect user name or password was detected. Contact your support administrator.

title "District Manager File Process"
type mb-ok
icon mb-warning-icon
go to all-tasks-complete-process
end-if

if parse-command-code = "250"

move "curdir" to unix-directory
initialise unix-receive-command
move 1 to unix-send-end-length
string "CWD " delimited by size
unix-send-end-length > max-cmd-length
move 1 to ok-status
display message box
"Unable to connect to the local server. The connection was refused. An incorrect user name or password was detected. Contact your support administrator.

title "District Manager File Process"
type mb-ok
icon mb-warning-icon
go to all-tasks-complete-process
end-if

if parse-command-code = "230"

setup-umx-server-file-transfer-session

Appendix A: COBOL Legacy System

call "clisocket" using ags-write, mix-connection-handle, type mb-ok
move I to unx-receive-command-byte-read
subtract I from unx-receive-command-byte-read
move I to num-bytes-read
end-perform

if pane-command-code = '125' or unx-receive-command-byte-read > max-end-length
move I to task-2-act-visible
move 0 to file-byte-read
move 0 to temp-byte-read
move 0 to prog-bar-col
divide file-total-byte-size by 41 giving prog-bar-read
remorder prog-bar-read
perform until file-byte-read > file-total-byte-size
initialize unu-receive-command-byte-read
move I to unx-receive-command-byte-read
subtract I from unx-receive-command-byte-read
if bytes-remaining = 1
close receiving-file
display frame, line 29.20, col bar-col, file-byte-read, background-high, lowered
end-perform
else
string "Unable to copy* delimited by size
move I to task-2-act-visible

end-if

else
string byte-hold delimited by size
move I to task-2-act-visible

end-perform
Appendix A: COBOL Legacy System

display message box
"Unable to copy manager files. The directory '/usr curt' was unable to be reached. Contact your support administrator." title "District Manager File Process" type mb-ok icon mb-warning-icon go to all-tasks-complete-process end-if.

if parse-command-code = '250' move '250' to unix-directory initialize unix-command-length move 1 to unix-send-end-length string 'CWD' delimited by size
unix-directory delimited by spaces
'x'0A' delimited by size
into unix-command-end with pointer
unix-send-end-length subtract 1 from unix-send-end-length call 'checkext' using ags-write, unix-connection-handle, unix-command, unix-send-end-length call 'checkext' using ags-flush, unix-connection-handle
initialise unix-receive-command-byte-held move 1 to unix-receive-end-length perform until byte-held = end-byte or
unix-receive-end-length > max-end-length
move 1 to num-byte-read call 'checkext' using ags-read, unix-connection-handle, byte-held, num-byte-read string byte-held delimited by size into
unix-receive-command with pointer
unix-receive-end-length end-perform move unix-receive-command to parse-receive-command
else move 0 to task-2-act-visible move 0 to task-2-ico-visible move 0 to task-2-in-complete-visible move 1 to task-2-ico-incomplete-visible move 1 to task-2-visible end-if.

if parse-command-code = '250' move '250' to unix-directory initialize unix-command-length move 1 to unix-send-end-length string 'CWD' delimited by size
unix-directory delimited by spaces
'x'0A' delimited by size
into unix-command-end with pointer
unix-send-end-length subtract 1 from unix-send-end-length call 'checkext' using ags-write, unix-connection-handle, unix-command, unix-send-end-length call 'checkext' using ags-flush, unix-connection-handle
initialise unix-receive-command-byte-held move 1 to unix-receive-end-length perform until byte-held = end-byte or
unix-receive-end-length > max-end-length
move 1 to num-byte-read call 'checkext' using ags-read, unix-connection-handle, byte-held, num-byte-read string byte-held delimited by size into
unix-receive-command with pointer
unix-receive-end-length end-perform move unix-receive-command to parse-receive-command
else move 0 to task-2-act-visible move 0 to task-2-ico-visible move 0 to task-2-in-complete-visible move 1 to task-2-ico-incomplete-visible move 1 to task-2-visible end-if.

if parse-command-code = '250' perform setup-user-server-file-transfer-exception
else move 0 to task-2-act-visible move 0 to task-2-ico-visible move 0 to task-2-in-complete-visible move 1 to task-2-visible

368
call "cSsocket" using ags-create-client, @p-server-port, move "password17" to ftp-password, if ftp-connection-handle = 0 move ftp-server-address to @p-server-address giving ftp-connection-handle move '11.1.11111' to ftp-server-address, end-if
move '21' to @p-server-port
move I to task-5-ico-ist-visible.
move I to task-5-act-visible.
move 1 to task-4-sco-vtsible.
move Ito task-4-visible.
move 0 to task-4-ico-act-visible.
end-perform.
call "cSsocket" using ags-read, ftp-connection-handle, display message box
"A problem was encountered while attempting to create the FTP server. The connection was refused by the server. The port or IP address may be incorrect. Contact your system administrator.",
title "District Manager File Process" type mb-ok
icon mb-warning-icon
go to all-tasks-complete-process
end-if.
call "cSsocket" using ags-flush, ftp-connection-handle.
call "cSsocket" using ags-write, ftp-connection-handle, byte-hold, num-bytes-read
string byte-hold delimited by size into umx-send-cmd-length.
initialise umx-send-command byte-hold.
call "cSsocket" using ags-read, ftp-connection-handle,
display message box
"Unable to connect to the FTP server. The connection was refused by the server. The port or IP address may be incorrect. Contact your system administrator.",
title "District Manager File Process" type mb-ok
icon mb-warning-icon
go to all-tasks-complete-process
end-if.
call "cSsocket" using ags-close, ftp-connection-handle.

call "cSsocket" using ags-read, ftp-connection-handle,
display message box
"A problem was encountered while attempting to create the FTP server. The connection was refused by the server. The port or IP address may be incorrect. Contact your system administrator.",
title "District Manager File Process" type mb-ok
icon mb-warning-icon
go to all-tasks-complete-process
end-if.
call "cSsocket" using ags-close, ftp-connection-handle.

end-if.
call "cSsocket" using ags-close, ftp-connection-handle.

end-if.
call "cSsocket" using ags-close, ftp-connection-handle.

end-perform.
call "cSsocket" using ags-read, ftp-connection-handle,
display message box
"A problem was encountered while attempting to create the FTP server. The connection was refused by the server. The port or IP address may be incorrect. Contact your system administrator.",
title "District Manager File Process" type mb-ok
icon mb-warning-icon
go to all-tasks-complete-process
end-if.
call "cSsocket" using ags-close, ftp-connection-handle.

end-perform.
call "cSsocket" using ags-read, ftp-connection-handle,
display message box
"A problem was encountered while attempting to create the FTP server. The connection was refused by the server. The port or IP address may be incorrect. Contact your system administrator.",
title "District Manager File Process" type mb-ok
icon mb-warning-icon
go to all-tasks-complete-process
end-if.
call "cSsocket" using ags-close, ftp-connection-handle.

end-perform.
call "cSsocket" using ags-read, ftp-connection-handle,
display message box
"A problem was encountered while attempting to create the FTP server. The connection was refused by the server. The port or IP address may be incorrect. Contact your system administrator.",
title "District Manager File Process" type mb-ok
icon mb-warning-icon
go to all-tasks-complete-process
end-if.
call "cSsocket" using ags-close, ftp-connection-handle.

end-perform.
call "cSsocket" using ags-read, ftp-connection-handle,
display message box
"A problem was encountered while attempting to create the FTP server. The connection was refused by the server. The port or IP address may be incorrect. Contact your system administrator.",
title "District Manager File Process" type mb-ok
icon mb-warning-icon
go to all-tasks-complete-process
end-if.
call "cSsocket" using ags-close, ftp-connection-handle.

end-perform.
call "cSsocket" using ags-read, ftp-connection-handle,
display message box
"A problem was encountered while attempting to create the FTP server. The connection was refused by the server. The port or IP address may be incorrect. Contact your system administrator.",
title "District Manager File Process" type mb-ok
icon mb-warning-icon
go to all-tasks-complete-process
end-if.
Appendix A: COBOL Legacy System

end-perform.
move ftp-receive-command to parse-receive-command.

if parse-command-code = '230' initialize ftp-send-command move 1 to ftp-end-command-length
string "PASS" delimited by = x$a
ftp-password delimited by spaces x'a0' delimited by end-of-record into ftp-end-command with pointer
ftp-end-receive-command-length subtract 1 from ftp-send-command-length

call "clsocket" using ags-write, ftp-connection-handle, ftp-send-command, ftp-send-command-length

call "clsocket" using ags-flush, ftp-connection-handle

else move 0 to task-6-act-visible move 0 to task-5-act-visible move 0 to task-4-act-visible move 0 to task-3-act-visible move 0 to task-2-act-visible move 0 to task-1-act-visible

endif.

move 'Sales' to ftp-directory

if parse-command-code = '250'
call 'cSsocket' using ags-write, ftp-connection-handle, ftp-send-command, ftp-send-command-length

call "clsocket" using ags-write, ftp-connection-handle, ftp-send-command, ftp-send-command-length

endif.

if parse-command-code = '230'
move '9998' to ftp-directory

endif.

display message box
* Unable to transmit manager 9998 file. The directory 'Sales/Updates' was unable to be reached. Contact your support administrator.
title 'District Manager File Process'
type mb-ok
icon mb-warn-dot

go to all-tasks-complete-process

end-perform.
move ftp-receive-command to parse-receive-command.

if parse-command-code = '230' move '9998' to ftp-directory

endif.

display message box
* Unable to transmit manager 9998 file. The directory 'Sales/Updates' was unable to be reached. Contact your support administrator.
title 'District Manager File Process'
type mb-ok
icon mb-warn-dot

go to all-tasks-complete-process

end-perform.
Appendix A: COBOL Legacy System

move 1 to num-bytes-read
string "PASS" delimited by space to ae
x'5A' delimited by space into ftp-send-command with pointer ftp-send-end-length
subtract 1 from ftp-send-end-length
end-perform.
move ftp-send-command to parse-receive-command.

if parse-command-code = '200'
move 1 to ftp-send-end-length
string "PASS" delimited by space to ae
x'5A' delimited by space into ftp-send-command with pointer ftp-send-end-length
end-perform.

move 0 to task-act-visible
move 0 to task-6-act-visible
move 1 to task-6-visible
move 1 to task-6-err-visible
move 1 to task-7-err-visible
move 1 to task-8-complete-visible
move 1 to ok-txt-status
display message box
"Unable to transmit manager 9998 file. The transmission was denied by the server. Contact your support administrator"

end-perform.

move 0 to task-act-visible
move 0 to task-6-act-visible
move 1 to task-6-visible
move 1 to task-6-err-visible
move 1 to task-7-complete-visible
move 1 to ok-txt-status
display message box
"Unable to transmit manager 9998 file. The file transmission was interrupted. Contact your support administrator"

end-perform.

move 0 to task-act-visible
move 0 to task-6-act-visible
move 1 to task-6-visible
move 1 to task-6-err-visible
move 1 to task-7-complete-visible
move 1 to ok-txt-status
display message box
"Unable to transmit manager 9998 file. The request for a data transmission port was denied by the server. Contact your support administrator"

end-perform.

move "Manager 9998" to ftp-file-path
initialize ftp-send-command
move 1 to ftp-send-end-length
string "STOP" delimited by space to ae
ftp-file-path delimited by space to ae
x'5A' delimited by space into ftp-send-command with pointer ftp-send-end-length
end-perform.

move ftp-receive-command to parse-receive-command
if parse-command-code = '205'
move 1 to ftp-send-end-length
string "PASS" delimited by space to ae
x'5A' delimited by space into ftp-send-command with pointer ftp-send-end-length
end-perform.

move 0 to task-act-visible
move 0 to task-6-act-visible
move 1 to task-6-visible
move 1 to task-6-err-visible
move 1 to task-7-complete-visible
move 1 to ok-txt-status
display message box
"Unable to transmit manager 9998 file. The transmission was interrupted. Contact your support administrator"

end-perform.

move 0 to task-act-visible
move 0 to task-6-act-visible
move 1 to task-6-visible
move 1 to task-6-err-visible
move 1 to task-7-complete-visible
move 1 to ok-txt-status
display message box
"Unable to transmit manager 9998 file. The file transmission was interrupted. Contact your support administrator"

end-perform.

move 0 to task-act-visible
move 0 to task-6-act-visible
move 1 to task-6-visible
move 1 to task-6-err-visible
move 1 to task-7-complete-visible
move 1 to ok-txt-status
display message box
"Unable to transmit manager 9998 file. The request for a data transmission port was denied by the server. Contact your support administrator"

end-perform.

move "Manager 9998" to ftp-file-path
initialize ftp-send-command
move 1 to ftp-send-end-length
string "QUIT" delimited by space to ae
ftp-file-path delimited by space to ae
x'5A' delimited by space into ftp-send-command with pointer ftp-send-end-length
end-read.
with pointer $s-end-length.

setup-unix-server-file-transfer-process.

if parse-command-code = "227" call "check" using $s-write, $f-connection-handle, $f-send-length.

end-perform.

end-perform.

add 1 to $s-length.

move 0 to task-2-visible.

move $s-end-length to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-ico-act-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.

move 0 to task-2-visible.

move 0 to task-2-ico-visible.
Appendix A: COBOL Legacy System

```cobol
move 'more custf' to user2-current-files(2).
when 7
end-perform.
perform until total-file-8 > filesize-cm
move total-file-8 to file-total-byte-size
end-if
perform until total-file-9 > filesize-cm
move total-file-9 to file-total-byte-size
when 8
move total-file-9 to file-total-byte-size
end-if
perform until total-file-10 > filesize-cm
move total-file-10 to file-total-byte-size
when 9
move total-file-10 to file-total-byte-size
end-if
perform until total-file-11 > filesize-cm
move total-file-11 to file-total-byte-size
end-perform.
move total-file-2 to file-total-byte-size
when 3
move total-file-2 to file-total-byte-size
end-perform.
move total-file-3 to file-total-byte-size
when 4
move total-file-3 to file-total-byte-size
end-perform.
move total-file-4 to file-total-byte-size
when 5
move total-file-4 to file-total-byte-size
end-perform.
move total-file-5 to file-total-byte-size
when 6
move total-file-5 to file-total-byte-size
end-perform.
move total-file-6 to file-total-byte-size
when 7
move total-file-6 to file-total-byte-size
end-perform.
move total-file-7 to file-total-byte-size
when 8
move total-file-7 to file-total-byte-size
end-perform.
move total-file-8 to file-total-byte-size
when 9
move total-file-8 to file-total-byte-size
end-perform.
move total-file-9 to file-total-byte-size
when 10
move total-file-9 to file-total-byte-size
end-perform.
move total-file-10 to file-total-byte-size
when 11
move total-file-10 to file-total-byte-size
end-perform.
move total-file-11 to file-total-byte-size
when 12
move total-file-11 to file-total-byte-size
end-perform.
move total-file-12 to file-total-byte-size
when 13
move total-file-12 to file-total-byte-size
end-perform.
move total-file-13 to file-total-byte-size
when 14
move total-file-13 to file-total-byte-size
end-perform.
move total-file-14 to file-total-byte-size
when 15
move total-file-14 to file-total-byte-size
end-perform.
move total-file-15 to file-total-byte-size
when 16
move total-file-15 to file-total-byte-size
end-perform.
move total-file-16 to file-total-byte-size
when 17
move total-file-16 to file-total-byte-size
end-perform.
move total-file-17 to file-total-byte-size
when 18
move total-file-17 to file-total-byte-size
end-perform.
move total-file-18 to file-total-byte-size
when 19
move total-file-18 to file-total-byte-size
end-perform.
move total-file-19 to file-total-byte-size
when 20
move total-file-19 to file-total-byte-size
end-perform.
move total-file-20 to file-total-byte-size
when 21
move total-file-20 to file-total-byte-size
end-perform.
move total-file-21 to file-total-byte-size
when 22
move total-file-21 to file-total-byte-size
end-perform.
move total-file-22 to file-total-byte-size
when 23
move total-file-22 to file-total-byte-size
end-perform.
move total-file-23 to file-total-byte-size
when 24
move total-file-23 to file-total-byte-size
end-perform.
move total-file-24 to file-total-byte-size
when 25
move total-file-24 to file-total-byte-size
end-perform.
move total-file-25 to file-total-byte-size
when 26
move total-file-25 to file-total-byte-size
end-perform.
move total-file-26 to file-total-byte-size
when 27
move total-file-26 to file-total-byte-size
end-perform.
move total-file-27 to file-total-byte-size
when 28
move total-file-27 to file-total-byte-size
end-perform.
move total-file-28 to file-total-byte-size
when 29
move total-file-28 to file-total-byte-size
end-perform.
move total-file-29 to file-total-byte-size
when 30
move total-file-29 to file-total-byte-size
end-perform.
move total-file-30 to file-total-byte-size
when 31
move total-file-30 to file-total-byte-size
end-perform.
move total-file-31 to file-total-byte-size
when 32
move total-file-31 to file-total-byte-size
end-perform.
move total-file-32 to file-total-byte-size
when 33
move total-file-32 to file-total-byte-size
end-perform.
move total-file-33 to file-total-byte-size
when 34
move total-file-33 to file-total-byte-size
end-perform.
move total-file-34 to file-total-byte-size
when 35
move total-file-34 to file-total-byte-size
end-perform.
move total-file-35 to file-total-byte-size
when 36
move total-file-35 to file-total-byte-size
end-perform.
move total-file-36 to file-total-byte-size
when 37
move total-file-36 to file-total-byte-size
end-perform.
move total-file-37 to file-total-byte-size
when 38
move total-file-37 to file-total-byte-size
end-perform.
move total-file-38 to file-total-byte-size
when 39
move total-file-38 to file-total-byte-size
end-perform.
move total-file-39 to file-total-byte-size
when 40
move total-file-39 to file-total-byte-size
end-perform.
move total-file-40 to file-total-byte-size
when 41
move total-file-40 to file-total-byte-size
end-perform.
move total-file-41 to file-total-byte-size
when 42
move total-file-41 to file-total-byte-size
end-perform.
move total-file-42 to file-total-byte-size
when 43
move total-file-42 to file-total-byte-size
end-perform.
move total-file-43 to file-total-byte-size
when 44
move total-file-43 to file-total-byte-size
end-perform.
move total-file-44 to file-total-byte-size
when 45
move total-file-44 to file-total-byte-size
end-perform.
move total-file-45 to file-total-byte-size
when 46
move total-file-45 to file-total-byte-size
end-perform.
move total-file-46 to file-total-byte-size
when 47
move total-file-46 to file-total-byte-size
end-perform.
move total-file-47 to file-total-byte-size
when 48
move total-file-48 to file-total-byte-size
end-perform.
move total-file-48 to file-total-byte-size
when 49
move total-file-49 to file-total-byte-size
end-perform.
move total-file-49 to file-total-byte-size
when 50
move total-file-50 to file-total-byte-size
end-perform.
move total-file-50 to file-total-byte-size
```
Appendix A: COBOL Legacy System

if division-remainder not = 0
    compute file-total-byte-size = total-file-9 - division-remainder
    move 1 to bytes-remaining-ew
else
    move total-file-9 to file-total-byte-size
    move 0 to bytes-remaining-ew
end-if
when 10
    move 1 to tot-10-cat
perform until tot-10-cat > file-size-cat
    move file-start-position(tot-10-cat) to tot-10-position(tot-10-cat)
    add 1 to tot-10-cat
end-perform
divide total-file-10 by 500 giving division-value
remainder division-remainder
if division-remainder not = 0
    compute file-total-byte-size = total-file-10 - division-remainder
    move 1 to bytes-remaining-ew
else
    move total-file-10 to file-total-byte-size
    move 0 to bytes-remaining-ew
end-if
end-evaluate.

all-tasks-complete-process.
move 0 to end-program.
display main-form.
perform until exit-dm-program = 1
accept main-form on exception continue
evaluate key-status
    when 999
    move 1 to exit-dm-program
when 96
evaluate event-type
    when end-close
    move 1 to exit-dm-program
end-evaluate
end-accept
end-perform.
exit program.
One HTML legacy system, which is named as Weather - ABC News, is regarded as the modelling example. It is some web pages that present the weather forecast through the Internet of News of Australian Broadcasting Corporation.
Appendix B: HTML Legacy System
Appendix B: HTML Legacy System

<table>
<thead>
<tr>
<th>Location</th>
<th>Temperature</th>
<th>Weather</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Islamabad, Pakistan</td>
<td>27°C</td>
<td>Partly cloudy</td>
<td></td>
</tr>
<tr>
<td>Jakarta, Indonesia</td>
<td>29°C</td>
<td>Rainy</td>
<td></td>
</tr>
<tr>
<td>Johannesburg, South Africa</td>
<td>29°C</td>
<td>Rainy</td>
<td></td>
</tr>
<tr>
<td>Johannesburg, South Africa</td>
<td>10°C</td>
<td>Snowy</td>
<td></td>
</tr>
<tr>
<td>Kuala Lumpur, Malaysia</td>
<td>24°C</td>
<td>Partly cloudy</td>
<td></td>
</tr>
<tr>
<td>Jakarta, Indonesia</td>
<td>24°C</td>
<td>Partly cloudy</td>
<td></td>
</tr>
<tr>
<td>Kuala Lumpur, Malaysia</td>
<td>24°C</td>
<td>Partly cloudy</td>
<td></td>
</tr>
<tr>
<td>Moscow, Russia</td>
<td>32°C</td>
<td>Snowy</td>
<td></td>
</tr>
<tr>
<td>New Delhi, India</td>
<td>25°C</td>
<td>Clear</td>
<td></td>
</tr>
<tr>
<td>New Delhi, India</td>
<td>26°C</td>
<td>Clear</td>
<td></td>
</tr>
<tr>
<td>New York, United States</td>
<td>23°C</td>
<td>Snowy</td>
<td></td>
</tr>
<tr>
<td>Sao Paulo, Brazil</td>
<td>26°C</td>
<td>Clear</td>
<td></td>
</tr>
<tr>
<td>Sao Paulo, Brazil</td>
<td>26°C</td>
<td>Clear</td>
<td></td>
</tr>
<tr>
<td>Rio de Janeiro, Brazil</td>
<td>25°C</td>
<td>Snowing shower</td>
<td></td>
</tr>
<tr>
<td>Rome, Italy</td>
<td>15°C</td>
<td>Snowing shower</td>
<td></td>
</tr>
<tr>
<td>Rome, Italy</td>
<td>20°C</td>
<td>Snowing shower</td>
<td></td>
</tr>
<tr>
<td>Rome, Italy</td>
<td>18°C</td>
<td>Snowing shower</td>
<td></td>
</tr>
<tr>
<td>Tokyo, Japan</td>
<td>23°C</td>
<td>Snowing shower</td>
<td></td>
</tr>
<tr>
<td>Tokyo, Japan</td>
<td>23°C</td>
<td>Snowing shower</td>
<td></td>
</tr>
<tr>
<td>Wellington, New Zealand</td>
<td>26°C</td>
<td>Snowing shower</td>
<td></td>
</tr>
<tr>
<td>Wellington, New Zealand</td>
<td>26°C</td>
<td>Snowing shower</td>
<td></td>
</tr>
</tbody>
</table>
Appendix B: HTML Legacy System

```html
<input type="submit" value="Search the ABC" title="Search the ABC">
```
Appendix B: HTML Legacy System

Appendix B: HTML Legacy System

Lismorc

Mostly sunny

Mostly sunny

Windy with showers

Windy with showers

Cleanng shower

Windy with showers

Windy with showers

Cleanng shower

Windy with showers

Cleanng shower

Windy with showers

Cleanng shower

Windy with showers

Cleanng shower

Windy with showers

Cleanng shower
Strong Wind Warning for Port Phillip and Western Port

Issued at 17.22PM on Wednesday 19 September 2007

Severe Wind Warning for East of Wilson Promontory

Issued at 15.07PM on Wednesday 19 September 2007

Severe Wind Warning for Localised Damaging Winds for the Alpine Districts.

Issued at 16.33PM on Wednesday 19 September 2007

Cleanng shower

Horsham

Mostly cloudy

Melbourne

Mostly cloudy

Mildura

Mostly cloudy

Northern Bass Strait

Mostly cloudy

Southern Bass Strait

Mostly cloudy

Western Bass Strait

Mostly cloudy

Appendix B: HTML Legacy System

<form action="http://search.abc.net.au/seaah/search.cgi" method="get" name="simple">
<input type="hidden" name="num_page" value="20" />
<input type="hidden" name="form" value="news" />
<input type="hidden" name="collection" value="abcnews" />
<input type="text" class="text" name="query" value="" size="15" />
</form>
Appendix B: HTML Legacy System
## Appendix B: HTML Legacy System

### Daily Weather Forecast

<table>
<thead>
<tr>
<th>Date</th>
<th>Weather Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wednesday</td>
<td>Mostly sunny</td>
</tr>
<tr>
<td>Thursday</td>
<td>Mostly sunny</td>
</tr>
<tr>
<td>Friday</td>
<td>Mostly sunny</td>
</tr>
<tr>
<td>Saturday</td>
<td>Mostly sunny</td>
</tr>
</tbody>
</table>

### Four Day Outlook

- **Wednesday**: Winds 10-15 knots, increasing to 20/30 knots on Friday mainly in the region northwest of Cairns in the morning. Lighter winds inshore early. Sea up to 1.6 m.
- **Thursday**: SE/E winds 10/15 knots though 15/20 knots are expected north of Cairns in the morning. Lighter winds inshore early. Sea up to 1.6 m. Winds easing to 10/15 knots east of Weipa in the afternoon. Inshore afternoon seabreezes. Sea up to 1.4 m.
- **Friday**: SE/E winds 10/15 knots. Winds easing to 10/15 knots inshore south of Weipa in the afternoon. Inshore afternoon seabreezes. Sea up to 1.8 m.
- **Saturday**: SW to SE winds 10/15 knots increasing to 20/30 knots on Friday mainly in coastal waters south of Fraser Island and inshore south of Weipa in the afternoon. Inshore afternoon seabreezes. Sea up to 1.4 m.

### State Centres Forecast

- **Darwin**: Winds 10-15 knots and increasing to 25/30 knots east of Weipa in the afternoon. Inshore afternoon seabreezes. Sea up to 1.4 m.
- **Townsville**: Winds 10-15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Cairns**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Mackay**: Winds 10-15 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Rockhampton**: Winds 10/15 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Bundaberg**: Winds 10-15 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Brisbane**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Peninsula Waters**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Southeast Coast**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Keppel Bay**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Great Barrier Reef**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Moreton Bay**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Hervey Bay**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Fog then sunny**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.

### State Coastal Waters

- **Great Barrier Reef**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Keppel Bay**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Great Barrier Reef**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Moreton Bay**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Hervey Bay**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Fog then sunny**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Great Barrier Reef**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Keppel Bay**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Great Barrier Reef**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Moreton Bay**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Hervey Bay**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
- **Fog then sunny**: Winds 10/15 knots increasing to 25/30 knots early afternoon ahead of a late afternoon 25-30 knot SW gale change. Sea up to 3.0 m.
Southeast Gulf

Wednesday: Variable winds 10 knots with inshore seabreezes 10/15 knots. Thursday: E/NE winds 10/15 knots, easing to 10 knots in the afternoon with inshore seabreezes. Sea: Up to 0.6 m.

Tropical Watten

Wednesday: SE/NE winds 10/15 knots, easing to 10/15 knots. Thursday: SE/NE winds 10/15 knots. Sea: Up to 1.2 m.
Appendix C: SQL Legacy System

One SQL legacy system, which is named as SimulatingINSERT, is regarded as the modelling example. It is one SQL example to create one statement that has the same function as the INSERT statement in SQL. The INSERT statement is presented as one procedure named as sp_generate_inserts in this legacy system. It is written by Narayana Vyas Kondreddi at http://vyaskn.tripod.com and tested on SQL Server 2000.

```sql
SET NOCOUNT ON
GO
PRINT 'Using Master database'
USE master
GO
PRINT 'Checking for the existence of this procedure'
IF (SELECT OBJECT_ID('sp_generate_inserts', 'P')) IS NOT NULL
BEGIN
PRINT 'Procedure already exists. So, dropping it'
DROP PROC sp_generate_inserts
END
GO
EXEC master.dbo sp_MS_sql般y schem category 1
GO
CREATE PROC sp_generate_inserts
@table_name varchar(776),
@target_table varchar(776) = NULL,
@include_column_list bit = 1,
@from varchar(800) = NULL,
@include_timestamp bit = 0,
@debug_mode bit = 0,
@owner varchar(64) = NULL,
@ommit_images bit = 0,
@ommit_identity bit = 0,
@top int = NULL,
@cols_to_include varchar(8000) = NULL,
@cols_to_exclude varchar(8000) = NULL,
@disable_constraints bit = 0,
@ommit_computed_cols bit = 0
AS
BEGIN
SET NOCOUNT ON
IF (@cols_to_include IS NOT NULL) AND (@cols_to_exclude IS NOT NULL)
BEGIN
RAISERROR('Use either @cols_to_include or @cols_to_exclude. Do not use both the parameters at once.', 16, 1)
RETURN -1
END
IF (@cols_to_include IS NOT NULL) AND (@include_column_list IS NOT NULL)
BEGIN
RAISERROR('Toward use of @cols_to_include property', 16, 1)
PRINT 'Specify column names surrounded by single quotes and separated by commas.'
SELECT @include_column_list = '$cols_to_include$'
RETURN -1
END
IF (@cols_to_exclude IS NOT NULL) AND (@include_column_list IS NOT NULL)
BEGIN
RAISERROR('Toward use of @cols_to_exclude property', 16, 1)
PRINT 'Specify column names surrounded by single quotes and separated by commas.'
SELECT @include_column_list = '$cols_to_exclude$'
RETURN -1
END
IF (PARSENAME(@table_name, 3)) IS NOT NULL
BEGIN
RAISERROR('Do not specify the database name. Be in the required database and just specify the table name.', 16, 1)
RETURN -1
END
IF (@owner IS NULL)
BEGIN
IF (OBJECT_ID(@table_name, 'U') IS NULL) AND (OBJECT_ID(@table_name, 'V') IS NULL)
BEGIN
RAISERROR('User table or view not found.', 16, 1)
PRINT 'You may see this error, if you are not the owner of the table or view. In that case use @owner parameter to specify the owner name.'
PRINT 'Make sure you have SELECT permission on that table or view.'
RETURN -1
END
ELSE
BEGIN
IF NOT EXISTS (SELECT 1 FROM INFORMATION_SCHEMA_TABLES WHERE TABLE_NAME = @table_name AND TABLE_TYPE = 'BASE TABLE' OR TABLE_TYPE = 'VIEW')
BEGIN
RAISERROR('User table or view not found.', 16, 1)
PRINT 'You may see this error, if you are not the owner of this table. In that case use @owner parameter to specify the owner name.'
PRINT 'Make sure you have SELECT permission on that table or view.'
RETURN -1
END
END
ELSE
BEGIN
SET @IDN = *
SET @Column_ID = *
SET @Column_Name = *
SET @Columns_list = *
SET @Actual_Values = *
IF (@owner IS NULL)
BEGIN
SET @Start_Insert = INSERT INTO ' + 
+ RTRIM(COALESCE(@target_table, @table_name)) + ' +
END
ELSE
BEGIN
SET @Start_Insert = INSERT INTO ' + 
+ RTRIM(COALESCE(@target_table, @table_name)) + '
+ T + 
+ T + RTRIM(COALESCE(@target_table, @table_name)) + ' +
END
SELECT @Column_ID = MIN(ORDINAL POSITION)
FROM INFORMATION_SCHEMA_COLUMNS (NOLOCK)
WHERE TABLE_NAME = @table_name AND
(OWNER IS NULL OR TABLE_SCHEMA = @owner)
WHERE @Column_ID IS NOT NULL
BEGIN
SET SELECT @Column_Name = QUOTENAME(COLUMN_NAME),
@DataType = DATA_TYPE
FROM INFORMATION_SCHEMA_COLUMNS (NOLOCK)
WHERE ORDINAL_POSITION = @Column_ID AND
TABLE_NAME = @table_name AND
(OWNER IS NULL OR TABLE_SCHEMA = @owner)
```
Appendix C: SQL Legacy System

columns
IF @cols_to_include IS NOT NULL
  -- Selecting only user specified columns
BEGIN
  IF CHARINDEX('=', @Column_Name) + 1 > 0
    BEGIN
      GOTO SKIP LOOP
    END
END
GOTO SKIP LOOP
IF @cols_toExclude IS NOT NULL
  -- Selecting only user specified columns
BEGIN
  IF CHARINDEX('=', @Column_Name) + 1 > 0
    BEGIN
      GOTO SKIP LOOP
    END
END
GOTO SKIP LOOP

IF (SELECT COLUMNPROPERTY(OBJECT_ID(QUOTENAME(COALESCE(@owner, USER_NAME))), @Column_Name) + 2, 'Identity') = 0
  BEGIN
  END
ELSE
  BEGIN
  END
IF (@include column_list = 0)
  BEGIN
    RETURN
  END
ELSE IF (@include column_list = 1)
  BEGIN
  END
BEGIN
  EXEC (@Actual_Values)
  GOTO SKIP LOOP
END
PRINT

PRINT 'Set NOCOUNT ON'
PRINT 'STOP'
IF (@debug_mode = 1)
  BEGIN
    PRINT '---------START OF DEBUG INFORMATION---------'
    PRINT 'Print Beginning of the INSERT statement:'
    PRINT '@Start_Insert'
    PRINT '
    PRINT 'The SELECT statement executed to generate the INSERT:
    PRINT '@Actual_Values'
    PRINT '---------END OF DEBUG INFORMATION----------'
  END
END
PRINT '---INSERTS generated by sp_generate_inserts stored procedure written by Vyys----'
PRINT 'Build number: 27'
PRINT '---Problems/Suggestions? Contact Vyys at vyys@hotmail.com---'
PRINT 'http://vyys.tripod.com'
PRINT 'STOP'
IF @dieable construan I AND (OBJECT_ID(QUOTENAME(COALESCE(@owner,USER_NAME)) + ' ' + @table_name, 'U') IS NOT NULL)
BEGIN
  IF @owner IS NULL
  BEGIN
    SELECT 'ALTER TABLE ' + QUOTENAME(COALESCE(@owner, USER_NAME)) + QUOTENAME(@table_name) + ' CHECK CONSTRAINT ALL' AS '--Code to enable the previously disabled constraints'
  END
  ELSE
  BEGIN
    SELECT 'ALTER TABLE ' + QUOTENAME(@owner) + QUOTENAME(COALESCE(@target_table, @table_name)) + ' CHECK CONSTRAINT ALL' AS '--Code to enable the previously disabled constraints'
  END
END
PRINT GO
PRINT 
IF (@SIGN <> ?)
BEGIN
  PRINT 'SET IDENTITY_INSERT ' + QUOTENAME(@owner, USER_NAME) + QUOTENAME(@table_name) + ' OFF
  PRINT GO'
END
PRINT SET NOCOUNT OFF
SET NOCOUNT OFF
RETURN 0 --Success. We are done
END
GO
PRINT 'Created the procedure
GO
EXEC master.dbo sp_MS_set_object_category 2
GO
PRINT 'Granting EXECUTE permission on sp_generate_inserts to all users'
GRANT EXEC ON sp_generate_inserts TO public
SET NOCOUNT OFF
GO
PRINT 'Done'
Appendix D: List of Publications by Author


