REVERSE ENGINEERING
ENCAPSULATED COMPONENTS
FROM LEGACY CODE

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2018

By
Rehman Arshad
School of Computer Science
Contents

Abstract ........................................ 14
Declaration ...................................... 15
Copyright ....................................... 16
Acknowledgements ............................... 17
Publication List ................................. 19

1 Introduction .................................. 20
  1.1 Research Problem ............................ 21
  1.2 Research Questions and Objectives .......... 25
  1.3 Research Methodology ....................... 26
      1.3.1 Identification of Research Problem .... 27
      1.3.2 Development of Prototype .............. 27
      1.3.3 Evaluation ................................ 28
  1.4 Research Contributions ..................... 28
  1.5 Thesis Overview ............................. 30

2 Reverse Engineering ......................... 32
  2.1 Introduction ................................ 32
  2.2 Classification based on Analysis Technique .. 34
      2.2.1 Reverse Engineering based on Static Analysis .. 34
          2.2.1.1 Non-Textual Static Analysis ............ 34
          2.2.1.2 Textual Static Analysis ............... 35
      2.2.2 Reverse Engineering based on Dynamic Analysis .. 36
      2.2.3 Reverse Engineering based on Hybrid Analysis .. 37
      2.2.4 State-of-the-art Approaches Based on Analysis Type .. 37
5.2.1 Mapping Rule Base and Scenarios of Interactions of Methods

5.2.1.1 Single Class with no interaction

5.2.1.2 Two Classes with public-public methods interaction

5.2.1.3 Two Classes with private-public OR public-private methods interaction

5.3 Mapping Rule Base for Control Statements

5.3.1 Mapping the if and if-else statements

5.3.2 Mapping the Switch statements

5.3.3 Mapping the Loop based statements

5.4 Implementing the Brake Control System

5.5 Summary

6 Reverse Engineering X-MAN

6.1 Introduction

6.2 Capturing the Code Base

6.2.1 Nodes Structuring by RX-MAN Parser

6.3 Implementing the Code Mapper

6.3.1 Example: Clustering of Brake Control System

6.3.2 Mapping from Clusters to Components

6.4 Summary

7 Implementing the RX-MAN Tool

7.1 Introduction

7.2 Tool Overview

7.3 Implementing the Technology Stack

7.3.1 Comparison With Previous X-MAN Tool

7.4 RX-MAN Tool: Features

7.4.1 Design Phase Composition

7.4.2 Reverse Engineering of Components

7.4.3 Deployment Phase Composition

7.5 Implementing the Brake Control System by RX-MAN Tool

7.6 Summary

8 Evaluation of RX-MAN

8.1 Introduction

8.2 Evaluating RX-MAN By Empirical Validation and Evaluation
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Reverse Engineering: Classification Based on Analysis Type</td>
<td>43</td>
</tr>
<tr>
<td>2.2</td>
<td>Reverse Engineering: Classification Based on Required Input</td>
<td>45</td>
</tr>
<tr>
<td>2.3</td>
<td>Reverse Engineering: Classification Based on Generated Output</td>
<td>47</td>
</tr>
<tr>
<td>2.4</td>
<td>Reverse Engineering: Classification based on Implementation Language</td>
<td>49</td>
</tr>
<tr>
<td>3.1</td>
<td>A Taxonomy of Component Models ([LW05])</td>
<td>57</td>
</tr>
<tr>
<td>3.2</td>
<td>Component Models Based on the Composition Mechanism</td>
<td>66</td>
</tr>
<tr>
<td>8.1</td>
<td>Reverse Engineering of Draw.io by Package Level Clustering</td>
<td>150</td>
</tr>
<tr>
<td>8.2</td>
<td>Reverse Engineering of Draw.io by Class Level Clustering</td>
<td>151</td>
</tr>
<tr>
<td>8.3</td>
<td>Reverse Engineering of EverNote SDK by Package Level Clustering</td>
<td>154</td>
</tr>
<tr>
<td>8.4</td>
<td>Reverse Engineering of EverNote SDK by Class Level Clustering</td>
<td>155</td>
</tr>
<tr>
<td>8.5</td>
<td>Reverse Engineering of JabRef by Package Level Clustering</td>
<td>158</td>
</tr>
<tr>
<td>8.6</td>
<td>Reverse Engineering of JabRef by Class Level Clustering</td>
<td>159</td>
</tr>
<tr>
<td>8.7</td>
<td>Reverse Engineering of JGraph5 by Package Level Clustering</td>
<td>161</td>
</tr>
<tr>
<td>8.8</td>
<td>Reverse Engineering of JGraph5 by Class Level Clustering</td>
<td>162</td>
</tr>
<tr>
<td>8.9</td>
<td>Reverse Engineering of TeamMates by Package Level Clustering</td>
<td>164</td>
</tr>
<tr>
<td>8.10</td>
<td>Reverse Engineering of TeamMates by Class Level Clustering</td>
<td>165</td>
</tr>
<tr>
<td>8.11</td>
<td>Evaluation Statistics by Using Package-Level Clustering</td>
<td>167</td>
</tr>
<tr>
<td>8.12</td>
<td>Evaluation Statistics by Using Class-Level Clustering</td>
<td>168</td>
</tr>
<tr>
<td>8.13</td>
<td>Informational Evaluation (a)</td>
<td>172</td>
</tr>
<tr>
<td>8.14</td>
<td>Informational Evaluation (b)</td>
<td>173</td>
</tr>
<tr>
<td>8.15</td>
<td>Informational Evaluation (c)</td>
<td>173</td>
</tr>
<tr>
<td>8.16</td>
<td>Evaluation by By-Product (a)</td>
<td>174</td>
</tr>
<tr>
<td>8.17</td>
<td>Evaluation by By-Product (b)</td>
<td>174</td>
</tr>
<tr>
<td>8.18</td>
<td>Evaluation by By-Product (c)</td>
<td>175</td>
</tr>
<tr>
<td>8.19</td>
<td>RX-MAN Versus Component Directed Approaches</td>
<td>177</td>
</tr>
<tr>
<td>8.20</td>
<td>Comparison With Approaches based on Explicit Components</td>
<td>178</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Generic Component</td>
<td>23</td>
</tr>
<tr>
<td>1.2</td>
<td>Components extracted from Heat Management System</td>
<td>24</td>
</tr>
<tr>
<td>1.3</td>
<td>Composition of Heat Management System</td>
<td>24</td>
</tr>
<tr>
<td>1.4</td>
<td>Research Methodology ([KP05])</td>
<td>26</td>
</tr>
<tr>
<td>1.5</td>
<td>Thesis Outline</td>
<td>31</td>
</tr>
<tr>
<td>2.1</td>
<td>Reverse Engineering: Framework of Classification</td>
<td>50</td>
</tr>
<tr>
<td>3.1</td>
<td>An Idealised Component Life Cycle ([LdC17a])</td>
<td>55</td>
</tr>
<tr>
<td>3.2</td>
<td>Koala Component Notation</td>
<td>60</td>
</tr>
<tr>
<td>3.3</td>
<td>Classification of Components based on Composition Units and Composition Mechanism</td>
<td>61</td>
</tr>
<tr>
<td>3.4</td>
<td>Adapter Connectors of X-MAN Component Model</td>
<td>63</td>
</tr>
<tr>
<td>3.5</td>
<td>X-MAN Component Model</td>
<td>64</td>
</tr>
<tr>
<td>3.6</td>
<td>X-MAN Meta-Model for Atomic Component</td>
<td>64</td>
</tr>
<tr>
<td>3.7</td>
<td>Composition By Sequencer</td>
<td>65</td>
</tr>
<tr>
<td>4.1</td>
<td>Overview of the Presented Approach</td>
<td>70</td>
</tr>
<tr>
<td>5.1</td>
<td>RX-MAN: Mapping from Object-Oriented Code to X-MAN</td>
<td>74</td>
</tr>
<tr>
<td>5.2</td>
<td>Scenario 1</td>
<td>75</td>
</tr>
<tr>
<td>5.3</td>
<td>Re-structured Code (Cluster) from Figure 5.2</td>
<td>76</td>
</tr>
<tr>
<td>5.4</td>
<td>Re-structured Code for Utility Cluster from Figure 5.2</td>
<td>76</td>
</tr>
<tr>
<td>5.5</td>
<td>Scenario 1: Restructured Code VS X-MAN Entities</td>
<td>77</td>
</tr>
<tr>
<td>5.6</td>
<td>Mapped Components from Scenario 1</td>
<td>78</td>
</tr>
<tr>
<td>5.7</td>
<td>Scenario 2</td>
<td>79</td>
</tr>
<tr>
<td>5.8</td>
<td>Re-structured Code (Cluster) from Figure 5.7</td>
<td>79</td>
</tr>
<tr>
<td>5.9</td>
<td>Mapped Component from Scenario 2</td>
<td>80</td>
</tr>
<tr>
<td>5.10</td>
<td>Scenario 3</td>
<td>80</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.11</td>
<td>Re-structured Code (Cluster 1) from Figure 5.10</td>
<td>81</td>
</tr>
<tr>
<td>5.12</td>
<td>Re-structured Code (Cluster 2) from Figure 5.10</td>
<td>81</td>
</tr>
<tr>
<td>5.13</td>
<td>Composition of Mapped Atomic Components from Scenario 3</td>
<td>82</td>
</tr>
<tr>
<td>5.14</td>
<td>Mapped Utility Component from Scenario 3</td>
<td>83</td>
</tr>
<tr>
<td>5.15</td>
<td>Simple <em>if Statement</em></td>
<td>85</td>
</tr>
<tr>
<td>5.16</td>
<td>Control Flow for Figure 5.15</td>
<td>86</td>
</tr>
<tr>
<td>5.17</td>
<td>Re-structured Code (Cluster 1) from Figure 5.15</td>
<td>86</td>
</tr>
<tr>
<td>5.18</td>
<td>Re-structured Code (Cluster 2) from Figure 5.15 for First Atomic Component of the Composite</td>
<td>87</td>
</tr>
<tr>
<td>5.19</td>
<td>Re-structured Code (Cluster 2) from Figure 5.15 for Second Atomic Component of Composite</td>
<td>87</td>
</tr>
<tr>
<td>5.20</td>
<td>Mapped Atomic and Composite Component for Simple <em>if Statement</em></td>
<td>88</td>
</tr>
<tr>
<td>5.21</td>
<td><em>if-else Statement</em></td>
<td>89</td>
</tr>
<tr>
<td>5.22</td>
<td>Control Flow for Figure 5.21</td>
<td>90</td>
</tr>
<tr>
<td>5.23</td>
<td>Re-structured Code (Cluster 1) from Figure 5.21</td>
<td>91</td>
</tr>
<tr>
<td>5.24</td>
<td>Re-structured Code (Cluster 2) from Figure 5.21 for First Atomic Component of Composite</td>
<td>91</td>
</tr>
<tr>
<td>5.25</td>
<td>Re-structured Code (Cluster 2) from Figure 5.21 for Second Atomic Component of Composite</td>
<td>92</td>
</tr>
<tr>
<td>5.26</td>
<td>Re-structured Code (Cluster 2) from Figure 5.21 for Third Atomic Component of Composite</td>
<td>92</td>
</tr>
<tr>
<td>5.27</td>
<td>Re-structured Code (Cluster 2) from Figure 5.21 for Fourth Atomic Component of Composite</td>
<td>92</td>
</tr>
<tr>
<td>5.28</td>
<td>Mapped Atomic and Composite Component for <em>if-else Statement</em></td>
<td>93</td>
</tr>
<tr>
<td>5.29</td>
<td><em>Switch Statement</em></td>
<td>95</td>
</tr>
<tr>
<td>5.30</td>
<td>Control Flow for Figure 5.29</td>
<td>96</td>
</tr>
<tr>
<td>5.31</td>
<td>Re-structured Code (Cluster 1) from Figure 5.29</td>
<td>96</td>
</tr>
<tr>
<td>5.32</td>
<td>Re-structured Code (Cluster 2) from Figure 5.29 for First Atomic Component of Composite</td>
<td>97</td>
</tr>
<tr>
<td>5.33</td>
<td>Re-structured Code (Cluster 2) from Figure 5.29 for Second Atomic Component of Composite</td>
<td>97</td>
</tr>
<tr>
<td>5.34</td>
<td>Re-structured Code (Cluster 2) from Figure 5.29 for Third Atomic Component of Composite</td>
<td>97</td>
</tr>
<tr>
<td>5.35</td>
<td>Re-structured Code (Cluster 2) from Figure 5.29 for Fourth Atomic Component of Composite</td>
<td>98</td>
</tr>
</tbody>
</table>
5.36 Re-structured Code (Cluster 2) from Figure 5.29 for fifth Atomic Component of Composite ........................................... 98
5.37 Mapped Atomic and Composite Component for if-else Statement With Nested switch .................................................. 98
5.38 While Loop with if Statement .............................................. 100
5.39 Control Flow for Figure 5.38 .............................................. 101
5.40 Re-Structured Code (Cluster 1) from Figure 5.38 ................. 101
5.41 Re-structured Code (Cluster 2) from Figure 5.38 for First Atomic Component of the Composite ........................................ 102
5.42 Re-structured Code (Cluster 2) from Figure 5.38 for Second Atomic Component of the Composite ........................................ 102
5.43 Re-structured Code (Cluster 2) from Figure 5.38 for Third Atomic Component of the Composite ........................................ 102
5.44 Mapped Atomic and Composite Component for While Loop With if-else Statement .................................................. 103
5.45 Mapped Atomic and Composite Component for do While Loop With if-else Statement .................................................. 104
5.46 Nested for Loop ......................................................... 106
5.47 Control Flow for Figure 5.46 .............................................. 107
5.48 Re-Structured Code (Cluster 1) from Figure 5.46 ................. 107
5.49 Re-Structured Code (Cluster 2) from Figure 5.46 for First Atomic Component of Composite ........................................ 108
5.50 Re-Structured Code (Cluster 2) from Figure 5.46 for Second Atomic Component of Composite ........................................ 108
5.51 Re-Structured Code (Cluster 2) from Figure 5.46 for Third Atomic Component of Composite ........................................ 108
5.52 Mapped Atomic and Composite Component for Nested for Loop ................................................................. 109
5.53 Brake Control System ..................................................... 110
5.54 Brake Control System: Cluster 1 ........................................ 111
5.55 Brake Control System: Cluster 2 ........................................ 112
5.56 Brake Control System: Utility Cluster ..................................... 113
5.57 Composition of Brake Control System .................................... 114
5.58 Utility Component from Cluster 2 ........................................ 114

6.1 RX-MAN: Capturing and Mapping the Code Base. ......................... 116
6.2 RX-MAN Parser .............................................................. 119
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3</td>
<td>Clustering of Brake Control System</td>
<td>124</td>
</tr>
<tr>
<td>7.1</td>
<td>RX-MAN: Technology Stack</td>
<td>135</td>
</tr>
<tr>
<td>7.2</td>
<td>RX-MAN: Selection of Atomic Design to Construct and Deposit Atomic Components</td>
<td>138</td>
</tr>
<tr>
<td>7.3</td>
<td>RX-MAN: Design Phase Composition</td>
<td>139</td>
</tr>
<tr>
<td>7.4</td>
<td>RX-MAN: Code Generation of an Atomic Component</td>
<td>139</td>
</tr>
<tr>
<td>7.5</td>
<td>RX-MAN: Generation of RX-MAN Re-Structured Nodes</td>
<td>140</td>
</tr>
<tr>
<td>7.6</td>
<td>RX-MAN: Selection of Atomic Design to Construct and Deposit Atomic Components</td>
<td>141</td>
</tr>
<tr>
<td>7.7</td>
<td>RX-MAN: Extraction of Components</td>
<td>142</td>
</tr>
<tr>
<td>7.8</td>
<td>RX-MAN: Retrieval of Components from CDO</td>
<td>143</td>
</tr>
<tr>
<td>7.9</td>
<td>RX-MAN: Composition of Brake Control System</td>
<td>145</td>
</tr>
<tr>
<td>7.10</td>
<td>RX-MAN: Retrieval of Composed Brake Control System</td>
<td>146</td>
</tr>
<tr>
<td>8.1</td>
<td>RX-MAN: Validation/Evaluation</td>
<td>149</td>
</tr>
<tr>
<td>8.2</td>
<td>Example of Code-Independent Composition in Draw.io</td>
<td>152</td>
</tr>
<tr>
<td>8.3</td>
<td>Instantiation of Composite Component after Composition/Deposition of Extracted Components</td>
<td>153</td>
</tr>
<tr>
<td>8.4</td>
<td>Example of Code-Independent Composition in EverNote SDK</td>
<td>156</td>
</tr>
<tr>
<td>8.5</td>
<td>Instantiation of Composite Component after Composition/Deposition of Extracted Components</td>
<td>157</td>
</tr>
<tr>
<td>8.6</td>
<td>Example of Code-Independent Composition in JabRef</td>
<td>159</td>
</tr>
<tr>
<td>8.7</td>
<td>Example of Code-Independent Composition in JGraph5</td>
<td>162</td>
</tr>
<tr>
<td>8.8</td>
<td>Instantiation of Composite Component after Composition/Deposition of Extracted Components</td>
<td>163</td>
</tr>
<tr>
<td>8.9</td>
<td>Example of Code-Independent Composition in TeamMates</td>
<td>165</td>
</tr>
<tr>
<td>8.10</td>
<td>Instantiation of Composite Component after Composition/Deposition of Extracted Components</td>
<td>166</td>
</tr>
<tr>
<td>8.11</td>
<td>Semantic Dimensions</td>
<td>170</td>
</tr>
<tr>
<td>8.12</td>
<td>Gannod and Cheng's Framework of Evaluation</td>
<td>171</td>
</tr>
<tr>
<td>9.1</td>
<td>RX-MAN Mapping</td>
<td>189</td>
</tr>
<tr>
<td>B.1</td>
<td>Guard</td>
<td>212</td>
</tr>
<tr>
<td>B.2</td>
<td>Loop</td>
<td>212</td>
</tr>
<tr>
<td>B.3</td>
<td>Sequencer</td>
<td>213</td>
</tr>
</tbody>
</table>
Abstract

Component-based development is an approach that revolves around the construction of systems form pre-built modular units (components). If legacy code can be reverse engineered to extract components, the extracted components can provide architectural re-usability across multiple systems of the same domain.

Current component directed reverse engineering approaches are based on component models that belong to architecture description languages (ADLs). ADL-based components cannot be reused without configurational changes at code level and binding every required and provided service. Moreover, these component models neither support code-independent composition after extraction of components nor the redeposition of a composed configuration of components for future reuse.

This thesis presents a reverse engineering approach that extracts components and addresses the limitations of current approaches, together with a tool called RX-MAN. Unlike ADL-based approaches, the presented approach is based on an encapsulated component model called X-MAN. X-MAN components are encapsulated because computation cannot go outside of a component. X-MAN components cannot interact directly but only exogenously (composition is defined outside of a component). Our approach offers code-independent composition after extracting components and does not need binding of all the services like ADLs. The evaluation of our approach shows that it can facilitate the re-usability of legacy code by providing code-independent composition and re-deposition of composed configurations of components for further reuse and composition.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University’s policy on presentation of Theses.
Acknowledgements

I would like to thank:
My late Father for every sacrifice.
My Life Partner and Mother for love.
My Brothers and Family for support.
My Research Fellows for memorable evenings.
&
Last but not least, My Supervisor and Mentor Kung-Kiu Lau for endless discussions, guidance and support. He taught me many things and giving up is not one of them.
A Thesis to the memory of my late Father!
Publication List


Chapter 1

Introduction

“Curiosity is the wick in the candle of learning.”
— William Arthur Ward

The term re-usability is not new for software developers and programmers. With evolution of software methodologies and product centric development, it is almost indispensable to reuse the development resources to their maximum potential. The term legacy systems usually refers to such software systems that are outdated, lack proper documentation and cannot support a new feature without breaking another logic, yet they are vital to an organisation [Ben95]. Unfortunately, most legacy systems that were designed with non-modular approach cannot exploit the luxury of re-usability to good extent [Fea04]. In most cases, they even cannot offer adequate comprehension of implementation and design, due to poor maintenance over the years. Lack of proper documentation and poor maintenance are responsible for the corrosion of such software systems. Sooner or later, poorly maintained and inadequately documented systems would reach a stage after which it would become practically impossible for the programmers to introduce a new feature in the system without breaking the overall logic of implementation. Such systems cannot provide re-usability and they are important yet costly to maintain for the companies. Code of such software systems is known as legacy code. Legacy code is also defined as a piece of code that is difficult to change or a code without tests [Fea04].

In the domain of software engineering, survival of such systems is dependent on the retrieval of their comprehension by a process called reverse engineering. The ultimate goal of reverse engineering is to provide an abstraction of comprehension of the legacy code that can help in understanding, maintaining or re-engineering the system.
For many companies, maintenance or comprehension of legacy code is crucial because some of their functions are too valuable to be discarded and too expensive to reproduce from scratch. According to [Bas97], 50-60% of software engineering effort is spent trying to understand the source code.

Reverse engineering is mostly used to extract a higher abstraction notation which is semantically equivalent to the source system (legacy code) or partial source system in some way but is much easier to understand. Semantic extraction can help in analysing the system for maintenance but cannot reconstruct it for reuse e.g., reverse engineering has been used in product line engineering and information systems to efficiently manage variability models [AL16] and identify feature locations in the source code [EKS03]. Neither feature locations nor the variability models can reconstruct the legacy code in a reusable manner. More semantic extraction notations like these include feature models, UML diagrams, graphical notations and concept lattices [AL16]. To reconstruct a legacy code base for re-usability, one needs an architectural notation of extraction that should be modular, reusable and executable.

Components can serve as such reusable modular architectural units. Component based development is a domain that revolves around the construction of systems from pre-built software units i.e., re-usability. A component is defined as "A unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition." [Szy02]. Therefore, if legacy code can be reverse engineered to obtain components (using semantics of a component model), one can get a notation of extraction that can provide architectural re-usability of legacy code (in the same domain). Instead of just analysing the system, such approach can reuse the extracted modular structures of legacy code across multiple systems and can support new features in the system itself without breaking another logic.

The real challenge is to get code-independent re-usability i.e. components should not be dependent on the manual configuration changes at code level for their reuse.

### 1.1 Research Problem

In order to introduce the research problem, it is important to understand few basic concepts related to a component model. The degree of re-usability of components
depends directly on two architectural properties of the extracted components: deposit-ability and re-composability [AL18]. Deposit-ability requires a repository for depositing components and retrieving them, whilst re-composability requires a component model [LW07, LdC17a] with well-defined composition mechanisms for components. Composition mechanisms are semantics by which a component model composes components together [LSST11]. Considering these parameters of re-usability:

1. Few component models do not support repository. It means components cannot be deposited after extraction and have to be used only once as they are (e.g., ArchJava [ACN02]). Such components cannot provide re-usability as they cannot be deposited and retrieved for reuse.

2. Few component models have a repository that does not allow retrieval (e.g., EJB [EK01]). Such components can be extracted from legacy code and can be deposited but cannot be retrieved for composition as retrieval is not allowed.

3. The component models that are most widely used are based on architecture description languages (ADLs) [MT00]. Components of these models can be deposited and retrieved to be reused later.

Almost all the current reverse engineering approaches that extract components are based on ADLs (e.g., [ARA+09] ). ADLs define required and provided services as ports (composition mechanism of ADLs). Ports use (indirect) method calls at code level to compose components together [ASM04, BBSK05]. ADL-based components have three major shortcomings from re-usability point of view:

1. In ADL-based components, one cannot select/de-select/alter ports without changing the code manually at all required places to compose the components after retrieval. This re-configuration of ports at code level is a manual and complex task. Each different integration of components demands different configuration to reuse the components. Especially, in case of legacy systems, where adequate comprehension of implementation is not expected, re-configuration at code level is much more complex and time consuming.

2. ADL-based components can only be retrieved to use as is. One has to bind all the ports in order to use an ADL component. Therefore, ADL-based components provide non-flexible re-usability.

3. It is not possible to re-deposit a configured integration of components for reuse (e.g., composite component). Components have to be retrieved and configured

---

1The term re-deposit-ability means ability to re-deposit the composed components after retrieval e.g., retrieve the deposited components A and B, compose them as A.B and deposit the A.B for future reuse. The composition after re-deposition is referred as re-composition.
1.1. RESEARCH PROBLEM

Consider the description of a heat management system below to demonstrate the stated shortcomings:

Consider a home heating system which consists of a master switch, a thermostat, a boiler and a status indicator. The master switch is used to turn on/off the system. The thermostat is used to set the desired temperature in the house. The thermostat also contains a temperature sensor that measures the current temperature, and a controller that (i) calculates the difference between the current temperature and the desired temperature and (ii) sets the heat level for the boiler required to maintain the desired temperature. The status indicator continuously takes readings from the boiler. If the temperature of the boiler is above the safe value, the status indicator instructs the master switch to turn off the system for a specified time interval; otherwise the system will keep on working.

Figure 1.1 is showing the notation of a generic component with required and provided services. These required and provided services are handled by input and output ports of ADL-based component models.

Suppose, a reverse engineering approach extracts ADL-based components from the code base of heat management system as shown in the Figure 1.2. One possible way to compose the reverse engineered components is shown in Figure 1.3. The configuration in Figure 1.3 involves the composition of 6 required and 8 provided services (excluding the HMS Status component as it is external to the system). It means one valid configuration of legacy code demands 14 configuration changes at code level to reuse the extracted components. In case of addition of new functionality in the system, these configurations will even increase because more services will be required to reuse and compose these components with the new ones. In case of legacy systems, extracted components can range from few dozens to hundreds [AL18]. Neither it is practically feasible nor efficient to configure hundreds of ports for every valid configuration of the system. It reflects our first point about shortcomings of ADLs and establishes the
Secondly, all the required and provided ports of an ADL-based component need to be closed (every required port has to be connected with a provided port and vice versa) to use the component. Therefore, required configuration changes at code level cannot be less than the total number of ports of all the components involved in a composition. Last but not the least, assume that the configuration of the system presented in Figure 1.3 is required to be reused with some other system (e.g., House Wiring Management System). One has to extract and re-configure all these ports again as ADL-based component models cannot re-deposit the heat management system for further reuse [LDC17b]. This shortcoming establishes our final point.

---

Figure 1.2: Components extracted from Heat Management System

To the best of our knowledge, no such component directed \(^2\) reverse engineering approach exists that can offer the following attributes:

\(^2\)By component directed we mean a reverse engineering approach that extracts components.
1) composition of the reverse engineered components without changing the code at all required places.
2) reuse of the components without binding all required and provided services.
3) re-deposition of the composed components for future reuse of the same integrated configuration.

To summarise, what is needed to facilitate better re-usability in legacy systems is a component model and a reverse engineering approach that can: deposit the extracted components, retrieve the components for reuse, compose the components without requiring any changes at code level, reuse components without binding all the dependencies and re-deposit the composed components (atomic or composite) for further reuse and composition.

### 1.2 Research Questions and Objectives

This thesis proposes a component directed reverse engineering approach that will address the limitations of current state-of-the-art approaches. In accordance with the research problem, this research is based on the following principle components: 1) a component model with a composition mechanism that does not need the code level configuration for reusing the components. The component model should also allow re-deposition and reuse of the composed components after composition. 2) an analysis (reverse engineering) technique that can capture and map a legacy code base to the semantics of selected component model.

The key research questions of this research thus are:

- whether such a component model exists or not, what should be the attributes of such a component model and whether an analysis technique can be designed to capture the legacy code bases and reverse engineered such components?
- to find out that how well a code base can be re-structured to reduce coupling as part of the reverse engineering process?

To satisfy the stated research questions, the primary aim of the research is to investigate the feasibility of establishing a component directed reverse engineering approach that can provide code-independent re-usability and composition of re-deposited components. To achieve this aim, following objectives have to be fulfilled:

1. the proposed reverse engineering approach that can map the legacy code to a higher abstraction notation (components) without losing any functionality in the process.
2. development of a reverse engineering tool, based on the proposed approach.
3. to evaluate whether or not the proposed research could provide code-independent re-usability and composition of the re-deposited components.

The secondary aim of the research is to find out how well a code base can be restructured to reduce coupling in the original source code. To satisfy this aim, the objective is to find out the possibility of a component model that can provide maximum cohesion by not allowing coupling of control statements in a code base. Therefore, this aim is also primarily related to the analysis of component models and component directed reverse engineering.

### 1.3 Research Methodology

The research methodology employed in this research is illustrated in Figure 1.4 and it is adopted from [KP05]. This methodology is the combination of software engineering and research. That is why the selected methodology is not only theoretical but empirical as well. The steps of this methodology are explained as follows:

![Figure 1.4: Research Methodology ([KP05])](image)
1.3. RESEARCH METHODOLOGY

1.3.1 Identification of Research Problem

In this stage, research problems are identified by background work and research questions are justified. In addition to existing research artefacts, other domains should also be consulted and explored to validate the potential solutions and their validity. If proposed potential solutions define a new way of doing something in the research domain, validity of the solutions has to be proved via prototype. An identified research problem is considered worthy if it can add new knowledge to an already established domain or establishes a new domain.

In case of our research, it has been identified after a comprehensive study of the domains of reverse engineering and component based development that the extraction of components is not as well-practiced area as the extraction of semantics from a code base by reverse engineering. It is due to the fact that none of the component models can offer easy re-usability because one still have to understand the code to make configuration changes in the extracted components. Therefore, the extracted components need to offer re-usability without demanding any code changes.

1.3.2 Development of Prototype

At this stage, research problems should be resolved by presenting the potential solutions via prototype. Development of prototype consists of following phases:

- **Conceptual Development**: It includes the development of a conceptual framework to narrow down the requirements and understanding of the building process. This framework should investigate and present functionalities of the prototype that can prove validity of the potential solutions.
- **Architecture Development**: It includes identification of system components, their interaction and integration.
- **Analysis/Design**: In this phase, design of data structures and databases are defined along with the proper system specifications.
- **Implementation**: This stage uses the conceptual development framework to utilise the architecture for implementation, based on the analysis/design choices.

For the proposed approach, the conceptual development consists of mapping of object oriented code constructs to the constructs of our component model using a defined rule base. Our conceptual framework defines the road map to capture, parse and transform the object oriented code into easy to reuse components.

The architecture development consists of a defined meta-model of our component
model along with the architectural notation of our rule base for reverse engineering.

The analysis/design of our prototype includes the identification of data types of the constructs of our component model, compatibility with a run time repository for our component model and compatibility of the available tool support e.g., parser.

Implementation phase of our prototype includes the coding of our defined meta-model and rule base (architecture) using the selected tools and designed choices (during analysis/design) to satisfy the conceptual framework (working of our approach).

1.3.3 Evaluation

During this stage, prototype is evaluated and tested. Results are analysed to establish the difference between expected and actual output. The results should be evaluated based on the conceptual framework and system specifications. Results can determine the required changes in conceptual framework, architecture or design specifications. The cycle from evaluation to prototyping is repeated until phases of prototypes are good enough to satisfy the research problems. If the research problems are satisfied and solutions are validated, the end result is research contributions.

Due to empirical nature of the research, we have evaluated our research by following ways:

1. By using empirical evaluation of five actual legacy code bases.
3. By using Bellay and Gall's framework [BG98] to evaluate the capability of our tool.
4. By a direct comparison with other related approaches.

1.4 Research Contributions

This research led to the following contributions:

• A reverse engineering approach that provides code-independent re-usability:
  The proposed approach is different and provides better re-usability than other ADL-based approaches because it provides re-usability of the extracted components without demanding any code-level changes. RX-MAN also does not enforce the binding of all the ports of all the components involved in a composition but only requires the order of execution of exposed methods and data
1.4. RESEARCH CONTRIBUTIONS

parameters by using click and drag interfaces. The reverse engineered components in RX-MAN can be re-deposited after code-independent composition to preserve the semantics of a composition for future reuse. Such a re-deposition is very efficient and time saving in constructing and enhancing new systems by reusing the legacy code.

- **A taxonomy of Reverse Engineering:** The proposed research presented a taxonomy of reverse engineering. All the relevant approaches aimed at legacy systems were classified based on their analysis type, programming languages they could reverse engineer, output notation and the input notation each approach required. Such a taxonomy is practical and precise in deciding the suitability of an approach for reverse engineering a code base.

- **Classification and analysis of Component Models:** The proposed research presented a taxonomy of component models. One of the major achievements of this research was to integrate the domain of component based development with reverse engineering to show the impact of one on the other and vice versa.

Along with these contributions, this research led to the publication of three conference papers and one journal paper. The three papers covered the classification of reverse engineering [AL16], reverse engineering of architectural notation [AL17] and reverse engineering of encapsulated components from object-oriented legacy code [AL18]. Hence, the first paper is related to literature review in the domain of reverse engineering (Chapter 2), the second one is related to preliminary results of our approach (Chapter 5) and the third one is related to the proposed rule base for reverse engineering the components (Chapter 5, 6 and 7). The journal paper is related to overall approach in detail along with the comprehensive evaluation (Chapter 5-8).

In terms of our research methodology (Figure 1.4), taxonomies of reverse engineering and component based development are related to Development of New Artefacts and New Domain Knowledge whereas, the first contribution is related to System Specification and Prototype System as our approach provides an independent reverse engineering tool that can reverse engineer any object oriented legacy code. System Specification in our first contribution includes specification of a component model’s meta-model, specification of a defined parser and specifications of a process that starts by capturing the code and ends after depositing components into a repository.
CHAPTER 1. INTRODUCTION

1.5 Thesis Overview

This thesis has been organised into following eight chapters:

- **Chapter 2** (Reverse Engineering) includes related approaches in the domain of reverse engineering. It is not limited to only those approaches that extract components but also include those that produce some other output notation yet their target is legacy systems. This chapter presents a taxonomy of reverse engineering and also compares our approach with other similar approaches.

- **Chapter 3** (Component Based Development) includes the basics of component based development, architectural properties of component models, classification of component models, a taxonomy of component models and detailed explanation of the component model selected for reverse engineering the legacy code. Difference in the architectural properties of the selected component model with other models motivates and justifies its selection, as these architectural properties are responsible for the current limitations in component directed reverse engineering.

- **Chapter 4** (Overview of RX-MAN: A Novel Reverse Engineering Approach) provides a road map for our component directed reverse engineering approach. It presents an overview of the approach and identifies theoretical and practical steps of the approach.

- **Chapter 5** (Object-Oriented Code VS X-MAN) introduces the mapping between object-oriented legacy code and entities of the selected component model. This chapter explains the basic rules of mapping with examples to show the equivalence between object-oriented code and extracted components.

- **Chapter 6** (Reverse Engineering X-MAN) introduces the customised parser and the proposed way to capture a code base without losing any functionality. This chapter also includes the semantics and algorithms that are used for code capturing, restructuring and formation of components using the rule base of Chapter 5.

- **Chapter 7** (RX-MAN Tool) introduces the prototype tool designed for this research. This chapter includes the technology stack and tool demonstration using an example.

- **Chapter 8** (Evaluation of RX-MAN) includes five legacy code bases for validation and empirical evaluation. This chapter also evaluates the proposed approach by discussing the results in terms of research aims and objectives and compares the proposed approach with others using evaluation frameworks.
• Chapter 9 (Conclusions and Future Work) summarises the contributions and concludes the thesis. This chapter also discusses the research limitations and proposes a direction for future aspects of this research.

Figure 1.5 illustrates the thesis structure. Overall, thesis can be classified into four themes: Background/Related Work, Approach Overview, Main Contributions and Validation/Evaluation.
Chapter 2

Reverse Engineering

“If you wish to make an apple pie from scratch, you must first invent the universe.”

— Carl Sagan, Cosmos

2.1 Introduction

"Reverse engineering can be viewed as a process of analysing a system to: identify the system's components and their interrelationships, create representation of the system in another form or at a higher level of abstraction" [CC90].

Reverse Engineering can be categorised into Re-documentation and Design Recovery. Re-documentation is defined as "A creation or revisions of a semantically equivalent representation within the same relative abstraction level” [Ram01]. Re-documentation can provide easier ways to visualize relationships among program's components. Design Recovery is defined as "A subset of Reverse Engineering in which domain knowledge, external information, deduction and fuzzy reasoning are added” [Ram01]. Design Recovery provides more meaningful abstractions than self-examination of the system. Output of Design Recovery is always at higher level of abstraction than the input. Component directed reverse engineering is different from general reverse engineering because component's extraction from a source code is Re-documentation from semantic point of view but higher abstraction of components as compared to the source code makes it a kind of Design Recovery as well.
2.1. INTRODUCTION

In this chapter, we introduce the current state-of-the-art reverse engineering approaches related to re-documentation or design recovery. We also present a taxonomy of reverse engineering based on analysis technique, output notation, required input notation and language of implementation [AL16]. A taxonomy based on these parameters can define the applicability of an approach according to one's requirements. This taxonomy can be used as a guide to filter appropriate approaches that can meet the requirements in terms of programming languages, nature of the output they produce and the notation of input they require e.g., one can easily filter only those approaches that are based on Java programming language or one can only consider those approaches that produce restructured code as an output notation. The presented taxonomy shows the relation between a specific analysis type and nature of the output it produces. Such a classification can help in understanding the domain of reverse engineering by grouping the shortcomings and advantages of each class of this taxonomy.

Section 2.2 describes the differences in analysis techniques of reverse engineering, relative benefits and limitations and state-of-the-art approaches. Section 2.3, 2.4 and 2.5 classify the reverse engineering approaches based on the required input notations, notations of their output and the programming languages they can reverse engineer respectively.
2.2 Classification based on Analysis Technique

Reverse engineering is generally classified into static, dynamic and hybrid reverse engineering, based on the underlying analysis technique. An analysis is the predefined methodology or a strategy to conduct reverse engineering for specific set of goals [DRGP13]. An analysis technique can determine the working and methodology of a reverse engineering approach. Each type of analysis has its own benefits and limitations. For component directed reverse engineering, it is important to analyse the relevancy and appropriateness of each analysis type and choose one to solve the stated research problem.

2.2.1 Reverse Engineering based on Static Analysis

Static reverse engineering is based on the static information of the code (non-behavioural). Static reverse engineering approaches consider control flow, data flow and dependencies in the code to usually identify the clusters of code-locations/features that belong together. Static approaches can be further classified into textual and non-textual analysis based approaches.

2.2.1.1 Non-Textual Static Analysis

Non-textual static approaches are based on structural information of the code. These approaches determine each possible route of the program by building a model of all possible states at each control instruction. Such a model is called the static abstraction model and it requires a fair consideration between preciseness and granularity [Ern04]. As these techniques consider all control flows, they provide the maximum recall.

These approaches consider every possible control path of the source code therefore, they are strong candidates for component directed reverse engineering. Extracted components have to keep the original semantics of a legacy code intact, that is why it is important to preserve every single functionality of the code. This can only be achieved by considering all the possible control routes of a code base.

These techniques are based on the control structure of the source code and therefore, they have very good recall but the biggest drawback is the lack of precision. False positive results are very common in static techniques because these techniques work on user-defined model of control flow rather than the actual trace of the program. Therefore, many parts of the code that are considered to be part of a cluster/feature are actually false positive results and it is impossible to separate them from the actual
results. The biggest advantage is the future re-usability. The static model can be used even after code modifications and it can be re-factored iteratively to produce better results. Moreover, a well-defined static abstraction model can be applied on multiple code bases.

2.2.1.2 Textual Static Analysis

As the name suggests, the textual analysis is dependent on the identifiers and comments in the source code. Textual analysis does not need any abstraction model and uses the query-based input to match the words with identifiers and comments in the code. Textual techniques usually convert source code into a corpus of documents and use some ranking mechanism to extract relevant portions of the code with respect to the terms of features, used in the query. Query designing is very crucial in textual analysis and there are many techniques that can exploit this power to use textual analysis for bug localisation [PM07] or for finding concept locations in the source code [LKE08]. "A concept location identifies parts of a software system that implement a specific concept that originates from the problem or the solution domain" [MSRM04]. Concept location is also known as feature location in many domains e.g., product lines.

Most textual analysis approaches of reverse engineering produce feature locations as an output. These code locations are displayed either by concept lattices (if Formal Concept Analysis is used) or by dependency graphs. Textual analysis helps instead of the non-textual analysis when there is no model available to get abstracted notation from the code and users have to be dependent on queries in order to extract the results. Like non-textual, textual analysis also has some drawbacks. The biggest problem is the user designed queries that are responsible for almost the whole analysis. Quality of the queries determine the quality or accuracy of the output. Another problem is polysemy and implicit implementation of a feature. A feature may compose of code units that are scattered across many places of the code. In such cases, it is very difficult for textual analysis to extract all the implicit locations of the code related to that feature.

Textual based reverse engineering is usually used to extract information about a specific functionality from the code base [BVD08]. These approaches are good for analysing the legacy systems from variability or from a feature point of view but they are not structured enough to transform the whole code base into components. Textual analysis is highly dependent on the code identifiers and comments and legacy systems usually do not cover the highest standards of programming practices.

Textual analysis can be conducted in three following ways [DRGP13]:

- **Formal Concept Analysis:** Uses concept lattices to represent the relationships between concepts and features.
- **Dependency Graphs:** Represents the relationships between code elements and their dependencies.
- **Query-Based Techniques:** Uses customizable queries to extract specific information from the code.
1. Pattern Matching
2. Information Retrieval (IR)
3. Natural Language Processing (NLP)

Pattern matching involves a search based on the patterns of words [PRV08]. Most general example is `grep` command in Linux. IR is widely used in search engines and there are hundreds of algorithms that can implement information retrieval. Latent semantic indexing (LSI) or vector space indexing are two prime examples that are used in almost every search engine. IR is more precise than pattern matching. Pattern matching cannot be much precise as it is just a search based on the text pattern whereas, IR is based on a query that matches code identifiers and comments with addition of some deduction logic, [MSRM04]. IR based approaches can produce better queries by using result feedback as part of iterative re-factoring. NLP based analysis is also like IR i.e. dependent on the query but query in NLP is in natural language [AT10].

### 2.2.2 Reverse Engineering based on Dynamic Analysis

Dynamic analysis uses execution trace of a program to follow and identify the concept locations. It imposes the first bottleneck i.e., system should be able to run and execute if one has to use dynamic reverse engineering. Test scenarios or profiling is needed in order to design an execution trace with respect to some feature/concept. Profiling is instrumentation of the code and it is a difficult task. Usually, one scenario can only involve one feature (functionality) therefore, in case of hundreds of features, dynamic analysis becomes more complex. Profiling on its own is a time consuming task that demands system comprehension and most legacy systems can be anything but comprehensible [Ben95, Fea04].

There is no uncertainty in the control paths like non-textual static analysis because process is dependent on the actual run time behaviour. This gives great precision to dynamic analysis but the price is lack of re-usability. Designing or profiling an execution trace is highly dependent on the programmer and it can highly affect the results. For every new profiling, old results are useless whereas, in non-textual static analysis we can reuse the rules of abstraction as many times as we want with continuous refinement.

Output of dynamic analysis is always a trace that shows feature locations in the code [EDV05]. This relation is represented either as concept lattices or view-based tools. The abstraction level of code in the trace varies from approach to approach.
2.2. CLASSIFICATION BASED ON ANALYSIS TECHNIQUE

2.2.3 Reverse Engineering based on Hybrid Analysis

A hybrid analysis in reverse engineering can be a combination of any preceding analysis techniques. A hybrid analysis does not carry the parallel execution of two different analyses. Hybrid analysis always uses one kind of analysis before the other in order to reinforce the accuracy of results and produce results by removing shortcomings of individual analysis techniques [EAAG08]. This accuracy however, comes at the price of complexity in hybrid techniques. Each kind of analysis also brings its shortcomings that have to be balanced against the advantages of a hybrid methodology.

Hybrid analysis can reinforce recall of non-textual static analysis with precision of dynamic analysis. Recall is required in order to make dynamic analysis reusable in future. A static analysis can obviate the collection of certain information and dynamic can be run over that collection in order to get better results. Hybrid techniques provide feature locations either by using concept lattices or view-based tools e.g., [RHLR08].

2.2.4 State-of-the-art Approaches Based on Analysis Type

Components extraction from the legacy systems is not as well-practised as the extraction of non-architectural notations i.e. feature models, feature traces, documents variability models etc. [ZB13, XXJ12]. This is mainly due to the fact that reverse engineering is mostly used to get higher abstraction notation that can analyse the system rather than reconstruct it. Another reason is the complexity of re-configuring the extracted components at code level in order to reuse them (Research Problem 1). This section includes state-of-the-art reverse engineering approaches belong to each analysis type.

2.2.4.1 Non-Textual Static Approaches

For the approaches based on non-textual static analysis, major focus of reverse engineering is on getting a configuration matrix, dependency graph or re-formation of the source code. These approaches are also used to extract a dependency matrix between source code and features in order to understand the relation between implementation and high level design. Non-textual static analysis is mostly used in domains like product lines and information systems [XXJ12, ZB14]. Almost all the component directed reverse engineering approaches belong to this category of analysis. High recall and ability to capture every single line of the source code without any instrumentation
CHAPTER 2. REVERSE ENGINEERING

makes this analysis type an ideal candidate for component directed reverse engineering.

RecoVar[ZB13] is a non-textual static analysis approach that is used to extract code based variability. Its output is a variability graph that shows correlation with the actual code. Such output is useful in finding the implicit code locations that are related to implementation of a feature.

FL-PV [ZHP+14] is another non-textual static reverse engineering approach that is used to transform the code into mandatory and optional parts. Mandatory is the part of code used by all the features for their implementation. Output of this approach is the transformed code. Semi-Automatic Approach[VBP12] also produces the same output but instead of transforming code, it just highlights the mandatory and optional parts of the code.

Concern Graph[RM02], Concern Identification[Tri09] and Dependency Graph[CR10] take source code of a system and produce concept lattices that show variability relationship between code and defined concept locations. These approaches demand some system comprehension.

Automatic Generation[Rob05] takes the source code and extracts feature to code trace.

RECAST[EM93] is another approach that takes COBOL source code as an input and produces the physical system design, used for future implementation of the system. The physical design notation is referred as Structured Systems Analysis and Design Method (SSADM).

Burd and Munro's approach [BM98] analyse a code base by clustering the data based on the evolution of interactions of data elements. This study showed the evolution of a code base over time, and increased complexity in maintainability.

Ward and Bennet's approach is based on transformation of a system by using wide spectrum language (WSL)[WB93]. The tool this approach presents is interactive and semi-automatic. WSL includes low level programming constructs and can generate a specification or code in high level language that is based on a formal theory. This approach is different from others as it is based on assembler code rather than code written in a programming language.

All the non-textual static approaches above are somehow related to comprehension of the system. Extraction of components demands way more than variability trace or concept locations identification.

Output of non-textual static component directed reverse engineering is dependent
on the definition of component each approach uses. Few like us, follow the szyperski’s definition of components. This definition defines component as “A unit of composition with contractually specified interfaces and explicit context dependencies only” [Szy02]. There are following approaches that follow the strict definition for extracted components.

JAVACompExt [ARA+09] is a heuristic based approach that extracts Abstract Data Type (ADT) components. The purpose of this research is to avoid the system corrosion by making architecture explicit in the source code. This approach also shows the behavioural aspects and relations between components via explicit interfaces.

The approach by Antoun et al. [AAAC07] re-engineers Java code into ArchJava components [ACN02]. This approach identifies the underlying architecture of the source code in UML notation and re-structures it to ArchJava components using a set of rules.

Chouambe et al. [CKK08] produces composite components from Java source code. The process includes the formation of an instance of the presented meta-model in component directed form after extracting source code metrics from the source code (source code metrics include interdependencies, weighted calls etc.).

Pattern-based Reverse Engineering of Design Components [KSRP99] extracts design components based on the structural descriptions of design patterns.

A Reverse Engineering Approach to Subsystem Structure Identification [MOTU93] is another approach that re-structures the system into a hierarchy of subsystems along with their high-level abstract representation as components. This approach uses composition measures (coupling, cohesion etc.) and composition relations (composition dependency graph) to cluster and map the components along with their explicit interfaces (required and provided nodes of dependency graph).

Archimetrix [DPB14] is an approach that can reconstruct architecture in the form of components from the source code after removing design deficiencies. This approach extracts abstract syntax graph (ASG) from the source code and uses ASG to extract system architecture. Components from this architecture are extracted using software model extractor (SoMoX) [RBH+16]. Design deficiencies are identified by matching with predefined deficiency patterns in ASG.

Quality centric approach [KSCC12] focuses on quality of explicit interfaces by following a semantic-correctness model. A mapping model is mapped from the source code (object-oriented concepts) to component based concepts (interfaces, sub-components etc.) and evaluated using the semantic-correctness model. The semantic-correctness
model uses fitness functions (measurable metric) to measure the semantics-correctness of a component.

Components extraction in memory-constrained environments [WF14] is an approach that identifies reusable part of an object-oriented code and re-factors the relative or surrounded code to reuse the identified part. This approach follows the JavaBeans component model [EK01]. Re-usability of code is offered by introducing a facade class that encapsulates the functionality and deals with the outside classes.

L2CBD [KC06] stands for legacy systems to component based development and this approach is different from others because it is a methodology rather than a concrete approach itself. Any proposed approach can use L2CBD methodology to transform legacy code into components. This methodology consists of planning phase, re-engineering phase, componentization phase and component testing.

Approach of Alshara et al. [AST+15] recovers architecture of object-oriented source code as SOFA [BHP06] or OSGi [CH02] components. The approach is consisted of architecture recovery (clustering) and mapping to the selected component language (ADLs in case of SOFA and OSGi framework for OSGi components).

The major shortcomings with most of these approaches are lack of automation [AAAC07], lack of repository and dependency on "componentization" [ARA+09, CKK08]. ¹ Secondly, all these approaches can extract components but none of them have the ability to achieve re-composition or code independent re-usability of the extracted components after retrieval. These approaches however, extract components with defined required and provided services, based on the semantics of a component model.

Unlike the approaches above, there are component directed approaches that use loose definition of component. For them, a component is consisted of methods that belong together as they offer specific functionality of the system. Such components can be giant classes, clusters or re-formation of the source code to get better cohesion and loose coupling. These approaches neither define the extraction of explicit interfaces nor the composition mechanism of the extracted components. Following reverse engineering approaches are based on loose definition of components.

---

¹Componentization is the process of atomizing resources into separate reusable packages to recombine them easily [Pol07] but most of the legacy systems were not implemented this way.
2.2. CLASSIFICATION BASED ON ANALYSIS TECHNIQUE

CORE [MKM09] stands for component directed reverse engineering. This approach extracts architectural information and services from legacy systems. It identifies the components in a source code by considering structural, sequence and use-case diagram of the system. Sequence diagrams provide behaviour of the system and structural provide static relations in a code base. End result is the identification of components with associated methods in them. Explicit interfaces or composition semantics are not provided. It is not wrong to say that this approach identifies potential components by clustering the methods together rather than extracting explicit reusable components.

Favre et al. [FDE+01] uses the concept of object modeller to propose a new component model for reverse engineer software product lines. The modeller was built via meta-model and it could map the object-oriented concepts (interface, extends etc.) to an abstract syntax that can form components.

Bunch Tool [MS06] is a component directed reverse engineering framework that uses heuristics to extract and refine the graph space from legacy code. This approach extracts a cyclic graph from the code base, clusters the graph nodes using heuristics and detaches the clustered nodes into associated parts of the code. The output of this approach is not the explicit components but associated clustered parts of a legacy code that can separate the concerns.

Systematic method approach [KC04] is a UML based approach that associates components with use cases, sequence calls and classes involved in those calls. Output of this approach is a set of clusters. Each cluster has set of classes that are allocated based on the degree of cohesion and coupling in the source code (association, aggregation, inheritance and composition relation between classes is considered).

Approach of Erdemir et al. [ETB11] presents a clustering algorithm for object-oriented legacy code. The presented algorithm creates a directed graph from the source code. It considers associations, extends relations, method calls and presence of class variables (e.g., variable of class A in class B) in the code. The end result is a set of clusters with associated methods together, based on cohesion and coupling.

Shatnawi et al. extracts reusable API-based components from object-oriented APIs [SSSA17]. This approach identifies every group of classes that can be considered as an implementation of an API (a component), then it identifies how these components should be organised (interfaces of components). The group of classes that form a component are based on source code structure and FUP (frequently usable pattern). FUPs are identified by analysing the interactions between an API and its clients (application
that uses the API). This approach is quite comprehensive because it is based not only on control structure but on FUPs as well. The interface extraction is not explicit i.e., it is based on the client application that is being used to interact with an API.

All the above component directed reverse engineering approaches (whether they provide explicit interfaces or non-explicit clusters) have same predefined order of steps. Every approach starts with capturing the code base in some notation. Based on a set of rules, formation of clusters is the second step (by using graph dependencies, directed graphs, code metrics etc.). The last step is the mapping from extracted structure to the semantics of a component model (if approach is explicit) or an architectural notation.

2.2.4.2 Textual Static Approaches

Textual static analysis is most suitable for finding a specific pattern or implementation of a feature in a code base. It is impossible to capture the whole code base using a textual approach alone. Textual analysis always needs an input query that can extract the required results. Quality of results are dependent on sophistication of an approach. It is quite common to have implicit feature implementation in legacy systems i.e. the relevant code can be distributed across many places and it is almost impossible for textual analysis to map the whole functionality to components without losing any attribute.

Reverse engineering based on textual analysis is mainly used for bug-localisation [LKE08] and finding feature locations in the source code [PM07] (These code locations are displayed either by concept lattices (if Formal Concept Analysis is used) or by dependency graphs).

Few textual approaches produce feature models from the provided description e.g., [LHGB+12, AMHS+14].

Few textual approaches also extract and show the variability in legacy code e.g., [XXJ12] and few provide code comprehension in non-technical language e.g., [AT10].

2.2.4.3 Dynamic Approaches

Dynamic analysis is dependent on the execution trace of the program. In case of legacy code, neither every feature can be executed nor it is feasible to design comprehensive profiling for the whole code base. Code evolution and addition of features make such profiling a repetitive and complex task. To the best of our knowledge, there is no such component directed reverse engineering approach that works on dynamic analysis alone.

Output of dynamic analysis is always a trace that shows feature locations in the
### 2.2. CLASSIFICATION BASED ON ANALYSIS TECHNIQUE

Table 2.1: Reverse Engineering: Classification Based on Analysis Type

<table>
<thead>
<tr>
<th>No. Of Approaches</th>
<th>Approach Name</th>
<th>Analysis Type</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static = 26</td>
<td>RECoVar</td>
<td>Static</td>
<td>[ZB13]</td>
</tr>
<tr>
<td></td>
<td>Language Independent Approach</td>
<td>Static</td>
<td>[ZHP^14]</td>
</tr>
<tr>
<td></td>
<td>Dependence Graph</td>
<td>Static</td>
<td>[CR10]</td>
</tr>
<tr>
<td></td>
<td>Concern Graph</td>
<td>Static</td>
<td>[RM02]</td>
</tr>
<tr>
<td></td>
<td>Concern Identification</td>
<td>Static</td>
<td>[Trl09]</td>
</tr>
<tr>
<td></td>
<td>Automatic Generation</td>
<td>Static</td>
<td>[Rob05]</td>
</tr>
<tr>
<td></td>
<td>Semi-Automatic Approach for Extraction</td>
<td>Static</td>
<td>[VBP12]</td>
</tr>
<tr>
<td></td>
<td>JAVACompExt</td>
<td>Static</td>
<td>[ARA^09]</td>
</tr>
<tr>
<td></td>
<td>Antoun <em>et al.</em></td>
<td>Static</td>
<td>[AAAC07]</td>
</tr>
<tr>
<td></td>
<td>RECAST</td>
<td>Static</td>
<td>[EM93]</td>
</tr>
<tr>
<td></td>
<td>Chouambe <em>et al.</em></td>
<td>Static</td>
<td>[CKK08]</td>
</tr>
<tr>
<td></td>
<td>Design Components</td>
<td>Static</td>
<td>[KSRP99]</td>
</tr>
<tr>
<td></td>
<td>Subsystem Structure Identification</td>
<td>Static</td>
<td>[MOTU93]</td>
</tr>
<tr>
<td></td>
<td>Archimetrix</td>
<td>Static</td>
<td>[DPB14]</td>
</tr>
<tr>
<td></td>
<td>Quality centric approach</td>
<td>Static</td>
<td>[KSCC12]</td>
</tr>
<tr>
<td></td>
<td>Memory-constrained environments</td>
<td>Static</td>
<td>[WF14]</td>
</tr>
<tr>
<td></td>
<td>L2CBD</td>
<td>Static</td>
<td>[KCO6]</td>
</tr>
<tr>
<td></td>
<td>CORE</td>
<td>Static</td>
<td>[MKM09]</td>
</tr>
<tr>
<td></td>
<td>Ward and Bennet’s approach</td>
<td>Static</td>
<td>[WB93]</td>
</tr>
<tr>
<td></td>
<td>Favre <em>et al.</em></td>
<td>Static</td>
<td>[FDE^01]</td>
</tr>
<tr>
<td></td>
<td>Burd and Munro’s approach</td>
<td>Static</td>
<td>[BM98]</td>
</tr>
<tr>
<td></td>
<td>Bunch Tool</td>
<td>Static</td>
<td>[MS06]</td>
</tr>
<tr>
<td></td>
<td>Systematic method approach</td>
<td>Static</td>
<td>[KCO4]</td>
</tr>
<tr>
<td></td>
<td>Erdemir <em>et al.</em></td>
<td>Static</td>
<td>[ETB11]</td>
</tr>
<tr>
<td></td>
<td>Shatnawi <em>et al.</em></td>
<td>Static</td>
<td>[SSSA17]</td>
</tr>
<tr>
<td></td>
<td>Alshara <em>et al.</em></td>
<td>Static</td>
<td>[AST^15]</td>
</tr>
<tr>
<td>Textual = 7</td>
<td>Product Variants</td>
<td>Textual</td>
<td>[XXJ12]</td>
</tr>
<tr>
<td></td>
<td>Natural language Parsing</td>
<td>Textual</td>
<td>[AI10]</td>
</tr>
<tr>
<td></td>
<td>Evolutionary Algorithms</td>
<td>Textual</td>
<td>[LHOB12]</td>
</tr>
<tr>
<td></td>
<td>Software Configurations using FCA</td>
<td>Textual</td>
<td>[AMHS14]</td>
</tr>
<tr>
<td></td>
<td>Source Code Retrieval</td>
<td>Textual</td>
<td>[LKE08]</td>
</tr>
<tr>
<td></td>
<td>Combining FCA with IR</td>
<td>Textual</td>
<td>[PM07]</td>
</tr>
<tr>
<td></td>
<td>Reverse Engineering Feature Models</td>
<td>Textual</td>
<td>[SLB11]</td>
</tr>
<tr>
<td>Dynamic = 9</td>
<td>Dynamic Feature Traces</td>
<td>Dynamic</td>
<td>[EDV05]</td>
</tr>
<tr>
<td></td>
<td>STRADA</td>
<td>Dynamic</td>
<td>[EBG07]</td>
</tr>
<tr>
<td></td>
<td>Call-Graphs</td>
<td>Dynamic</td>
<td>[BD06]</td>
</tr>
<tr>
<td></td>
<td>Focused views on Execution Traces</td>
<td>Dynamic</td>
<td>[BVDP08]</td>
</tr>
<tr>
<td></td>
<td>Concept Analysis</td>
<td>Dynamic</td>
<td>[EKS01]</td>
</tr>
<tr>
<td></td>
<td>Trace Dependency Analysis</td>
<td>Dynamic</td>
<td>[Egy03]</td>
</tr>
<tr>
<td></td>
<td>Scenario Driven Dynamic Analysis</td>
<td>Dynamic</td>
<td>[SMADP06]</td>
</tr>
<tr>
<td></td>
<td>Featureous</td>
<td>Dynamic</td>
<td>[OJ10]</td>
</tr>
<tr>
<td></td>
<td>Software Evolution Analysis</td>
<td>Dynamic</td>
<td>[GDG05]</td>
</tr>
<tr>
<td>Hybrid = 7</td>
<td>Static and Dynamic Analysis</td>
<td>Hybrid</td>
<td>[RHLR08]</td>
</tr>
<tr>
<td></td>
<td>Heuristic-Based Approach</td>
<td>Hybrid</td>
<td>[ADPAG10]</td>
</tr>
<tr>
<td></td>
<td>Landmarks and Barriers</td>
<td>Hybrid</td>
<td>[WRW07]</td>
</tr>
<tr>
<td></td>
<td>Component-oriented Architecture</td>
<td>Hybrid</td>
<td>[ASSF11]</td>
</tr>
<tr>
<td></td>
<td>Locating Features in Source Code</td>
<td>Hybrid</td>
<td>[EKS03]</td>
</tr>
<tr>
<td></td>
<td>SNIAFL</td>
<td>Hybrid</td>
<td>[ZZL^06]</td>
</tr>
<tr>
<td></td>
<td>Cerberus</td>
<td>Hybrid</td>
<td>[EAAG08]</td>
</tr>
</tbody>
</table>
code. This relation is represented either as concept lattices or view-based tools. The abstraction level of code in the trace varies from approach to approach. Dynamic Feature Traces [EDV05], Feature to code trace [EBG07], Locating and Understanding Features [BVD08], Software evolution Analysis [GDG05], Trace Dependency Analysis [Egy03], Featureous [OJ10], Embedded Call-Graphs [BD06], Scenario-Driven Dynamic Analysis [SMADP06] and Concept Analysis [EKS01] are few examples of approaches that are based on dynamic analysis.

2.2.4.4 Hybrid Approaches

Hybrid analysis provides feature locations either by using concept lattices or view-based tools. Static and Dynamic Analysis [RHLR08], Cerberus [EAAG08], Sniat [ZZL+06], Locating Features in Source Code[EKS03], Using Landmarks and Barriers [WRW07] and A Heuristic-Based Approach [ADPAG10] are few approaches based on hybrid analysis.

Component-oriented Architecture [ASSF11] is a component directed reverse engineering approach that uses static analysis to capture the code base and dynamic scenarios to capture the interactions between methods using heuristics. Based on the results, the relevant chunks of code are classified into clusters and object-oriented interfaces are identified from each cluster. This approach extracts OSGi components that interact with one another using direct method calls.

Extraction of components from legacy systems is neither identification of feature locations nor the representation of a variability model. It is transformation of the whole code base into an executable yet higher abstraction notation. The executable notation is a notation that depicts behaviour of the system along with the data computation [AL17]. Due to this major difference, non-textual static analysis is the best option to conduct a component directed reverse engineering. Table 2.1 shows the classification of state-of-the-art reverse engineering approaches based on analysis type.

2.3 Classification based on Required Input

The input notation required by reverse engineering approaches can be classified into source code, feature-set (code metrics etc.) and description based input (queries, document-corpus, textual input). Most of the non-textual static reverse engineering approaches require source code as an input notation. Most of the approaches that are
### 2.3. CLASSIFICATION BASED ON REQUIRED INPUT

<table>
<thead>
<tr>
<th>No. of Approaches</th>
<th>Approach Name</th>
<th>Required Input</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RECoVar</td>
<td>Source Code</td>
<td>[ZB13]</td>
</tr>
<tr>
<td></td>
<td>Call-Graphs</td>
<td>Source Code</td>
<td>[BD06]</td>
</tr>
<tr>
<td></td>
<td>Concern Identification</td>
<td>Source Code</td>
<td>[Tri09]</td>
</tr>
<tr>
<td></td>
<td>Scenario Driven Dynamic Analysis</td>
<td>Source Code</td>
<td>[SMADP06]</td>
</tr>
<tr>
<td></td>
<td>Burd and Munro’s approach</td>
<td>Source Code</td>
<td>[BM98]</td>
</tr>
<tr>
<td></td>
<td>Featureous</td>
<td>Source Code</td>
<td>[OJ10]</td>
</tr>
<tr>
<td></td>
<td>Ward and Bennet’s approach</td>
<td>Source Code</td>
<td>[WB93]</td>
</tr>
<tr>
<td></td>
<td>Language Independent Approach</td>
<td>Source Code</td>
<td>[ZHIP14]</td>
</tr>
<tr>
<td></td>
<td>Component-oriented Architecture</td>
<td>Source Code</td>
<td>[ASSF11]</td>
</tr>
<tr>
<td></td>
<td>Semi-Automatic Approach for Extraction</td>
<td>Source Code</td>
<td>[VBP12]</td>
</tr>
<tr>
<td></td>
<td>SNIAFL</td>
<td>Source Code</td>
<td>[ZZL06]</td>
</tr>
<tr>
<td></td>
<td>RECAST</td>
<td>Source Code</td>
<td>[EM93]</td>
</tr>
<tr>
<td></td>
<td>Static and Dynamic Analysis</td>
<td>Source Code</td>
<td>[RHLR08]</td>
</tr>
<tr>
<td></td>
<td>Cerberus</td>
<td>Source Code</td>
<td>[EAA08]</td>
</tr>
<tr>
<td></td>
<td>Source Code Retrieval</td>
<td>Source Code</td>
<td>[LKE08]</td>
</tr>
<tr>
<td></td>
<td>Focused views on Execution Traces</td>
<td>Source Code</td>
<td>[BVD08]</td>
</tr>
<tr>
<td></td>
<td>Heuristic-Based Approach</td>
<td>Source Code</td>
<td>[ADPG10]</td>
</tr>
<tr>
<td></td>
<td>Dependence Graph</td>
<td>Source Code</td>
<td>[CR10]</td>
</tr>
<tr>
<td></td>
<td>Software Evolution Analysis</td>
<td>Source Code</td>
<td>[GDL05]</td>
</tr>
<tr>
<td></td>
<td>Concern Graphs</td>
<td>Source Code</td>
<td>[RM02]</td>
</tr>
<tr>
<td></td>
<td>Concept Analysis</td>
<td>Source Code</td>
<td>[EKSO1]</td>
</tr>
<tr>
<td></td>
<td>JAVACompExt</td>
<td>Source Code</td>
<td>[ARA09]</td>
</tr>
<tr>
<td></td>
<td>Antoun et al.</td>
<td>Source Code</td>
<td>[AAAC07]</td>
</tr>
<tr>
<td></td>
<td>Chouambe et al.</td>
<td>Source Code</td>
<td>[CKK08]</td>
</tr>
<tr>
<td></td>
<td>Design Components</td>
<td>Source Code</td>
<td>[KSRP99]</td>
</tr>
<tr>
<td></td>
<td>Subsystem Structure Identification</td>
<td>Source Code</td>
<td>[MOTU93]</td>
</tr>
<tr>
<td></td>
<td>Archimetrix</td>
<td>Source Code</td>
<td>[DPB14]</td>
</tr>
<tr>
<td></td>
<td>Memory-constrained environments</td>
<td>Source Code</td>
<td>[WF14]</td>
</tr>
<tr>
<td></td>
<td>Quality centric approach</td>
<td>Source Code</td>
<td>[KSCC12]</td>
</tr>
<tr>
<td></td>
<td>L2CBD</td>
<td>Source Code</td>
<td>[KC06]</td>
</tr>
<tr>
<td></td>
<td>CORE</td>
<td>Source Code</td>
<td>[MKM09]</td>
</tr>
<tr>
<td></td>
<td>Favre et al.</td>
<td>Source Code</td>
<td>[FDE01]</td>
</tr>
<tr>
<td></td>
<td>Bunch Tool</td>
<td>Source Code</td>
<td>[MS06]</td>
</tr>
<tr>
<td></td>
<td>Erdemir et al.</td>
<td>Source Code</td>
<td>[ETB11]</td>
</tr>
<tr>
<td></td>
<td>Systematic method approach</td>
<td>Source Code</td>
<td>[KC04]</td>
</tr>
<tr>
<td></td>
<td>Shatnawi et al.</td>
<td>Source Code</td>
<td>[SSSA17]</td>
</tr>
<tr>
<td></td>
<td>Alshara et al.</td>
<td>Source Code</td>
<td>[AST15]</td>
</tr>
</tbody>
</table>

**Source Code = 37**

<table>
<thead>
<tr>
<th>Feature Set = 7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Description = 9**

<table>
<thead>
<tr>
<th>Description-Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cerberus</td>
</tr>
<tr>
<td>Landmarks and Barriers</td>
</tr>
<tr>
<td>Ward and Bennet’s approach</td>
</tr>
<tr>
<td>Locating Features in Source Code</td>
</tr>
<tr>
<td>Trace Dependency Analysis</td>
</tr>
<tr>
<td>SNIAFL</td>
</tr>
<tr>
<td>Combining FCA with IR</td>
</tr>
<tr>
<td>Automatic Generation</td>
</tr>
<tr>
<td>Reverse Engineering Feature Models</td>
</tr>
</tbody>
</table>

Table 2.2: Reverse Engineering: Classification Based on Required Input
based on dynamic analysis require feature-set and most of the textual approaches require description based input. Table 2.2 shows the classification of reverse engineering approaches based on the required input. This classification is useful to compare the compatibility of each approach with available input notation.

2.4 Classification based on Retrieval

Based on the notation of their output, the reverse engineering approaches can be classified into following categories\(^2\).

2.4.1 Non-Architectural

Approaches belong to this category produce feature models, re-structured code or view-based output. View-based output is further classified into concept lattices or graphical notations (including documentation etc.) and code to feature trace or rank based trace to identify concept locations.

2.4.2 Non-Explicit Components

All approaches under this category produce some notation of architecture or non-explicit components. This category has those component directed reverse engineering approaches that do not follow the strict definition of component by Szyperski [Szy02]. These approaches neither extract explicit interfaces nor define the composition of components with respect to some component model. As explained in Section 2.2.4.1, CORE [MKM09], Bunch Tool [MS06] and Systematic method approach [KC04] are some of the component directed approaches that reverse engineer non-explicit components. Output of all these approaches is a set of clustered methods or classes, based on the parameters of coupling and cohesion via methods' interactions and invocations. Such clusters can help in confining the implementation of a feature but cannot exploit the architectural re-usability due to non-explicit nature of the generated output.

\(^2\)Table 2.3 Legend:

- F = Feature Model
- C = Code
- NEC = Non-Explicit Components
- EC = Explicit Components
- VB = View Based
- CL/G = Concept-Lattices/Graphs
- M/CT = Rank-Based Mapping/Code Trace
### 2.4. CLASSIFICATION BASED ON RETRIEVAL

<table>
<thead>
<tr>
<th>Approach</th>
<th>Generated Output</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>C</td>
</tr>
<tr>
<td>Evolutionary Algorithms</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Software Configurations using FCA</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Reverse Engineering Feature Models</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Product Variants</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Language Independent Approach</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Ward and Bennet's approach</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CORE</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Favre et al.</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Bunch Tool</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Systematic method approach</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Erdemir et al.</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Shatnawi et al.</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>JAVACompExt</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Antoun et al.</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Alshara et al.</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Chouambe et al.</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Component-oriented Architecture</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Design Components</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Subsystem Structure Identification</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Archimetrix</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Quality centric approach</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Memory-constrained environments</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>L2CBD</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Combining FCA with IR</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Landmarks and Barriers</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>RECAST</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Call-Graphs</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Burd and Munro's approach</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Concern Identification</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Dependence Graph</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Concern Graphs</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Software Evolution Analysis</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>RECoVar</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Focused views on Execution Traces</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Featureous</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Cerberus</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SNIAFL</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Source Code Retrieval</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Scenario Driven Dynamic Analysis</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>STRADA</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Natural language Parsing</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Trace Dependency Analysis</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Concept Analysis</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Dynamic Feature Traces</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Static and Dynamic Analysis</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Locating Features in Source Code</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Heuristic-Based Approach</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Semi-Automatic Approach for Extraction</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Focused views on Execution Traces</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Featureous</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Automatic Generation</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.3: Reverse Engineering: Classification Based on Generated Output
2.4.3 Explicit Components

These approaches follow the strict definition of component i.e. unit of composition with contractually specified interfaces and explicit context dependencies. The proposed research also belongs to this category of reverse engineering. Inexplicit components are neither easy to reuse nor they can follow the fixed semantics of composition for reconstructing the legacy code bases. Table 2.3 shows the overall classification of reverse engineering based on the generated output of approaches.

2.5 Classification based on Implementation Language

Implementation language shows the language on which an approach can be applied. Approaches that generate feature models and require description based documents as input are language independent, e.g., Evolutionary Algorithms [LHGB+12] and Software Configurations using FCA [AMHS+14]. Few approaches like RECoVar [ZB13] and L2BCD [KC06] are methodologies and hence they can be implemented in any language. Approaches like Focused views on Execution Traces [BVD08], Featureous [OJ10] and Call Graph [BD06] are language dependent as their tools are dependent on the programming language they have designed for.

In case of component directed approaches; Systematic method approach [KC04], Archimetrix [DPB14], Bunch Tool [MS06] and L2CBD [KC06] are language independent. Few approaches like CORE [MKM09] are not dependent on any specific language but they only work for object-oriented languages. Approaches like Chouambe et al. [CKK08], Antoun et al. [AAAC07], Alshara et al. [AST+15] and JAVACompExt [ARA+09] are based on Java. Like many component directed approaches, our proposed approach is also based on object-oriented code. For implementation, we chose Java therefore, it can be placed among the Java based approaches. RECAST [EM93] and Burd and Munro's approach [BM98] work with COBOL source code. All the approaches reverse engineer high level programming languages except the Ward and Bennet's approach that aims for the assembler code. Table 2.4 shows the implementation language of each approach.

Based on all the discussed parameters, Figure 2.1 shows the framework of classification for reverse engineering approaches. Details about required expertise and prerequisite requirements to use an approach are classified in Appendix A\(^3\).

\(^3\)This classification is not part of this chapter because it is not applicable on component directed reverse engineering approaches.
## 2.5. Classification Based on Implementation Language

<table>
<thead>
<tr>
<th>No. of Approaches</th>
<th>Approach Name</th>
<th>Implementation Language</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language Independent = 7</td>
<td>RECoVar</td>
<td>Language Independent</td>
<td>[ZB13]</td>
</tr>
<tr>
<td></td>
<td>Evolutionary Algorithms</td>
<td>Language Independent</td>
<td>[LHGB+12]</td>
</tr>
<tr>
<td></td>
<td>Systematic method approach</td>
<td>Language Independent</td>
<td>[KC04]</td>
</tr>
<tr>
<td></td>
<td>Ward and Bennet's approach</td>
<td>Language Independent</td>
<td>[WB93]</td>
</tr>
<tr>
<td></td>
<td>Archimetry</td>
<td>Language Independent</td>
<td>[DPB14]</td>
</tr>
<tr>
<td></td>
<td>Bunch Tool</td>
<td>Language Independent</td>
<td>[MS08]</td>
</tr>
<tr>
<td></td>
<td>L2CBD</td>
<td>Language Independent</td>
<td>[KC06]</td>
</tr>
<tr>
<td>Object Oriented = 8</td>
<td>CORE</td>
<td>Object Oriented</td>
<td>[MKM09]</td>
</tr>
<tr>
<td></td>
<td>Locating Features in Source Code</td>
<td>Object Oriented</td>
<td>[EKS03]</td>
</tr>
<tr>
<td></td>
<td>Quality centric approach</td>
<td>Object Oriented</td>
<td>[KSCC12]</td>
</tr>
<tr>
<td></td>
<td>Memory-constrained environments</td>
<td>Object Oriented</td>
<td>[WF14]</td>
</tr>
<tr>
<td></td>
<td>Favre et al.</td>
<td>Object Oriented</td>
<td>[FDE+01]</td>
</tr>
<tr>
<td></td>
<td>Erdemir et al.</td>
<td>Object Oriented</td>
<td>[ETB11]</td>
</tr>
<tr>
<td></td>
<td>Shaiwali et al.</td>
<td>Object Oriented</td>
<td>[SSSA17]</td>
</tr>
<tr>
<td></td>
<td>Software Configurations using FCA</td>
<td>Object Oriented</td>
<td>[AMHS+14]</td>
</tr>
<tr>
<td>Java = 19</td>
<td>Language Independent Approach</td>
<td>Java</td>
<td>[ZHP+14]</td>
</tr>
<tr>
<td></td>
<td>Natural language Parsing</td>
<td>Java</td>
<td>[AT10]</td>
</tr>
<tr>
<td></td>
<td>Semi-Automatic Approach for Extraction</td>
<td>Java</td>
<td>[VRP12]</td>
</tr>
<tr>
<td></td>
<td>Dynamic Feature Traces</td>
<td>Java</td>
<td>[EDV05]</td>
</tr>
<tr>
<td></td>
<td>Static and Dynamic Analysis</td>
<td>Java</td>
<td>[RHLR08]</td>
</tr>
<tr>
<td></td>
<td>Cerberus</td>
<td>Java</td>
<td>[EAA008]</td>
</tr>
<tr>
<td></td>
<td>STRADA</td>
<td>Java</td>
<td>[EBG07]</td>
</tr>
<tr>
<td></td>
<td>Source Code Retrieval</td>
<td>Java</td>
<td>[LKJ08]</td>
</tr>
<tr>
<td></td>
<td>Combining FCA with IR</td>
<td>Java</td>
<td>[PM07]</td>
</tr>
<tr>
<td></td>
<td>Heuristic-Based Approach</td>
<td>Java</td>
<td>[ADPG10]</td>
</tr>
<tr>
<td></td>
<td>Concern Identification</td>
<td>Java</td>
<td>[TR99]</td>
</tr>
<tr>
<td></td>
<td>Featureous</td>
<td>Java</td>
<td>[OF10]</td>
</tr>
<tr>
<td></td>
<td>Landmarks and Barriers</td>
<td>Java</td>
<td>[WRW07]</td>
</tr>
<tr>
<td></td>
<td>Chouambe et al.</td>
<td>Java</td>
<td>[CKK08]</td>
</tr>
<tr>
<td></td>
<td>Antoun et al.</td>
<td>Java</td>
<td>[AAC07]</td>
</tr>
<tr>
<td></td>
<td>Alshara et al.</td>
<td>Java</td>
<td>[AS+15]</td>
</tr>
<tr>
<td></td>
<td>Component-oriented Architecture</td>
<td>Java</td>
<td>[ASSF11]</td>
</tr>
<tr>
<td></td>
<td>Alshara et al.</td>
<td>Java</td>
<td>[AS+15]</td>
</tr>
<tr>
<td></td>
<td>JAVACompExt</td>
<td>Java</td>
<td>[ARA+09]</td>
</tr>
<tr>
<td>C = 7</td>
<td>Concept Analysis</td>
<td>C</td>
<td>[EKS01]</td>
</tr>
<tr>
<td></td>
<td>Call-Graphs</td>
<td>C</td>
<td>[BD06]</td>
</tr>
<tr>
<td></td>
<td>Product Variants</td>
<td>C</td>
<td>[XXJ12]</td>
</tr>
<tr>
<td></td>
<td>SMIAPL</td>
<td>C</td>
<td>[ZZL+16]</td>
</tr>
<tr>
<td></td>
<td>Focused views on Execution Traces</td>
<td>C</td>
<td>[BVD08]</td>
</tr>
<tr>
<td></td>
<td>Dependence Graph</td>
<td>C</td>
<td>[CR10]</td>
</tr>
<tr>
<td></td>
<td>Subsystem Structure Identification</td>
<td>C</td>
<td>[MOTU93]</td>
</tr>
<tr>
<td>COBOL = 2</td>
<td>RECAST</td>
<td>COBOL</td>
<td>[EM93]</td>
</tr>
<tr>
<td></td>
<td>Burd and Munro's approach</td>
<td>COBOL</td>
<td>[BM98]</td>
</tr>
<tr>
<td>C++ = 7</td>
<td>Source Code Retrieval</td>
<td>C++</td>
<td>[LKE08]</td>
</tr>
<tr>
<td></td>
<td>Focused views on Execution Traces</td>
<td>C++</td>
<td>[BVD08]</td>
</tr>
<tr>
<td></td>
<td>Scenario Driven Dynamic Analysis</td>
<td>C++</td>
<td>[SMADP06]</td>
</tr>
<tr>
<td></td>
<td>Call-Graphs</td>
<td>C++</td>
<td>[BD06]</td>
</tr>
<tr>
<td></td>
<td>Trace Dependency Analysis</td>
<td>C++</td>
<td>[FEG93]</td>
</tr>
<tr>
<td></td>
<td>Design Components</td>
<td>C++</td>
<td>[KSRP99]</td>
</tr>
<tr>
<td></td>
<td>Reverse Engineering Feature Models</td>
<td>C++</td>
<td>[SLB+11]</td>
</tr>
<tr>
<td>Assembler Code = 1</td>
<td>Ward and Bennet's Approach</td>
<td>Assembler Code</td>
<td>[WB93]</td>
</tr>
</tbody>
</table>

Table 2.4: Reverse Engineering: Classification based on Implementation Language
Classification of Reverse Engineering

Analysis Technique
- Static
- Textual
- Dynamic
- Hybrid
- Code

Input Notation
- Feature Set
- Description Based
- Code

Output Notation
- Non-Explicit Components
- Explicit Components
- View Based
- Concept Lattices/Graphs
- Feature to Code Trace

Implementation Language
- COBOL
- Java
- C/C++
- Language Independent
- Object Oriented
- Assembler Code

Figure 2.1: Reverse Engineering: Framework of Classification
2.6 Summary

Non-textual static reverse engineering is the most suitable analysis technique for component directed reverse engineering. It can capture the whole code base, can be reused after modifications in the legacy code, does not need execution of the system, does not include the complex process of code instrumentation for every feature, does not depend on just code identifiers and does not lose the code functionality due to implicit implementation of features.

Component directed reverse engineering is used to retrieve architectural notation from legacy systems. Most of the approaches in this domain do not define the composition or required and provided services of the extracted components explicitly. For such approaches, a component is a set of clustered methods that implement a common feature. Lack of explicit services, well-defined composition and repository are hurdles in the way of proper re-usability.

A reverse engineering approach can be classified based on its analysis technique, the input notation it requires, the output notation it generates and the programming languages it can be applied on. Such a classification can help in defining the limitations and pros and cons of a proposed approach. From re-usability point of view, extraction of explicitly defined components is more reusable as compared to clustered methods or classes.
Chapter 3

Component Based Development

“Once you break a habit into its components, you can fiddle with gears.”
— Charles Duhigg, New York Times

3.1 Introduction

Component based development (CBD) or component based software engineering (CBSE) is a domain that revolves around the construction of systems from pre-built software units. A component is defined as "A unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition." [Szy02]. Components evolved as a solution of problems like scalability, unreliability and re-usability in software development. CBD gave the concept of product centric development rather than process centric [LLE07].

This chapter includes the basic concepts of CBD, architectural properties of component models, classification of component models, a taxonomy of component models from re-usability point of view and detailed explanation of the component model selected for reverse engineering the legacy code. All component models are not created equal and the semantics of component models play an important role in imposing the limitations of current component directed reverse engineering approaches. Section 3.2 of this chapter explains the component models and their classification. Section 3.3.1
3.1. INTRODUCTION

describes the architectural properties of a component model that are important for re-
verse engineering. Section 3.3 includes the life cycle of a component and a taxonomy
of component models (from re-usability point of view) based on this life cycle. This
chapter also includes (Section 3.4.3) the detailed explanation and semantics of the se-
lected component model for this research.
3.2 Component Models

A component can be described as follows:

- "A software element that (a) encapsulates a reusable implementation of functionality. (b) can be composed without modification. (c) adheres to a component model" [LVDH].
- Deployable piece of software without persistent state [Szy02].
- "A software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard" [HC01].

The basic idea behind all these well-known definitions is the same i.e., a component is an independent deployable piece of software that should be reusable. But, any reusable part of a code cannot be qualified as a component. The code needs to be compliant with a component model. "A component model is a combination of (a) a component standard that governs how to construct individual components and (b) a composition standard that governs how to organize a set of components into an application and how those components globally communicate and interact with each other." [LVDH].

In other words, every component model needs to define two main entities:

1. **Semantics of a component (unit of composition)** i.e., the notation of a piece of code that defines boundary, granularity and representation of a component with respect to that component model.
2. **Composition mechanism** i.e., how and in what ways two or more than two unit of compositions (components) can be composed together to form an application, executable system or a composite component.

3.3 Component Life Cycle

An idealised component life cycle has following phases [LLE07].

1. Design phase
2. Deployment phase
3. Run phase

Depending on the component model, composition is possible in design and deployment phases. In design phase of a component, components are constructed and deposited into a repository by a builder tool (e.g., an IDE) for later retrieval. In deployment phase, ideally the deposited components from design phase should be retrieved
3.3. COMPONENT LIFE CYCLE

Figure 3.1: An Idealised Component Life Cycle ([LdC17a]).

and tailored for deployment. Composition in deployment phase is conducted by an assembler. The assembler is responsible for retrieving, composing, compiling and assembling a system during this phase. In run-time phase, there is no composition but only instantiation of the components. Figure 3.1 is showing the idealised component life cycle presented by [LW05, LW07].

3.3.1 Architectural Properties of Components

Following the life cycle of section 3.3, there are two architectural properties of component models that define the degree of re-usability of component directed reverse engineering i.e., deposit-ability and re-composability [AL18]. Deposit-ability requires a repository for depositing components and retrieving them, whilst re-composability requires a component model [LW07, LdC17a] with well-defined composition mechanisms for components.

Few component models do not support the repository e.g., ArchJAVA [ACN02] and UML2.0 [BO06]. Few have the repository but contents of the repository cannot
be retrieved after deposition e.g., EJB [EK01] and FRACTAL [BCL+06]. Few can do composition in deployment phase but cannot offer any composition in the design phase e.g., JavaBeans. Few can compose in design phase but cannot offer any composition in the deployment phase e.g., Koala [ASM04]. The only component model that offers composition in both design and deployment phases is X-MAN component model. X-MAN component model provides better deposit-ability and re-composability because it defines proper composition mechanism due to separation of control and computation.

An X-MAN component can be: composed and deposited in design phase, retrieved in deployment phase, composed in deployment phase, re-deposited after composition and retrieved in future for further reuse or re-composition. The re-usability that X-MAN offers is code-independent as one does not need to re-configure ports at code level but can compose new instances of tailored components by using appropriate composition connectors and exposed methods. A detailed analysis of X-MAN component model will be presented in Section 3.4.4.

3.3.2 A taxonomy of Component Models

Based on the idealised component life cycle, [LW05] presented a taxonomy of component models. The presented taxonomy is based on the same parameters that are identified to be responsible for re-usability of the extracted components i.e., deposit-ability and re-composability. According to this taxonomy, component models are classified, based on the attributes like repository, design phase composition and deployment phase composition. This taxonomy considers desiderata of component based development [Lau14] and idealised component life cycle proposed by [LdC17a]. This taxonomy can be used as a reference for component directed reverse engineering because re-usability of reverse engineered components is dependent on the same parameters i.e., repository (deposit-ability) and ability to do re-composition (composition of re-deposited components in deployment phase). According to this taxonomy, component models are classified into following categories:

- **Design Without Repository:** Component models belong to this category can only compose in design phase. There is no repository or assembler for deposition or re-composition. Therefore, component models of this category are not good from re-usability point of view. Few component models belong to this category are POJOs, Acme [GMW00, GMW95], ArchJava [ACN02] and UML 2.0 [BO06].
### 3.3. COMPONENT LIFE CYCLE

<table>
<thead>
<tr>
<th>Category</th>
<th>Component Models</th>
<th>Design</th>
<th>Deploy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Without Repository</td>
<td>POJOs, Acme, UML2, ArchJava</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Design With Deposit-only Repository</td>
<td>EJB, OSGi, CCM, FRACTAL</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Deployment With Repository</td>
<td>JavaBeans, Web Services</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Design With Repository</td>
<td>Koala, SOFA, Palladio</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Design and Deployment With Repository</td>
<td>X-MAN</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deposit</th>
<th>Retrieve</th>
<th>Compose</th>
<th>Deposit</th>
<th>Compose</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 3.1: A Taxonomy of Component Models ([LW05])

- **Design With Deposit-only Repository**: Models of this category do not have an assembler therefore, composition is only possible in the design phase. Unlike previous category, these models have a repository but it is deposit-only i.e., you cannot re-compose the deposited components. Component models belonging to this category are also not good from re-usability point of view as they do not allow re-composition of the deposited components. Few component models from this category are EJB [CI01], OSGi, CCM [LDC17b] and FRACTAL [BCS04].

- **Deployment With Repository**: As the name suggests, this category has a repository as well as an assembler. Repository is not deposit-only but composition is only possible in deployment phase. JavaBeans [CI01] and Web Services belong to this category of classification. Inability to compose in design phase makes this category less reusable.

- **Design With Repository**: In this category, composition is only possible in design phase. This category has a repository but there is no assembler to do composition in deployment phase. Component models belonging to this category includes SOFA [BHP06], Koala [ASM04] and Palladio [BKR09].

- **Design and Deployment With Repository**: This category can compose in both design and deployment phases. There is an assembler and a repository (not deposit-only). Only component model belongs to this category is X-MAN. Ability of composition in both design and deployment phases is due to exogenous
CHAPTER 3. COMPONENT BASED DEVELOPMENT

composition. This category can provide maximum re-usability as one can re-compose and re-deposit the components for further reuse. This is also the only category that provides code-independent re-usability without restricting to use all the methods of a component. The X-MAN component model will be explained in detail in Section 3.4.4.

The overall classification of component models based on this taxonomy is presented in Table 3.1. In the table, Deposit-N means ability to deposit new components into repository, Retrieve means ability to retrieve the components from repository, Compose stands for ability to compose, Deposit-C means ability to deposit composite components and Compose under deploy means ability to compose in deployment phase.

3.4 Classification Based on Composition Mechanism

Based on the unit of composition and composition mechanism, component models can be classified into three main categories [LdC17a].

1. Models based on objects
2. Models based on architecture units
3. Models based on encapsulation

3.4.1 Component Models based on Objects

Component models belong to this category have objects as unit of composition and their composition mechanism is method delegation (direct method calls). In this category, each component is an object which does not show its required services in its interface. The most general example of such component models is POJOs (plain old java objects). In POJOs, a component is a Java object and objects are composed by method calls.

Enterprise Java Beans (EJB) [CI01, Gro06] is another component model based on objects. In EJB, a component is a Java object in EJB container inside a J2EE server. Remote interfaces are used for method calls.

Another well-known component model based on objects is JavaBeans [CI01]. In JavaBeans, each component is a bean. A bean is a Java class that has methods and events synchronised by a container (BeanBox). The container is responsible for managing and synchronising events delegation and method calls between all the beans. In other words, this container composes all the beans together by delegation of events.
In CCM [LDC17b], each component is a meta-type of CORBA (component model) [OMG02]. Components exist in a CCM container. CCM composes the components by event and method delegation using four types of ports: Facets (provided services) to match with receptacles (required services) and event sources to match with event sinks.

In OSGi (Open Services Gateway Initiative), each component is a POJO inside a bundle (consists of a JAR file, classes and a manifest file). Manifest file has all the information about the JAR file (dependencies, exports etc.). Two bundles cannot be composed, but only POJOs within them are composed by direct method calls. Therefore, an OSGi bundle is not a component itself. Beanome [CH02] is a component model based on OSGi framework.

Web services also belong to this category of components. In web services, orchestration or invocation of methods in a SOAP (Simple Object Access Protocol) [MPD02], REST (Representational State Transfer) or JSON (JavaScript Object Notation) [Ihr13] is used as a composition mechanism.

In these component models, order of the execution of methods (control structure of composition mechanism) and data computation (execution of methods) both exist at the code level and cannot be separated from each other.

3.4.2 Component Models based on Architectural Units

These models are based on architecture description languages (ADLs). In these component models, unit of composition is an architecture unit and ports are used as composition mechanism. An architecture unit consists of implementation and required and provided ports (services that are invoked by indirect method calls). Most of the component models belong to this category of classification and almost all the component directed reverse engineering approaches use component models based on architecture units.

The most well-known component model based on ADLs is UML 2.0 [BO06]. In UML 2.0, each component has required and provided services as ports. Components are always composed inside a higher level component that is a composite component (hierarchical composition). Composition between required and provided ports is achieved via assembly connectors (within a composite component) and external composition (with higher level interfaces) uses delegation connectors.

In Koala component model [ASM04], each component is a unit of design and implementation. Each component interacts with others via interfaces. Each component
implements a function and calls other components for their defined functions. Figure 3.2 is showing the notation of a Koala component. Direction of arrows in the figure shows whether a function is required (tip pointing inside) or provided (tip pointing outside). Interfaces and components are stored in global repositories using C language. Every required function in Koala has to be connected with at least one provided function and composition takes place by connecting interfaces. A component is composed with others via interface-less component called module.

Acme [GMW00, GMW95] is a meta-model for ADLs. Interfaces in Acme can be invoked in specific orders based on procedural calls. Components compose in Acme by a connector that conducts co-ordination and communication between them.

ArchJava component model [ACN02] is based on Acme and it uses an extension of Java to define component class and ports at code level. Every component in ArchJava is an instance of the component class. All required and provided ports have to be connected at code level to execute the component’s implementation.

In SOFA component model [BHP06], every component is an architectural unit defined by its frame and its implemented architecture. SOFA has multiple communication style options e.g., SOFA components can communicate via remote calls, indirect message passing, streaming or shared memory. Composition takes place via ports defined by the frame.

FRACTAL [BCS04] components are like UML 2.0 components but with addition of membrane. The membrane supports interfaces for internal re-configuration (re-configuration of sub-components). Composition takes place via required and provided interface along with the control interfaces (interfaces of the membrane).

Palladio [BKR09] is a meta-model based on ADL that is designed to enable quality parameters i.e., performance and reliability in component based architectures defined
3.4. **CLASSIFICATION BASED ON COMPOSITION MECHANISM**

by ADLs. Like UML 2.0, composition takes place via assembly and delegation connectors.

Similar to component models based on objects, these component models also have data computation and control structures of composition mechanism at code level and these two cannot be separated. For both architectural units and objects, composition is defined *endogenously*, i.e. inside components, by indirect method calls (via ports) and direct method calls respectively. Endogenous composition hinders code-independent re-usability since it requires manual re-coding of components (re-wiring of ports).

3.4.3 **Component Models based on Encapsulation**

Encapsulated components are like ADLs but they do not have required services i.e., every encapsulated component only have provided methods. Unlike ADLs and objects based models, composition in encapsulated components is *exogenous*, i.e., defined outside of the components. In other words, control and computation are separated in exogenous composition unlike ADLs where both are transmitted together via ports.

The notations of the components from all three categories are presented in Figure 3.3\(^1\). The only component model based on encapsulation is X-MAN\(^2\). The following section will explain the working of X-MAN component model and its exogenous composition that separates control form computation.

---

\(^1\)Component models based on objects have dotted required services in the figure because they do not show required services. 

\(^2\)An in-depth discussion of component models can be found in [LW07, LdC17a].
3.4.4 X-MAN Component Model

3.4.4.1 Basic Semantics of X-MAN

Every atomic X-MAN component consists of a computation unit that has the implementation of all the methods in it. Computation units are encapsulated as no computation can go outside of it. In other words, two computation units of two components cannot interact with each other via required and provided ports like ADLs. Two or more than two computation units can only interact by composition that makes them a composite component. Such interaction is based on special types of connectors that are called composition connectors. These connectors are responsible for the flow of control and order of execution of methods in computation units of a composite component that is composed by these connectors.

Each atomic X-MAN component has a compositional service which is intractable with composition connectors. It means, composition of two atomic components involves the composition of two services whereas, each service represents the definition of methods that are implemented in each computation unit. Hence, services in X-MAN are compositional i.e., service $S_1$ and $S_2$ of components $C_1$ and $C_2$ compose to form a service $S_3$ which is composed of integration of $S_1$ and $S_2$. It means composition of $C_1$ and $C_2$ involves formation of a new service $S_3$ that provides the integration of services of those components.

In case of object oriented code, two method calls cannot integrate to return one call i.e., method calls are not compositional like services of X-MAN. Therefore, the entity service of X-MAN component model cannot be extracted from a legacy code as no such compositional integration is valid from source code point of view. In order to resolve this shortcoming, this research has presented a modified version of X-MAN that does not involve any service. Instead of services, the presented version uses exposed methods that can interact with composition connectors. The following section will explain the details of our X-MAN model that will be used to reverse engineer components.

3.4.4.2 X-MAN for Reverse Engineering

In our presented version, an atomic X-MAN component consists of a computation unit that has the implementation of methods and exposed functionality of selected methods (the methods that can be selected for composition at deployment phase). Methods are exposed as interfaces which are implemented in the computation unit. Any method
of a composition unit can be selected to be exposed before instantiating a component and method's inputs and outputs can be used with the exposed methods of other X-MAN components. Computation only takes place in a computation unit, which is why this component model is encapsulated [LDC17b]. There are two types of connectors in X-MAN: a) adapter connectors. b) composition connectors. Adapter connectors include Guard and Loop. The Guard is used for conditional invocation of an atomic component. Guard can take a conditional statement as an input and only invokes the component if the condition is validated. Figure 3.4 (a) is showing an atomic component with Guard. The Loop is used for conditional looping of an atomic component. It takes an iterator with an initial value, looping interval (increment by x or decrement by x if x is an integer) and a condition of termination. Figure 3.4 (b) is showing an atomic component with Loop connector.

In case of a composite component, encapsulation is preserved by composition because a composite component consists of two or more atomic components composed together by composition connectors. Composition connectors are used to compose 2-n atomic components. All such atomic components can only provide exposed methods and co-ordinate with one another using composition connectors. Composition connectors in X-MAN are actually control structures that direct the route of execution. Sequencer (SEQ) composition connector provides sequencing of execution between two or more than two components and Selector (SEL) provides branching based on specific conditions\(^3\). If two components A and B will be instantiated with one exposed method each and composed by a sequencer, then there will be only two methods that

\(^3\)With SEQ (sequencing), SEL (branching) and LOOP (looping), X-MAN is Turing complete.
will be involved in this composition. Basic semantics of the defined X-MAN component model are shown in Figure 3.5 (lollipop in the Figure is showing the presence of exposed methods). One computation unit cannot interact with other units directly but only via composition connectors. Control of the components exists outside of computation units and that is why one does not need code-level configurations to reuse the components. Any component can be reused by composing it with others using appropriate composition connector [LDC17b] and any exposed method can be selected that is needed for a specific composition of components.

Figure 3.6 is showing the X-MAN meta-model relevant to semantics of an atomic component and Figure 3.7 is showing an example of composition between component
A and B. In Figure 3.6, each component can either be atomic or composite. Each *Atomic Component* has one *Computation Unit* that has the implementation of *Methods*. At the time of deployment, methods can be selected to be exposed for composition (*E.Methods* in Figure 3.6) and order of the execution of selected exposed methods can be set by their reference value i.e., *E.MethodsRef*.

In Figure 3.7, component A has two exposed methods (m1, m2) and component B has three of them (m3, m4, m5). These components are composed as a composite component by sequencer (SEQ). The sequencer in the Figure will trigger the component A first as this route has been assigned the lower number (higher priority). The selection of exposed methods and their order of execution is selected at deployment phase while instantiating the components e.g., in component A either method m1 can be executed before m2 or vice versa. After completion of their execution the sequencer will trigger the component B (the order of execution of exposed methods in B can also be defined during instantiation of B). The entity *E.MethodsRef* in Figure 3.6 sets the reference to execute the exposed methods in defined order. There are \( n! \) ways to order the execution of selected exposed methods in an atomic X-MAN component whereas, \( n \) is the number of methods selected to be exposed in an atomic component before instantiation. Further details about all the entities in X-MAN meta-model are presented in Appendix B.

In objects-based and ADL-based component models, control cannot be separated from computation and therefore, one needs a code-level configuration to compose
method calls or required and provided services (control has to define the composition, and transmission of control and data together demands the code level changes for each different composition). In X-MAN, one does not need to bind all the methods of a computation unit but only needs to select the methods to be exposed for a composition. Table 3.2 sums up the classification of component models based on the composition units and composition mechanism.

<table>
<thead>
<tr>
<th>Component Models</th>
<th>Unit of Composition</th>
<th>Composition Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>POJOs, JavaBeans, EJB, Web Services, OSGi, CCM, Beanome</td>
<td>Object</td>
<td>Direct Method Calls</td>
</tr>
<tr>
<td>ArchJava, Koala, SOFA, FRACTAL, Acme, Palladio, UML 2.0.</td>
<td>ADL</td>
<td>Indirect Method Calls (Ports)</td>
</tr>
<tr>
<td>X-MAN</td>
<td>Encapsulated Unit</td>
<td>Connectors</td>
</tr>
</tbody>
</table>

Table 3.2: Component Models Based on the Composition Mechanism
3.5 Summary

All component models are not created equal. There are some architectural properties of component models that determine the degree of re-usability of the components. Component models based on objects and ports cannot separate control from computation as both transmit together. Inability to separate control from computation is the reason of code-dependent re-usability in component directed reverse engineering.

The maximum re-usability after reverse engineering can be obtained by using a component model that offers exogenous composition and ability of composition in both design and deployment phases. The only component model belongs to this category is X-MAN. X-MAN components offer composition by composition connectors that exist outside of the computation units. X-MAN components have the ability to achieve code-independent re-usability, re-deposit-ability and re-composition of re-deposited components.
Chapter 4

Overview of RX-MAN: A Novel Reverse Engineering Approach

“When you want to know how things really work, study them when they’re coming apart.”
— William Gibson, Zero History

4.1 Introduction

This chapter intends to provide the road map of building a component directed reverse engineering approach that can reuse the extracted components code independently. Background work in the domains of reverse engineering (Chapter 2) and component based development (Chapter 3) suggested that the most appropriate analysis technique to conduct component-directed reverse engineering is static analysis. The literature review also showed that the transmission of control and computation together is the core reason behind the code-dependent re-usability of component-directed reverse engineering and for code-independent re-usability, one needs a component model that defines exogenous composition and has the ability to compose in both design and deployment phases.

Component-directed reverse engineering consists of four major steps [AL18]: 1) capturing the code base to reverse engineer. 2) re-structure the captured code to a mapable notation by implementing a rule base. 3) a defined rule base to map the captured
code notation to a set of clusters. 4) mapping of clusters to the desired notation (component model).

There are multiple ways of implementing these four steps e.g., a code base can be captured by a parser [FDE+01], by conducting a formal context analysis [XXJ12] or by forming directed graphs using heuristics [ETB11]. A rule base can be built on many possible parameters e.g., methods' invocations, methods' interactions [AL18], interaction of data elements [BM98], cohesion threshold or heuristics etc. [MKM09, ETB11]. A cluster can be defined as a set of methods, set of UML diagrams or a set of classes with re-structured code. Mapping of clusters to the desired component model (if approach yields explicitly defined components) depends on the semantics of composition unit and rules of composition of that component model.

These four steps can be separated into two primary categories of implementation: 1) Theoretical construction of the approach by defining semantics and a rule base for reverse engineering. 2) Practical implementation that maps the theory to applicable parameters of construction e.g., how to implement the rule base? Which parser can be customised for capturing the code base? How semantics can be implemented as a set of algorithms? etc. Out of the above four steps of component directed reverse engineering, first two points are related to practical implementation whereas, the other two are related to theoretical construction.

Implementation is always based on a theoretical framework and therefore, it is important to explain the theoretical mapping from source code to components by presenting a rule base. Such semantics then can be translated to the details of implementation e.g., selection of a parser, design of a parser etc.

This chapter provides a road map of our approach (RX-MAN). The road map includes the theory and rule base of the proposed approach, implementation of the rule base, evaluation of the proposed approach and, discussion and limitations of the proposed approach.
4.2 Overview of RX-MAN

RX-MAN stands for reverse engineering X-MAN as our approach uses X-MAN component model to reverse engineer components from legacy code bases. Figure 4.1 is showing the overall roadmap of RX-MAN along with the related chapters to each step of component-directed reverse engineering.

Chapter 5 includes all the theoretical discussion that shows the equivalence between Java constructs and X-MAN constructs and a rule base that clusters methods based on their interactions and invocations. This chapter also includes a rule base to map branching (e.g., if-else, switch statements etc.) and looping (e.g., while, do-while, for loops etc.) constructs to X-MAN adapter and composition connectors.

Chapter 6 is all about implementation. It includes a proposed parser to capture and restructure a Java code base and a set of algorithms that implement the rule base and constructs equivalency between Java entities and X-MAN entities. In other words, this chapter is the realisation of semantics presented in the Chapter 5.

Chapter 7 discusses the technology stack and the features of implemented RX-MAN tool. The tool is based on the implementation details of Chapter 6.

Chapter 8 includes validation and evaluation of the proposed approach. Our evaluation includes empirical evaluation via 5 code bases, evaluation in terms of accuracy, precision and traceability of the proposed approach via Gannod and Cheng’s framework [GC99] and tool evaluation via Bellay and Gall’s framework [BG98]. This chapter also includes a comparison between RX-MAN and other approaches that reverse engineer explicitly defined components.

Chapter 9 includes conclusions, discussion in terms of research objectives, answers...
4.3. INTENDED RESEARCH CONTRIBUTIONS OF RX-MAN

RX-MAN stands for reverse engineering X-MAN. It is an approach aims at the extraction of X-MAN components from object oriented legacy code. The contributions of RX-MAN can be summed up as follows:

1. A customised version of X-MAN component model suitable for reverse engineering.
2. A reverse engineering approach that offers code-independent re-usability.
3. An eco-system of reverse engineering that offers design and deployment phase composition of extracted components.
4. Flexible re-usability of extracted components that does not require binding of all the ports like ADLs.
4.4 Summary

RX-MAN approach is based on an exogenous component model and static analysis. The proposed approach is based on a defined rule base that presents an equivalent transformation between source code constructs and constructs of X-MAN component model. The research will be validated by extracting and depositing valid X-MAN components into a repository and the approach will be evaluated by following means: a) statistical evaluation and possible cases of composition in the presented code bases. b) demonstration of code-independent re-usability. c) demonstration of re-deposition. d) Gannod and Cheng's framework [GC99] and BELLAY and Gall's capability assessment.
Chapter 5

Object-Oriented Code vs X-MAN

“Truth can only be found in one place: the code.”
— Robert C. Martin, Agile Manifesto

5.1 Introduction

This chapter includes two of the four steps of component directed reverse engineering as defined in Section 4.1. The steps that are part of this chapter are related to theoretical framework and semantics of the proposed rule base. This chapter established the proposed rule base and equivalency between Java constructs and X-MAN constructs whereas, next chapter will include the realisation of this rule base.

Section 5.2 of this chapter explains the basic principles that are used to form clusters of methods that belong together. Section 5.3 shows the mapping of control structures from these clusters to X-MAN entities. Section 5.4 shows an example of brake control system to demonstrate the mapping and a simple case of composition in the mapped X-MAN components.
CHAPTER 5. OBJECT-ORIENTED CODE VS X-MAN

5.2 Object-Oriented Code VS X-MAN

This chapter covers the semantics of mapping Java constructs to X-MAN constructs (we call it "component mapper") and the "rule base" i.e., semantics to reverse engineer the object-oriented code to X-MAN component model for deposition of the extracted components. Coloured boxes in the Figure 5.1 are showing the relevant entities of our research related to this chapter.

As explained in the Section 3.4.3, an atomic X-MAN component consists of a computation unit that has the implementation of methods and exposed functionality of specific methods for composition at deployment phase whereas, a composite component can compose multiple atomic X-MAN components via composition connectors. Hence, a source code has to be mapped to these entities in X-MAN meta-model (Appendix B). The input to the proposed approach is a code scenario and output would be (1-n) X-MAN components. The succeeding section will explain the rules to form clusters of methods that belong together. These rules are based on methods' invocations and modifiers. The notation of the captured code (on which rule base can be applied) and the clusters will be explained in the Chapter 6 (Code Mapper in Figure 5.1).

5.2.1 Mapping Rule Base and Scenarios of Interactions of Methods

Before defining any rules to cluster the methods together, an important thing to define is the granularity level of a cluster. There needs to be a defined boundary that can decide the maximum number of clusters, extractable from a code base. For this chapter and for explaining the rules with multiple scenarios, we assume that this granularity

![Figure 5.1: RX-MAN: Mapping from Object-Oriented Code to X-MAN.](image-url)
Figure 5.2: Scenario 1

level is a Java class e.g., a code base with n number of classes can extract clusters, ranging from 1 to n along with the utility clusters. Every cluster can have its utility cluster which contains all methods of a cluster that do not call any other method i.e., utility methods. These utility clusters specifically useful if the source code is a library i.e., such utility methods can be used by newly created components to use these utility methods. The exact number of clusters from a code base are dependent on the invocations and modifiers of methods in that code base.

The interaction of methods in an object-oriented code depends on their invocations, and the constraints of invocations are defined by their modifiers (public, protected, private etc.). Considering this, the following scenarios explain the basic rules of clustering.

5.2.1.1 Single Class with no interaction

A single non-interactive class (smallest executable entity in object-oriented code) is the most trivial scenario in RX-MAN. In this scenario, all the methods that call each other belong to the same class. As the smallest defined granularity for a cluster is a Java class therefore, this scenario will yield one cluster with all the methods and their dependencies (imports, class variables etc.).

The next step is to map this cluster to X-MAN meta-model. To get map-able notation for exposed methods and computation unit of an atomic X-MAN component, the code in the cluster will be re-structured as an interface and its implementation e.g.,

```java
public class A {

    public int provideSpeed(int speed) {
        speed = 100;
        this{returnSpeed(speed);
        return speed;
    }

    public int returnSpeed(int topSpeed) {
        this.saveInLog(topSpeed);
        return topSpeed;
    }

    private void saveInLog(int value) {
        System.out.println("Value is saved");
    }
}
```
in the scenario of Figure 5.2, the code will be re-structured as Figure 5.3. Implementation of all the methods of the cluster will be mapped to computation unit of an X-MAN component whereas, any method can be selected to be exposed for a composition in deployment phase by mapping the definition methods (interface) as exposed methods. In Figure 5.2, methods $provideSpeed()$, $returnSpeed()$ and $saveInLog()$ will be mapped to an interface and their implementation will be mapped to a class that implements that interface.

Figure 5.5 is showing the mapping between the re-structured Java Code constructs and the constructs of X-MAN meta-model. In Figure 5.5, the definition of methods (interface) in the restructured code will be mapped to $Exposed\ Methods$ construct of
X-MAN meta-model. Parameters of methods will be mapped to \textit{Input} construct of meta-model and method's returns will be mapped to \textit{Output} construct. Implementation of methods will be mapped to \textit{Computation Unit} in X-MAN meta-model. Each method's implementation is also mapped to \textit{Method} construct to choose individual methods to be exposed for a composition. The Figure 5.5 is showing restructured code, notation of an atomic X-MAN component and the exposed notation of the component that shows select-able exposed methods. Further details about the semantics of X-MAN meta-model are presented in Figure B.5.

As the method \textit{saveInLog()} is a utility method, another cluster i.e., utility cluster will also be extracted as shown in the Figure 5.4. Figure 5.6 shows the notation of X-MAN components mapped from this scenario. Red rectangles in Figure 5.6 are showing the exposed functionality of these components i.e. exposed methods that can be selected for a composition.

5.2.1.2 Two Classes with public-public methods interaction

Next possible scenario is the interaction of two Java classes in a code base. According to the defined granularity of a cluster, the output can be one or two clusters. As our approach is based on interaction and invocation of methods, modifiers of methods play an important role in defining a scenario. Figure 5.7 is showing an example of two Java classes that interact with each other via methods with public modifiers.

In this scenario, output would be just one cluster. All the callers would be mapped to one computation unit along with the methods they called. If a method M is in invocation list of more than one methods, it would be placed in the computation unit only once to avoid redundancy. This scenario assumes that all the interactions are between
public methods and no method is neither invoking any private method nor dealing with any private class level variable. Figure 5.8 is showing the notation of re-structured cluster from scenario 2 and Figure 5.9 is showing the X-MAN component mapped from this cluster. The method call `obj.evaluateSpeed()` in method `provideSpeed()` will be changed to `this.evaluateSpeed()` in mapped notation of code, as both methods now belong to the same component. The creation of object of class B will also be omitted in mapped code notation. These changes in the mapped code indicate two important things: 1) if two methods belong to same component, they do not need to call one another via object any more therefore, method calls via objects will be replaced by simple calls. 2) Invocation of methods using objects is not required among X-MAN components as methods can only access each other via data channels of exposed methods. As the method `evaluateSpeed()` is a utility method, a utility cluster will also be extracted and its component is shown in the Figure 5.9.

### 5.2.1.3 Two Classes with private-public OR public-private methods interaction

This scenario has more possible outcomes than the previous two. If a private or a public method in Class A calls a public method in Class B, there are following possible scenarios.

- public method in Class B is neither accessing any private variable of the class nor it is calling any private method of B. In this case, such public method will be placed with its caller in the same cluster (along with any public class fields and imports it uses).
5.2. OBJECT-ORIENTED CODE VS X-MAN

```java
public class A {
    B obj = new B();

    public int provideSpeed(int speed) {
        speed = 100;
        obj.evaluateSpeed(speed);
        return speed;
    }

    public class B {
        public int evaluateSpeed(int topSpeed) {
            int maxSpeed = 200;
            return maxSpeed - topSpeed;
        }
    }
}
```

Figure 5.7: Scenario 2

```java
package com.AB.scenario;
public interface AB {
    public int provideSpeed(int speed);
    public int evaluateSpeed(int topSpeed);
}
```

```java
package com.AB.scenario;
public class ABImpl implements AB {
    public ABImpl() {
    }

    public int provideSpeed(int speed) {
        speed = 100;
        this.evaluateSpeed(speed);
        return speed;
    }

    public int evaluateSpeed(int topSpeed) {
        int maxSpeed = 200;
        return maxSpeed - topSpeed;
    }
}
```

Figure 5.8: Re-structured Code (Cluster) from Figure 5.7

- if public method in Class B uses a private variable of Class B or it calls some other private method of B, it cannot be placed with its caller (as defined granularity is class level). In this case, public method of B will only be part of the cluster of B along with the private methods it calls. The end result will be two clusters that will be mapped to two atomic X-MAN components.

Figure 5.10 shows a scenario of public-private case. Method `provideSpeed()` of Class A has method `evaluateSpeed()` of Class B in its invocation list. Method `evaluateSpeed()` is accessing private method `saveInLog()` of Class B. Output of this scenario will be two
public class A {
    B obj;

    public int provideSpeed(int speed) {
        speed = 100;
        obj = new B();
        obj.evaluateSpeed(speed);
        return speed;
    }

    public class B {
        private int maxSpeed = 200;

        public int evaluateSpeed(int topSpeed) {
            int recordSpeed = maxSpeed - topSpeed;
            this.saveInLog(recordSpeed);
            return recordSpeed;
        }

        private void saveInLog(int recordValue) {
            System.out.println("Value Logged Successfully");
        }
    }
}

Figure 5.10: Scenario 3
5.2. OBJECT-ORIENTED CODE VS X-MAN

clusters i.e., two atomic components. One atomic component will have one method i.e., 
provideSpeed(). The other atomic component will have evaluateSpeed() and saveIn-
Log(). Figure 5.11 and 5.12 are showing the re-structured code for two clusters from 
the scenario 3. Figure 5.13 is showing the extracted components from this scenario 
and a possible case of their composition.

At deployment phase, the Sequencer will trigger the first atomic component (ascribed 
with the lower number i.e., higher priority) and exposed method provideSpeed() of 
component 1 will be executed. Then, the Sequencer will trigger the second compo-
nent and its exposed method (evaluateSpeed()) will be executed. This exposed method 
then, invokes the private method in its computation unit i.e., saveInLog(). This map-
ping extracts the control of execution out from the computation of methods (as needed
for code-independent re-usability). As the method `saveInLog()` is a utility method, another utility cluster will also be extracted. All utility clusters will be mapped as atomic components as shown in the Figure 5.14.

The above scenarios showed the basic mapping from object-oriented code to atomic X-MAN component and it consists of following steps:

- clustering of methods that belong together, based on their invocations and modifiers.
- code re-structuring as an interface (exposed methods) and implementation (implementation of all the methods).
- mapping of each cluster to an atomic X-MAN component i.e., mapping of exposed methods to `E.Methods`, exposed method's parameters to `Input`, exposed method's return to `Output`, each method's implementation to `Method` and implementation class to `Computation Unit` in X-MAN meta-model.

Clustering of methods based on their invocations and modifiers provide good cohesion as only those methods would belong to the same cluster that are associated and have loose coupling with rest of the components. In case of more than two classes, same rules will be used against invocation list of each method, whether the invocations are direct or indirect (invoked via some other method indirectly), to associate every method in a cluster. The granularity and rules of method's invocations and modifiers are dependent on the defined boundary of a cluster. For illustration and explanation, this chapter assumes the class level granularity (i.e., look for the public-private checks when a call goes out of a class). The algorithms and further details of implementation will be presented in Chapter 6.

Along with the interactions and modifiers of methods in a code base, there are few
5.3 Mapping Rule Base for Control Statements

The previous section (Section 5.2.1) showed the mapping from object-oriented source code to clusters i.e., atomic X-MAN components, based on the invocations and modifiers of methods in a code base. But, that mapping did not discuss or include the control structures in a code base. Control structures of a code base are based on control statements that define the order of execution, based on specific conditions. To preserve the functionality and semantics of a code base, reverse engineered components need to have a mapping from control statements to semantics of X-MAN component model. Deposition of atomic X-MAN components, mapped from clusters is applicable if all the interactions are just method calls, without any control structure in any cluster. But,
that is not the case for an actual code base. In case of control structures, every associated method call and control statement of a cluster have to be mapped as a composite X-MAN component. The allocation of methods to atomic components of that composite, and the composition connectors between those atomic components depends on the nature of the control structures between execution of methods.

Control statements in a code base can be divided into two main categories:

a) statements based on branching (they include if-else, switch etc.).

b) statements based on looping (they include while, for and do-while).

These control statements can exist in a code base in any combination. A loop can have many if statements and a switch statement can have many loops. This section will map the possible combinations of control statements by showing the equivalent X-MAN notation. To demonstrate the flow of control in reverse engineered components, this section will show the equivalent flow chart notation for every control statement in a code base, followed by the equivalent X-MAN notation for every flow chart. This mapping will help to understand the semantics that our approach uses to map the control structures in a legacy code to equivalent X-MAN representation without losing any behaviour (control flow) of the code base.

The general idea behind the mapping of control structures is to map two method of a cluster to two different atomic components by adding the behaviour of associated control structure between them, using composition connectors (adapter connectors for atomic components). If two methods; m1 and m2 belong to the same cluster (by rules of invocation), but instead of simple invocation, their invocation includes a control structure between them, the control structure has to be mapped to appropriate adapter or composition connector to compose a composite component. Computation units of such composite components will only have method's implementation and the control will be extracted out, separated from the computation.

The section below will demonstrate the code scenarios with each control statement (and few collaboration of different control statements) to demonstrate the rule base. For the succeeding section, the rule for the utility cluster is still the same, though utility components will not be discussed or presented, as the concept is already presented in the Section 5.2.1. Every cluster can have its utility cluster part if there is at least one utility method present in that cluster.
5.3. MAPPING RULE BASE FOR CONTROL STATEMENTS

```
public class A {
    B obj;

    public int provideSpeed(int speed) {
        speed = 100;
        obj = new B();
        obj.evaluateSpeed(speed);
        return speed;
    }

    public class B {
        private int maxSpeed=200;

        public int evaluateSpeed(int topSpeed) {
            int recordSpeed = maxSpeed - topSpeed;
            if (recordSpeed < 100)
                this.saveInLog(recordSpeed);
            return recordSpeed;
        }

        private void saveInLog(int recordValue) {
            System.out.println("Value Logged Successfully");
        }
    }
}
```

Figure 5.15: Simple if Statement

### 5.3.1 Mapping the if and if-else statements

- **if Statement:**
  
  **Rule:** Every method invocation under if statement in a code base will be mapped by sequencing a conditional atomic component (by Guard) with rest of the associated atomic components of a composite one.

  The most trivial scenario is a single if statement in a code base. For simple if statement, consider a scenario shown in Figure 5.15.

  In the Figure, method `provideSpeed()` calls method `evaluateSpeed()` from another class. The method `evaluateSpeed()` calls a private method in its class, based on a conditional statement (if statement). Figure 5.16 is showing the flow chart notation to demonstrate the control flow of the given scenario. Similar to scenario 3 of section 5.2.1, the code should be mapped to two atomic components. But, this time the second component will be further composed into two atomic components to implement the conditional statement as Guard. Guard is an adapter connector in X-MAN that invokes an atomic component based on the given condition (Appendix B).

  Figure 5.17 is showing re-structured code for the first cluster and Figure 5.18,
CHAPTER 5. OBJECT-ORIENTED CODE VS X-MAN

Figure 5.16: Control Flow for Figure 5.15.

```java
package com.A.scenario;

public interface A {
    public int provideSpeed(int speed);
}
```

```java
package com.A.scenario;

class AImpl implements A {
    public AImpl() {
    }

    public int provideSpeed(int speed) {
        speed = 100;
        return speed;
    }
}
```

Figure 5.17: Re-structured Code (Cluster 1) from Figure 5.15

5.19 are showing the re-structured code for the atomic components of second cluster (these components will be composed to form a composite). Figure 5.20
5.3. MAPPING RULE BASE FOR CONTROL STATEMENTS

Figure 5.18: Re-structured Code (Cluster 2) from Figure 5.15 for First Atomic Component of the Composite

```java
package com.B.scenario;
public interface B {
    public int evaluateSpeed(int topSpeed);
}
```

```java
package com.B.scenario;
public class BImpl implements B {
    private int maxSpeed = 200;
    public int evaluateSpeed(int topSpeed) {
        int recordSpeed = maxSpeed - topSpeed;
        return recordSpeed;
    }
}
```

Figure 5.19: Re-structured Code (Cluster 2) from Figure 5.15 for Second Atomic Component of Composite

```java
package com.C.scenario;
public interface C {
    private void saveInLog(int recordValue);
}
```

```java
package com.C.scenario;
public class CImpl implements C {
    private void saveInLog(int recordValue) {
        System.out.println("Value Logged Successfully");
    }
}
```

is showing an atomic component mapped from cluster1 and a composite component mapped by composing cluster 2 and 3 (in coloured box). The Figure is also showing a possible case of composition between these two components. In Figure 5.20, the Sequencer will invoke first atomic component with exposed method `provideSpeed()`. The execution of method will also provide the data output to input of the exposed method `evaluateSpeed()` of component C2 (part of the composite component). Sequencer of composite component will execute `evaluateSpeed()` of C2 and `saveInLog()` of C3 respectively. Then, control will go back to the outer Sequencer and it will terminate the execution. The order of control flow in the given composition is same as shown in the flow chart of the given scenario (Figure 5.16). Hence, the functionality and the control flow of original source code is preserved in the reverse engineered component. These two components (composed by Sequencer) can be re-deposited for future reuse of the same composition.
• *if-else Statement:*

**Rule:** Every method invocation under if-else statement will be mapped as an atomic component under a Selector. The number of branches of Selector are dependent on the number of respective else parts and a Selector can have multiple conditions to facilitate each path.

The next possible variation of if statement is the associated else part with it. Association of else with if requires a Selector (a Selector can have many inputs as conditional statements (Figure B.5)) rather than Guard. The number of possible branches of that Selector are dependent on the associated else statements and the invocation of methods in every control statement. Figure 5.21 is showing a scenario that includes the if-else statement. In the Figure, method `provideSpeed()` calls method `evaluateSpeed()` from another class. The method `evaluateSpeed()` calls one of the two private methods, depending on the value of speed. The given scenario has one associated else part with if statement and another standalone if statement in the end. The control flow (flow chart) for this code base is shown in Figure 5.22.

Figure 5.23 is showing re-structured code for the first cluster and Figure 5.24,
public class A {
    B obj;

    public int provideSpeed(int speed) {
        speed = 100;
        obj = new B();
        obj.evaluateSpeed(speed);
        return speed;
    }
}

public class B {
    private int maxSpeed = 200;

    public int evaluateSpeed(int topSpeed) {
        int recordSpeed = maxSpeed - topSpeed;
        if (recordSpeed < 100)
            this.saveInLog(recordSpeed);
        else if (recordSpeed > 100)
            this.discardLog();
        if (maxSpeed < 100)
            this.resume();
        return recordSpeed;
    }

    private void saveInLog(int recordValue) {
        System.out.println("Value Logged Successfully");
    }

    private void discardLog() {
        System.out.println("Log Discarded");
    }

    private void resume() {
        System.out.println("Speed Resumed");
    }
}

Figure 5.21: if-else Statement
5.25, 5.26 and 5.27 are showing the re-structured code for the atomic components of second cluster (these components will be composed to form a composite), extracted from the code of Figure 5.21. Figure 5.28 is showing the extracted atomic and composite component (with break down of control in coloured box). This Figure is also showing a case of composition to reflect the behaviour of given code base. In the given composition, the Sequencer will invoke first atomic component with exposed method provideSpeed(). The execution of method also provides the output to input of the exposed method evaluateSpeed() of C2 (belongs to composite component). Then, the Sequencer will invoke the route 1 i.e., second Sequencer. After the execution of evaluateSpeed() by second Sequencer, the Selector will be invoked and the Selector will invoke one of the two atomic components (for if-else), depending on the conditional value. After
the execution of the selected component, the second *Sequencer* will invoke the last atomic component that have the conditional statement for its execution (if statement). After execution of $C_5$ (if condition holds), control will go back to inner and outer *Sequencer* respectively and the execution will terminate. This composition is showing the same control flow as shown by the flow chart of the given code scenario in Figure 5.22. If there are $n$ number of consecutive *if-else* statements in a code base, all the invoked methods inside these statements will be mapped by a *Selector* and sequenced with rest of the associated atomic components. Any independent *if* statement after that will be mapped using *Guard*. This scenario has explained the basic rules of mapping for *if-else* and independent *if* statements in a method’s body.

Figure 5.23: Re-structured Code (Cluster 1) from Figure 5.21

Figure 5.24: Re-structured Code (Cluster 2) from Figure 5.21 for First Atomic Component of Composite
Figure 5.25: Re-structured Code (Cluster 2) from Figure 5.21 for Second Atomic Component of Composite

Figure 5.26: Re-structured Code (Cluster 2) from Figure 5.21 for Third Atomic Component of Composite

Figure 5.27: Re-structured Code (Cluster 2) from Figure 5.21 for Fourth Atomic Component of Composite
5.3. MAPPING RULE BASE FOR CONTROL STATEMENTS

### Figure 5.28: Mapped Atomic and Composite Component for if-else Statement
5.3.2 Mapping the Switch statements

**Rule:** Every method invocation under switch statement will be mapped as an atomic component under a Selector. The number of branches of Selector are dependent on the number of cases of switch with method invocation and a Selector can have multiple conditions to facilitate each case.

Switch statement is very similar to if-else statement. Though, switch is usually used for branching, based on a single expressions. The basic rule base behind switch is same as if-else statement i.e., branching of the methods (by Selector) that are invoked inside switch cases. From implementation point of view, the parser will look for the break statement to verify if more than one case should belong to same computation unit (along with the statements before each method invocation in the case). Instead of just showing the mapping via Selector for switch statement, the code scenario (Figure 5.29) is showing a nested case of switch within the if-else statement. Figure 5.30 is showing flow chart for the given code scenario to show the control flow.

Figure 5.31 is showing re-structured code for the first cluster and Figure 5.32, 5.33, 5.34, 5.35 and 5.36 are showing the re-structured code for the atomic components of second cluster (these components will be composed to form a composite). Figure 5.37 is showing the atomic and composite component (consists of five atomic components) extracted from the given code scenario. In the extracted composite component (in coloured box), there are two Selectors instead of one to map the nested if-else inside switch statement.

Figure 5.37 is also showing a case of composition between the extracted atomic and composite component via Sequencer. The Sequencer will invoke C1 (atomic component) and exposed method provideSpeed() will be executed. Then, the Sequencer will invoke composite component via another Sequencer. The inner Sequencer will invoke C2 and then the first Selector. The Selector will either invoke C3, C4 or the second Selector. In case of second Selector, either C5 or C6 will be invoked and the control will go back to the first Sequencer via the second one. The flow of control in the given composition is same as in the flow chart of Figure 5.30. But, the control has been extracted out from the computation of methods.
public class A {
    B obj;

    public int provideSpeed(int speed) {
        speed = 100;
        obj = new B();
        obj.evaluateSpeed(speed);
        return speed;
    }
}

public class B {
    private int maxSpeed = 200;
    public void evaluateSpeed(int topSpeed) {
        int recordSpeed = maxSpeed - topSpeed;

        if (recordSpeed < 100) {
            switch(recordSpeed) {
                case 90: saveLog(recordSpeed); break;
                case 80: discardLog(); break;
                default: resume(); break;
            }
        } else {
            this.stop();
        }
    }

    private void saveLog(int recordValue) {
        System.out.println("Value Logged Successfully");
    }

    private void discardLog() {
        System.out.println("Log Discarded");
    }

    private void resume() {
        System.out.println("Speed Resumed");
    }

    private void stop() {
        System.out.println("Stop Vehicle");
    }
}

Figure 5.29: Switch Statement
CHAPTER 5. OBJECT-ORIENTED CODE VS X-MAN

Figure 5.30: Control Flow for Figure 5.29.

package com.A.scenario;
public interface A {
    public int provideSpeed( int speed);
}

package com.A.scenario;
public class AImpl implements A {
    public AImpl() {}
    public int provideSpeed( int speed) {
        speed = 100;
        return speed;
    }
}

Figure 5.31: Re-structured Code (Cluster 1) from Figure 5.29
5.3. MAPPING RULE BASE FOR CONTROL STATEMENTS

Figure 5.32: Re-structured Code (Cluster 2) from Figure 5.29 for First Atomic Component of Composite

```java
package com.B.scenario;
public interface B {
    public int evaluateSpeed(int topSpeed);
}
```

```java
package com.B.scenario;
public class BImpl implements B {
    private int maxSpeed = 200;
    public int evaluateSpeed(int topSpeed) {
        int recordSpeed = maxSpeed - topSpeed;
        return recordSpeed;
    }
}
```

Figure 5.33: Re-structured Code (Cluster 2) from Figure 5.29 for Second Atomic Component of Composite

```java
package com.C.scenario;
public interface C {
    private void saveInLog(int recordValue);
}
```

```java
package com.C.scenario;
public class CImpl implements C {
    private void saveInLog(int recordValue) {
        System.out.println("Value Logged Successfully");
    }
}
```

Figure 5.34: Re-structured Code (Cluster 2) from Figure 5.29 for Third Atomic Component of Composite

```java
package com.D.scenario;
public interface D {
    private void discardLog();
}
```

```java
package com.D.scenario;
public class DImpl implements D {
    private void discardLog() {
        System.out.println("Log Discarded");
    }
}
```
package com.E.scenario;

public interface E {
    private void resume();
}

package com.F.scenario;

public interface F {
    private void stop();
}

Figure 5.35: Re-structured Code (Cluster 2) from Figure 5.29 for Fourth Atomic Component of Composite

package com.E.scenario;

public class EImpl implements E {
    private void resume() {
        System.out.println("Speed Resumed");
    }
}

package com.F.scenario;

public class FImpl implements F {
    private void stop() {
        System.out.println("Speed Resumed");
    }
}

Figure 5.36: Re-structured Code (Cluster 2) from Figure 5.29 for fifth Atomic Component of Composite

Figure 5.37: Mapped Atomic and Composite Component for if-else Statement With Nested switch
5.3. MAPPING RULE BASE FOR CONTROL STATEMENTS

5.3.3 Mapping the Loop based statements

Loop based statements can be implemented in following three common ways: a) while loop. b) do while loop. c) for loop. The difference in the control flow requires the different mapping for every type of loop. The loop based control statements in a code base will be mapped to Loop which is an adapter connector in X-MAN. The Loop connector repeats the execution of an atomic component, based on the given condition of looping.

• **While Loop:**
  
  **Rule:** Every While loop in the code is mapped to Loop connector with initial value of iterator, iteration interval and the condition of termination. If a loop statement does not have the iterator, the termination condition will be used to iterate it repeatedly without mapping values of iterator and iteration interval.

  The Figure 5.38 is showing a code base with a while loop in it and the Figure 5.39 is showing the flow chart to demonstrate the flow of control in the given scenario. Figure 5.40 is showing re-structured code for the first cluster and Figure 5.40, 5.41, 5.42 and 5.43 are showing the re-structured code for the atomic components of second cluster (these components will be composed to form a composite).

  Figure 5.44 is showing the atomic and composite component (consists of three atomic components, in the coloured box) extracted from these clusters. The Figure is also showing a possible composition between these two extracted components to reflect the flow control in the given code base. The given composition will start with invocation of C1 by a Sequencer. After the execution of C1, Sequencer will execute the inner Sequencer. The inner Sequencer will execute the C2 and Selector respectively. The Selector will either invoke C3 or keep in invoking C4 until the Loop condition holds. Loop connector has initial value of counter and the condition of termination. On termination of Loop, control will go back to outer Sequencer via the inner one. The flow of control in the given composition is same as shown in the flow chart of given scenario (Figure 5.39).

• **do While Loop:**
  
  **Rule:** Every do While loop in the code is mapped to an atomic component with required execution under Loop Connector (with initial value of iterator, iteration interval and the condition of termination.), sequenced by the same component.
```java
public class A {
    B obj;

    public int provideSpeed(int speed) {
        speed = 100;
        obj = new B();
        obj.evaluateSpeed(speed);
        return speed;
    }
}

public class B {
    private int maxSpeed = 200;
    int index=10;

    public int evaluateSpeed(int topSpeed) {
        int recordSpeed = maxSpeed - topSpeed;
        if (recordSpeed < 100)
            this.saveInLog(recordSpeed);
        else if (recordSpeed > 100)
            while (index < recordSpeed)
                this.discardLog();
            index++;
        
        return recordSpeed;
    }

    private void saveInLog(int recordValue) {
        System.out.println("Value Logged Successfully");
    }

    private void discardLog() {
        System.out.println("Log Discarded");
    }
}
```

Figure 5.38: While Loop with if Statement
5.3. MAPPING RULE BASE FOR CONTROL STATEMENTS

Sequence of Methods
- `provideSpeed`
- `evaluateSpeed`
- `recordSpeed < 100`
- `saveInLog`
  - `Yes`
  - `No`
- `discardLog`
- `index < record Speed`
- `True`
- `discardLog`
- `index++`

Figure 5.39: Control Flow for Figure 5.38.

```java
package com.A.scenario;

public interface A {
    public int provideSpeed(int speed);
}
```

```java
package com.A.scenario;

public class AImpl implements A {
    public AImpl() {}
    public int provideSpeed(int speed) {
        speed = 100;
        return speed;
    }
}
```

Figure 5.40: Re-Structured Code (Cluster 1) from Figure 5.38
CHAPTER 5. OBJECT-ORIENTED CODE VS X-MAN

Figure 5.41: Re-structured Code (Cluster 2) from Figure 5.38 for First Atomic Component of the Composite

```java
package com.B.scenario;

public interface B {
    public int evaluateSpeed(int topSpeed);
}
```

```java
package com.B.scenario;

public class BImpl implements B {
    private int maxSpeed = 200;
    public int evaluateSpeed(int topSpeed) {
        int recordSpeed = maxSpeed - topSpeed;
        return recordSpeed;
    }
}
```

Figure 5.42: Re-structured Code (Cluster 2) from Figure 5.38 for Second Atomic Component of the Composite

```java
package com.C.scenario;

public interface C {
    private void saveInLog(int recordValue);
}
```

```java
package com.C.scenario;

public class CImpl implements C {
    private void saveInLog(int recordValue) {
        System.out.println("Value Logged Successfully");
    }
}
```

Figure 5.43: Re-structured Code (Cluster 2) from Figure 5.38 for Third Atomic Component of the Composite

```java
package com.D.scenario;

public interface D {
    private void discardLog();
}
```

```java
package com.D.scenario;

public class DImpl implements D {
    int index = 0;
    private void discardLog() {
        System.out.println("Log Discarded");
    }
}
```
Figure 5.44: Mapped Atomic and Composite Component for While Loop With if-else Statement
The do While loop is different from the While loop because it executes at least once, even if the condition of looping is false. The main difference in mapping between While and do While is the addition of another component (same as C4) in Figure 5.44 that will be sequenced with the C4 (component under Loop). Figure 5.45 is showing the composite component that will be mapped in case of do While from code base of Figure 5.38. This mapping will make sure that the atomic component under do While (discardLog()) will execute at least once.
5.3. MAPPING RULE BASE FOR CONTROL STATEMENTS

- **for Loop**:
  
  **Rule:** Every for loop in the code is mapped to the relevant component under Loop connector with initial value of iterator, iteration interval and the condition of termination. For nested loops, each nested case will be mapped to another Loop connector and a Selector to handle condition of execution for both loops (inner and outer).

Control in the for loop works similar to While loop. To demonstrate a little bit complexity, Figure 5.46 is showing a scenario with nested for loop and Figure 5.47 is showing a flow chart to demonstrate the flow of control in the given scenario. Figure 5.48 is showing re-structured code for the first cluster, and Figure 5.49, 5.50 and 5.51 are showing the re-structured code for the atomic components of second cluster (these components will be composed to form a composite).

Figure 5.52 is showing the extracted atomic and composite component (consist of three components, in the coloured box) and a possible case of composition to reflect the control flow in the given code base.

The composition will start by invocation of C1 by Sequencer. After executing C1, Sequencer will invoke the composite component via its Sequencer. In the composite component, C2 will be executed by the Sequencer. Then, the call to Loop will be validated by termination condition in the first Loop. If the condition is fulfilled, second Loop (for nested for) will be validated and the Selector will check the condition for second for loop. If this condition is fulfilled, component C4 will be executed. After that, the control will go back to the inner Loop and it will invoke C4 again. Same iteration will be repeated until the condition of Selector invokes C3 i.e., the condition of inner Loop will be invalidated and the control will go back to the outer Loop, and eventually to Sequencer after termination condition in the outer Loop.

Similarly, n nested loops can be added by following the same hierarchy of the inner Loop and the Selector. From implementation point of view, parser will capture the for loop nodes via a visitor and nested loops will be checked by regular expressions. For n number of consecutive loops, n number of same hierarchical structure will be mapped. The control flow of this composition is same as shown by the flow chart of this scenario (Figure 5.47). All these scenarios show that X-MAN connectors work in a hierarchy. Control from the top connector goes
public class A {

    B obj;

    public int provideSpeed(int speed) {
        speed = 100;
        obj = new B();
        obj.evaluateSpeed(speed);
        return speed;
    }
}

public class B {

    private int maxSpeed = 200;
    int index=0;
    int index2=0;

    public int evaluateSpeed(int topSpeed) {
        int recordSpeed = maxSpeed - topSpeed;
        for (index=0; index < recordSpeed; index++) {
            for (index2=1; index2 < recordSpeed; index2++) {
                this.saveInLog(recordSpeed);
            }
            this.discardLog(recordSpeed);
        }
        return recordSpeed;
    }

    private void saveInLog(int recordValue) {
        System.out.println("Value Logged Successfully");
    }

    private void discardLog(int recordValue) {
        System.out.println("Log Discarded");
    }
}

Figure 5.46: Nested for Loop
5.3. MAPPING RULE BASE FOR CONTROL STATEMENTS

Figure 5.47: Control Flow for Figure 5.46.

Figure 5.48: Re-Structured Code (Cluster 1) from Figure 5.46

package com.A.scenario;

public interface A {
    public int provideSpeed(int speed);
}

package com.A.scenario;

public class AImpl implements A {
    public AImpl() {}
    public int provideSpeed(int speed) {
        speed = 100;
        return speed;
    }
}

down till the computation units and each connector returns the control to the connector above it i.e., the execution stops when control gets back to the top.

All the scenarios above have explained the mapping of control structures to X-MAN components. The proposed approach can extract n number of atomic or composite components, depending on the nature of interactions between methods. One major
benefit of extracting the control out from computation is re-usability and the ease to add new components in the extracted system. The extracted X-MAN components can be composed with newly created components without changing or accessing the code in them. Composition connectors and exposed methods can define any required
configuration of composition though, semantically correct composition requires the knowledge of the code base being reverse engineered.

All the control structures mapped above can exist in a code base in any combination and the approach will use the same mapping rules to connect the structures and their respective methods step by step. From implementation point of view, the methods that belong together (in a cluster) will be accessed and the control structure between them will define the semantics of composition connectors. The succeeding section will explain an example of brake control system to demonstrate the mapping of multiple control structures that exist together in a code base.
5.4 Implementing the Brake Control System

Fig 5.53 shows a simple example of brake control system to show the applicability of above scenarios. In the given example, there are two classes. Class SpeedMon-

```java
package vehicle.control.speedcontrol;
import vehicle.brakecontrol.BreakControl;

public class SpeedMonitoring {
    BreakControl obj;

    public SpeedMonitoring() {
    }

    public double collisionTimeCalculator(double speed, double distance) {
        speedMonitoringValue(distance / speed);
        return distance / speed;
    }

    public void speedMonitoringValue(double collisionEstimate) {
        if (collisionEstimate < 15) {
            obj.collisionParametersActivation(true, collisionEstimate);
        } else if (collisionEstimate < 25) {
            BrakeSystemActivation();
        } else if (collisionEstimate < 25) {
            while (collisionEstimate > 15) {
                wait.sleep(1);
                TimeTriggerValue(flag);
                collisionEstimate--;
            }
            return flag;
        }

        private void BrakeSystemActivation() {
            System.out.println("Brakes Applied");
        }

        public void TimeTriggerValue() {
            System.out.println("Warning, Possible Collision Upcoming");
        }
    }
}
```

Figure 5.53: Brake Control System

`SpeedMonitoring` has methods `collisionTimeCalculator` (for calculating time till collision) and `speedMonitoringValue` (for automatic brake mode if time till collision is less than 15 seconds). Method `speedMonitoringValue()` invokes `collisionParametersActivation()` from Class `BreakControl`. Depending on the value of `collisionEstimate`, method `collisionParametersActivation()` either invokes `BrakeSystemActivation()` or `TimeTriggerValue()`.

According to rule base, method `speedMonitoringValue()` has one method node
5.4. IMPLEMENTING THE BRAKE CONTROL SYSTEM

against its invocation node i.e., collisionParametersActivation() and method collisionParametersActivation() has two method nodes against its invocation node i.e., BrakeSystemActivation() and TimeTriggerValue() (hence two indirect invocation nodes against speedMonitoringValue()). As the method BrakeSystemActivation() is private (scenario 3) therefore, the approach will map the whole code to two clusters.

First cluster will have methods speedMonitoringValue() and collisionTimeCalculator(). Second cluster will have methods collisionParametersActivation(), BrakeSystemActivation() and TimeTriggerValue(). From second cluster, BrakeSystemActivation() and TimeTriggerValue() are utility methods therefore, there will be another cluster i.e., utility cluster for cluster 2 (Figure 5.56).

Figure 5.54 and 5.55 are showing the re-structured code for both clusters of Brake Control System. Both the extracted clusters have control structures in them therefore, they cannot be mapped as atomic components. Figure 5.57 is showing the breakdown of both clusters into composite components. The Figure is also showing a composition of both composite components via Sequencer.
package vehicle.brakecontrol;
import java.util.*;
import java.util.Date;

public interface BrakeControl {
    public boolean collisionParametersActivation(boolean flag, double estimate);
    public void TimeTriggerValue(boolean value);
    private void BrakeSystemActivation();
}

package vehicle.brakecontrol;
import java.util.*;
import java.util.Date;

public class BrakeControlImpl {
    public boolean collisionParametersActivation(boolean flag, double estimate) {
        int Estimate = Integer.parseInt(estimate);
        if (flag == true) { 
            BrakeSystemActivation();
        } else if (Estimate < 25) {
            while (Estimate > 15) {
                wait.sleep(1);
                TimeTriggerValue(flag);
                Estimate--;
            }
        }
        return flag;
    }

    private void BrakeSystemActivation() {
        System.out.println("Brakes Applied");
    }

    public void TimeTriggerValue() {
        System.out.println("Warning, Possible Collision Upcoming");
    }
}

Figure 5.55: Brake Control System: Cluster 2

In Figure 5.57, component C1 will be instantiated and will provide collision Estimate to C2 and C3. Depending on the condition in Selector, either C2 or C3 will be invoked. The Sequencer that is composing two composite components will invoke the Sequencer of second composite component. The inner Sequencer will invoke C4 and the next Selector respectively. Depending on the conditions in this Selector, either C5 will be executed once or C6 will be executed iteratively until the mapped termination condition to Loop will be fulfilled (i.e., iterative counter will be set to the value of Estimate from the code, termination condition will be 25 and direction of counter will be reversed).

The given example showed that six components have been extracted from two
5.4. IMPLEMENTING THE BRAKE CONTROL SYSTEM

clusters due to the presence of control structures in the clusters. The flow of execution of methods in the composed components of Figure 5.57 is same as the flow of the source code. But, all the control has been extracted out from the computation of methods, and by such mapping we have achieved exogenous composition. Along with these six components, the utility cluster will also be mapped as an atomic component (Figure 5.58). In all the code scenarios and brake control system, encapsulation is preserved in all the extracted components i.e., a method from a computation unit CU1 of component C1 cannot interact or call any method from another computation unit CU2 of component C2 directly but only via composition connectors that define the order of execution of all the extracted methods. In other words, the extracted components are fully cohesive and there is no coupling between them.
CHAPTER 5. OBJECT-ORIENTED CODE VS X-MAN

Figure 5.57: Composition of Brake Control System

Figure 5.58: Utility Component from Cluster 2
5.5 Summary

The component mapper proposed in this chapter mapped the different scenarios of object-oriented code to X-MAN meta-model. The mapping starts by clustering the methods, based on the defined granularity of clusters and rules of invocations. This mapping works by mapping the: interfaces of a cluster to exposed methods, implementation of methods in a cluster to computation unit and inputs/outputs of exposed methods to inputs/outputs of X-MAN meta-model.

For those clusters that have control structures, their control statements will be mapped to respective adapter and composition connectors. Such clusters will be deposited as composite components in which, each atomic component will only have the computation of methods and control will be extracted out from each atomic component. Such transformation provides code-independent re-usability as components can be composed by connecting appropriate connectors without configuring code for every composition. Utility clusters are based on the utility methods that do not call other methods. Each utility cluster will be deposited as an atomic component to reuse these methods to their full potential. Next chapter will discuss the implementation strategy and the captured notation of source code for reverse engineering.
Chapter 6

Reverse Engineering X-MAN

“The most effective way to do it, is to do it.”

— Amelia Earhart

6.1 Introduction

The chapter 4 stated that a component directed reverse engineering approach consists of four major steps: 1) capturing the code base to reverse engineer. 2) re-structure the captured code to a map-able notation. 3) a defined rule base to map the captured code notation to a set of clusters. 4) mapping of clusters to the desired notation (component model). This chapter includes the implementation (code capturing, re-structuring and implementation of the rule base) of Chapter 5. Red box in Figure 6.1 is showing the

Figure 6.1: RX-MAN: Capturing and Mapping the Code Base.
entities of our approach that are included in this chapter.

Section 6.2 of this chapter discusses the parser which is used for capturing the code base. This section also explains the notation of re-structured code for analysis. Section 6.3 explains the "Code Mapper" that analyse the re-structured captured code for clustering and mapping to X-MAN components, using the rule base of Chapter 5. The output of "Code Mapper" is Java code in the notation of X-MAN components, that can be deposited.
CHAPTER 6. REVERSE ENGINEERING X-MAN

6.2 Capturing the Code Base

A code base can be captured in multiple notations. Few approaches (e.g., [WF14]) use the directed graphs. Few uses UML notations (e.g., [KC04]) and few like us, use a parser (e.g., [DPB14]). In every notation, the main considerable factor is to capture a code base without losing any functionality or semantics.

Our approach is based on object-oriented code and it uses Java as a language of implementation. The source code it can take as an input can be an application, a library or an API. The parser that our approach uses is an extension of Abstract Syntax Tree (AST) parser. AST parser extracts one big tree of nodes from a code base in which all the nodes are connected hierarchically e.g., starting node will be compilation unit (class level or package level) connected with its sub nodes i.e., class declarations, class variables etc. Each class declaration node is further connected to its method nodes and each method node is connected with its sub nodes i.e., method parameters nodes, method return node, method body node etc. This hierarchy of nodes goes to the last level which is simple name nodes i.e., name of local variables (e.g., int abc etc.).

6.2.1 Nodes Structuring by RX-MAN Parser

The parser designed for this approach (RX-MAN parser) is more powerful than the default AST parser due to following reasons: 1) it also extracts and maps invocation nodes from each method node in a code base e.g., if a method A invokes method B, and method B invokes method C then, the parser can extract and connect all nodes of the method C to method A as both are indirectly connected by method B. 2) it uses special visitors to locate if-else, switch and loops in a method's body. These visitors are used to locate the control structures in the extracted clusters (AST has this functionality by default but the proposed parser associates the functionality with connectors of X-MAN). 3) the parser is also associated with a defined back end model that is used to convert the outcome of "Code Mapper" to Java code before depositing the components. 4) the parser includes the implementation of defined rules (of Chapter 5) to map adapter/composition connectors with their input values (e.g., iterator for Loop) from control structures in every cluster (The details of implementation and the relevant code is presented in the Appendix C).

RX-MAN parser indexes each method of a code base and connect all associated nodes with every method. Figure 6.2 is showing the extraction of nodes using RX-MAN parser. Each indexed method node is associated with its parent class (Type Dec-
6.2. CAPTURING THE CODE BASE

Figure 6.2: RX-MAN Parser

laration Node in Figure 6.2), parent package and class variables (Field Declaration Node in Figure 6.2), this method uses. Along with this information, each indexed method node is connected with all the method it invokes directly or indirectly (Invoked List Node in Figure 6.2) along with the control structures of every method (Statement Type Node in Figure 6.2). This mapping makes sure that no indirect invocation goes undetected. In short, starting from each method in a code base, each indexed method node is connected with whole chain of invocations it causes in a code base. The proposed rule base will use this notation to start analysing and clustering each method, based on the rules of invocations and modifiers (Section 5.2.1).

The overall process of extracting and structuring nodes is shown in the algorithm 1. In algorithm 1, first step is to extract AST nodes (by customised AST parser) of a code base (ExtractAST() in the algorithm). Each method node from this instance is indexed (Indexed(Mi)) and connected with its parent nodes (i.e., the nodes of class, package and class fields by (ConnectParentsNodes())) and control statement nodes (GetControlStatementNodes()) in that method. A designed visitor extracts each method's node invoked by every indexed method (GetInvocationListByVisitor()). For every method node invoked by every indexed method, its parent nodes, control structure nodes and dependency nodes are connected with it by a specified node called InvokedList. All the invoked methods are connected with every indexed method using the same semantics.
Algorithm 1 Extraction and Structuring of Nodes from a Code Base

Require: Source Code

ExtractAST(SourceCode)

while $i < MethodNodeList$ do
  IndexedMethod($Mi$)
  if $Mi.Visited ← False$ then
    ConnectParentsNodes($Mi, Mi.Package, Mi.Class$) //Package.imports & FieldDeclarations
    GetControlStatementNodes($Mi, Mi.Package, Mi.Class$)
    GetInvocationListByVisitor($Mi$)
    while $j < InvocationList(Mi)$ do
      ConnectInvokedMethod($Mi, Mi[j]$)
      ConnectParentsNodes($Mi[j], Mi.Package[j], Mi.Class[j]$)
      GetControlStatementNodes($Mi[j], Mi.Package[j], Mi.Class[j]$)
      $j++$
    end while
  end if
  $i++$
end while

and the end result is presented in Figure 6.2.

6.3 Implementing the Code Mapper

The ”Code Mapper” implements the rule base, defined in Chapter 5. According to that rule base, methods should be clustered, based on their invocations and modifiers. The rule base has also specified that the granularity of clusters has to be selected. In the proposed approach, the granularity can vary from a class (smallest executable entity in object-oriented code) to a package (units in object oriented code used for separation of concerns). The proposed tool presents both these options. The granularity scale can be really useful to set the expected size of extracted components to be used as modular structures. $^1$

Assuming the granularity level of clusters to be at package level, algorithm 2 summarises the clustering of methods, based on the rules of Section 5.2.1.

---

$^1$Package based granularity does not mean that a method $M1$ in class $C1$ of package $P1$ always belongs to the component of $P1$. It means that the maximum number of extracted components cannot exceed the total number of packages in the code whereas, the minimum number of components that can be extracted is 1. Method $M1$ can belong to any component depending on the rules of invocation.
Algorithm 2 Methods_Clustering(Mi, index)

Require: Indexed methods in a code base i.e., Mi and their index reference in the tree i.e., index

while i < IndexedMethodsListSize do
    if Mi.invocationList == Empty then
        if C."utility".concat(PackageName) ← FALSE then
            CreateUtilityCluster("utility".concat(PackageName))
            ADD (Mi,"utility".concat(PackageName))
        else
            ADD (Mi,"utility".concat(PackageName))
        end if
    end if
    while Mi.invocationList != Empty do
        if Mi.PackageName == Mi.invocationList(index).packageName then
            if C.PackageName ← FALSE then
                CreateCluster(PackageName)
            end if
            CheckDuplication(Mi,Mi.invocationList(index),PackageName)
            if Duplication == FALSE then
                ADD (Mi,Mi.invocationList(index),PackageName)
            end if
            ExtractEachInvocation(Mi.invocationList(index),index) //Extract invocations of Mi
        else
            if Mi.invocationList(index).package ← FALSE then
                CheckDuplication(Mi,Mi.invocationList(index),PackageName)
                if Duplication == FALSE then
                    CheckPrivateAccess(Mi.invocationList(index))
                    if CheckPrivateMemberAccess ← FASLE then
                        ADD (Mi,Mi.invocationList(index),PackageName)
                    else
                        SetAllocation(Mi.invocationList(index),PackageName)
                    end if
                end if
                //Start same chain for each invocation of Mi to cover all indirect invocations
                Methods_Clustering(Mi.invocationList(index),index)
            end if
        end if
    end while
end if
In algorithm 2, for each indexed method node, its invocation list is compared with it. If invoked list is empty, the indexed method will be added in a utility cluster. All such utility clusters will be deposited as atomic components as they are specifically useful if the source code is a library i.e., such utility methods can be used by newly created components to use these utility methods.

If the invoked list of indexed method is not empty then, each invoked method's nodes will be compared with the indexed method. If the method Mi and its invoked method belongs to the same package (same Type Declaration node), a cluster with that package name is created (if it does not exist already) and both methods will be placed in it along with the nodes of their imports and the class variables they use. If both belong to different packages, then a privacy check will be conducted by function CheckPrivateAccess(). If the method being invoked, accesses the private variables, class variables or calls private/protected methods in its invocation list (and the invoked entities do not belong to the package of the indexed method), then that method cannot be placed with method Mi in its cluster. In that case, the invoked method will be placed in a newly created cluster (if it does not exist already with the name of the package of invoked method) along with the private entities it is accessing (SetAllocation() in Algorithm 2). The same process will be applied to all the invoked methods in the invocation list of Mi. The function Duplication() in the algorithm 2 checks whether the method being invoked is already part of the cluster because the same method can be part of the invocation lists of many methods. The same algorithm will run for every invocation of every indexed method node in order to cover all the invocations in a code base. At the end of this cycle, each method will be placed in an appropriate cluster in RX-MAN nodes notation. The implementation of methods used in the above algorithm are shown in Appendix C. Same algorithm can be run at class level to restrict class level granularity of clusters.

6.3.1 Example: Clustering of Brake Control System

The captured code of brake control system (the example given in the Figure 5.53) in RX-MAN notation is presented in Appendix C. This section will use the same example of Chapter 5 to demonstrate the clustering of methods, according to algorithm 2.

According to algorithm 2, method speedMonitoringValue() has one method node against its invocation node i.e. collisionParametersActivation() and method collisionParametersActivation has two method nodes against its invocation node i.e. BrakeSystemActivation() and TimeTriggerValue() (hence two indirect invocation nodes against
speedMonitoringValue() via collisionParametersActivation()). As the methods BrakeSystemActivation() and TimeTriggerValue() are private and they belong to a different package (not to the package of method that invokes them indirectly i.e. speedMonitoringValue()) therefore, the approach will map the whole code base to two clusters (scenario 3 in Section 5.2.1). The first cluster will have methods speedMonitoringValue() and collisionTimeCalculator(). The second cluster will have methods collisionParametersActivation(), BrakeSystemActivation() and TimeTriggerValue(). BrakeSystemActivation() and TimeTriggerValue() are utility methods i.e., their invocation list is empty therefore, another cluster (utility cluster for cluster 2) will be created that have these two methods. The end result of algorithm 2 are re-structured clustered RX-MAN nodes as shown in the Figure 6.3.

6.3.2 Mapping from Clusters to Components

Each cluster, extracted by algorithm 2 consists of methods that belong together (based on the rules of invocations). Starting from the first method of every cluster, the "Code Mapper" will start applying rules of mapping of Section 5.3. The end result will be extraction of atomic/composite components from every cluster. In case of a composite component, the control structures will be extracted out from the computation to achieve exogenous composition. Algorithm 3 and 4 are showing the overall strategy of implementation (at an abstract level) of Section 5.3 (mapping from clusters to X-MAN components).

In algorithm 3, if a cluster is utility (it contains all utility methods of that cluster), all the methods in it will be mapped to an atomic X-MAN component. For non-utility
Figure 6.3: Clustering of Brake Control System
Algorithm 4 Process $M_i$

Require: List $<\text{StackStructure}>$ stack[]

List list = $M_i$.getstructuralType().getChildListDescriptor()
stack.add([list[]])

for i = 0; i < list.size(); i++ do
  if node has type of if Statement then
    stack[$M_i$][i].getStartPosition
    stack[$M_i$][i].getEndPosition
  if list[i].getNodeType().hasElsePresent() then
    stack[$M_i$][i].hasElsePresent() ← TRUE
  else if list[i].getNodeType() has stand alone Else then
    stack[$M_i$][i].hasStandAloneElse() ← TRUE
  end if
end if

if (list[i].getNodeType() == WhileStatement()) ||
  (list[i].getNodeType() == ForStatement()) ||
  (list[i].getNodeType() == doWhileStatement()) ||
  (list[i].getNodeType() == MethodInvocation()) ||
  (list[i].getNodeType() == SwitchStatement()) ||
  (list[i].getNodeType() == Statementexpressions()) then
  stack[$M_i$][i].getStartPosition
  stack[$M_i$][i].getEndPosition
end if

for a = 0; a < stack.size(); a++ do
  for b = a + 1; b < stack.size(); b++ do
    if (stack[$M_i$][a].getStartPosition() < stack[$M_i$][b].getStartPosition())
      AND(stack[$M_i$][a].getEndPosition() > stack[$M_i$][b].getEndPosition()) then
      stack[$M_i$][b].setParent(stack[$M_i$][a])
      stack[$M_i$][a].setChild(stack[$M_i$][b])
    end if
  end for
end for

SortNodes(stack)

clusters, only those methods of these cluster will be considered to be processed that
have at least one method invocation node in their structures (utility methods will be
mapped automatically as other methods call them in their body).

Algorithm 4 takes each selected method from algorithm 3 and gets all the child nodes of each method. Child nodes can be present in any possible combination e.g., an if statement can have a loop in it or a method invocation node can be present in a switch statement node. Before extraction of components, it is important to get the order of control structure and method invocations inside each method. Algorithm 4 stores all the child nodes of a method in a two-dimensional list (stack) of type StackStructure (defined in the model). This list has all the child nodes of a method against method's index 2. Depending on start and end position of each node, this algorithm calculates the parent and child relation between all the nodes of a method. If two nodes n1 and n2 have start and end positions (x1,y1) and (x2,y2) respectively then, n2 is child node of n1 if (x1<x2) and (y1>y2). If (x1<x2) and (y1<y2), then there is no such relation. In this case, nodes are present at the same hierarchical level inside the method. The level of hierarchy of nodes determines the position of adapter and composition connectors of X-MAN e.g., if a while loop node is child of a if statement node, then the Loop connector should be composed under the Guard connector of the if statement. At the end of this algorithm, all methods of a cluster will be stored in stack node list along with the hierarchy of child nodes in descending order (SortNodes() in algorithm 4).

Algorithm 5 takes two-dimensional stack list of each cluster and searches for the position of every invocation node in every method of the cluster. Starting from the first method Mi, all the nodes between invocation node position and start node of that method will be checked for statements (after this, start node will be replaced by the invocation node and invocation node will be replaced by the new method invocation node, if another method call exists in the body). If a child node is just a statement, it will be placed in the current component (after trimming the Java object related calls). If a child node is a control statement (if, if-else, switch, loops etc.), each type of statement will be parsed via an algorithm (line 19-33). If there is no control structure between the start node of a method and the invocation node, the invoked method will be placed in the same computation unit as the method Mi (line 37-42).

---

2^8 is the AST node type value of standalone else statement in algorithm 4 i.e., this value is used to find out if there is an independent else statement exists after all the chains of if-else statements.
6.3. IMPLEMENTING THE CODE MAPPER

Algorithm 5 Components Extraction

Require: List<StackStructure> stack, int[] NoOfComponents = 0
for (i = 0; i < stack.size(); i++) do
    int index = 0
    bool flag = true, bool control = true
    for (j = 0; j < stack[i].length(); j++) do
        if (i == 0) then
            MethodDeclaration m = stack[i][j].getParentNodeByType(MethodDeclaration)
            NoOfComponents.add(1)
            CreateAtomicComponent("C".concat(NoOfComponents.size().toString()), m) ← current
        end if
        if stack[i][j].getType() == methodInvocation then
            if (flag == true) then
                index = j; flag = false
            end if
            Continue
        else if stack[i][j].getType() == methodInvocation then
            for (a = index; a < j; a++) do
                if stack[i][a].getType() == Statement.expression then
                    AddInCurrentComponent(stack[i][a].getType().getExpressions())
                else if stack[i][a].getType() == While.Statement then
                    control = false
                    Execute Algorithm 6 For while loop and resumes from here
                else if stack[i][a].getType() == For.Statement then
                    control = false
                    Execute Algorithm 6 For for loop and resumes from here
                else if stack[i][a].getType() == Switch.Statement then
                    control = false
                    Execute Algorithm 8 For switch and resumes from here
                else if stack[i][a].getType() == doWhile.Statement then
                    control = false
                    Execute Algorithm 7 For do-while and resumes from here
                else if stack[i][a].getType() == IF.Statement then
                    control = false
                    Execute Algorithm 8 For if and resumes from here
                end if
            end for
        end if
    end if
end for
if control == true then
    MethodDeclaration m1 = stack[i][j].getParentNodeByType(MethodDeclaration)
    if m.cluster == m1.cluster then
        AddInCurrentComponent(m)
    else
        AddInvocationParametersInCurrentComponentOutPut(m.getInvokedParameters)
    end if
end if
end for
end for
Algorithm 6 While and For Statements

if ($a == j - 1$) then
    MethodDeclaration m1=stack[i][a+1].getParentNodeByType(MethodDeclaration)
    if (m.cluster == m1.cluster) then
        String con=stack[i][a].getCondition().toString()
        Connector c = new LoopConnector(con)
        NoOfComponents.add(1)
        CreateAtomicComponent("C".concat(NoOfComponents.size())
                        .toString()), m1) ← current
        c.add("C".concat(NoOfComponents.size().toString())) ← current
        Connector c1= new SEQConnector ("C".concat(NoOfComponents.size() −
                        1.toString()), c.current) ← current
        stack[i][a].removeNode()
    else
        String con=stack[i][a].getCondition().toString()
        Connector c = new LoopConnector(con)
        c.add("C".concat(NoOfComponents.size().toString())) ← current
        stack[i][a].removeNode()
    end if
else
    String con=stack[i][a].getCondition().toString()
    Connector c = new LoopConnector(con)← current
    Connector c1= new SEQConnector ("C".concat(NoOfComponents.size())
                        .toString()), c) ← current
    stack[i][a].removeNode()
end if
Algorithm 7 do-While Statement

if \((a == j - 1)\) then
MethodDeclaration m1=stack\[i\][a+1].getParentNodeByType(MethodDeclaration)
if \((m.cluster == m1.cluster)\) then
String con=stack\[i\][a].getCondition().toString()
Connector c = new LoopConnector(con)
NoOfComponents.add(1)
CreateAtomicComponent("C".concat(NoOfComponents.size().toString()),m1) ← current
NoOfComponents.add(1)
CreateAtomicComponent("C".concat(NoOfComponents.size().toString()),m1) ← current
c.add("C".concat(NoOfComponents.size().toString())) ← current
Connector c1= new SEQConnector ("C".concat(NoOfComponents.size() - 1.toString()),c.current) ← current
Connector c2= new SEQConnector ("C".concat(NoOfComponents.size() - 2.toString()),c1) ← current
stack\[i\][a].removeNode()
else
String con=stack\[i\][a].getCondition().toString()
Connector c = new LoopConnector(con)
c.add("C".concat(NoOfComponents.size().toString())) ← current
Connector c1= new SEQConnector ("C".concat(NoOfComponents.size() - 1.toString()),c.current) ← current
end if
else
String con=stack\[i\][a].getCondition().toString()
Connector c = new LoopConnector(con)
NoOfComponents.add(1)
CreateAtomicComponent("C".concat(NoOfComponents.size().toString()),m1) ← current
Connector c1= new SEQConnector ("C".concat(NoOfComponents.size() .toString()),c) ← current
Connector c2= new SEQConnector ("C".concat(NoOfComponents.size() - 1.toString()),c1))
stack\[i\][a].removeNode()
end if
Algorithm 8 if, if-else and switch Statement

Require: flag=true

\( a = j - 1 \) then

MethodDeclaration m1=stack[i][a+1].getParentNodeByType(MethodDeclaration)

if (m.cluster == m1.cluster) then

String con=stack[i][a].getCondition.toString()

if stack[i][a].hasElsePresent || stack[i][a].hasStandAloneElse then

if flag == true then

Connector c = new SelectorConnector(con)

flag=false

end if

NumberOfComponents.add(1)

CreateAtomicComponent(\"C\".concat(NumberOfComponents.size().toString()), m1) ← current

c.add("C".concat(NumberOfComponents.size().toString())) ← current

Connector c1= new SEQConnector("C".concat(NumberOfComponents.size() - 1.toString()), c.current) //current points here if it was last else condition

stack[i][a].removeNode()

else

Connector c = new GuardConnector(con)

NumberOfComponents.add(1)

CreateAtomicComponent(\"C\".concat(NumberOfComponents.size().toString()), m1) ← current

c.add("C".concat(NumberOfComponents.size().toString())) ← current

Connector c1= new SEQConnector("C".concat(NumberOfComponents.size() - 1.toString()), c.current) ← current

stack[i][a].removeNode()

end if

else

String con= stack[i][a].getCondition.toString()

if stack[i][a].hasElsePresent || stack[i][a].hasStandAloneElse then

if flag == true then

Connector c = new SelectorConnector(con)

flag=false

end if

end if

else

//Same as Line 24 to 37 except c.add() because more control statements are expected.

end if
6.3. IMPLEMENTING THE CODE MAPPER

Algorithms 6, 7 and 8 are showing the implementation of the rule base for control structures\(^3\). These algorithms are actually part of the Algorithm 5 and only explained separately due to their complexity and length. In algorithm 6, line 1 checks whether the given loop condition has any more control structures after it or not. In case, it is the last control statement and the method being invoked belongs to the same cluster as its caller, the condition of the loop will be extracted and the current component will be sequenced with a newly created atomic component that will have the invoked method under the Loop connector (line 1-10). If there are more control statements after the given loop statement, the current component will be sequenced with the newly created Loop connector. If the method being invoked belongs to a different cluster then the Loop connector will be added to current component with stated conditions. The algorithm 7 is same as the 6, with a difference of extra sequenced atomic component composed with the component under loop (because do-while has to execute the method at least once).

Algorithm 8 covers the logic of switch, if and if-else statements. A standalone if statement needs a Guard whereas, 2-n if-else or switch statement require Selector. Again, the upcoming control statement can be the last one (line 1), invoked method can belong to different cluster (line 23) or there are more control statements after the current statement (line 39). Whatever the case is, the algorithm determines the needed connector (Guard or Selector) and sets the current position accordingly for the upcoming statements or invocation nodes. The sequencer connector takes two parameters; first parameter can be a component and the second parameter can be a component or another connector (to map upcoming node). The term current refers to the current index that will be connected with the next upcoming control statement or atomic component.

These eight algorithms sum up the whole logic behind implementation of the defined rule base i.e., Code Mapper\(^4\). The end result after all this processing is the Java code in atomic/composite X-MAN components notation. These components will be deposited to RX-MAN repository (Chapter 7).

\(^3\)current in these algorithms show the position of association for the next upcoming statement. If a component is added in a Loop connector, then the current position connects the next connector/component under the Loop.

\(^4\)Due to length and complexity, the only scenario that is missing from the algorithms 6 and 7 is the nested loops. In that case, two Loop connectors will be connected with a Selector to tackle the conditions in both outer and inner loop (Figure 5.52).
6.4 Summary

This chapter has presented the RX-MAN parser and algorithms that implement the rule base of Chapter 5. Based on their invoked nodes, all the methods of a code base are re-structured after indexing and clustered by the "Code Mapper". After clustering, the "Code Mapper" extracts atomic/composite components from each cluster and map them to X-MAN meta-model. Next chapter will introduce the RX-MAN tool and demonstrate the deposited components in the RX-MAN repository.
Chapter 7

Implementing the RX-MAN Tool

“You can mass-produce hardware; you cannot mass-produce software-you cannot mass-produce the human mind.”

— Michio Kaku, American Physicist

7.1 Introduction

This chapter describes the development of RX-MAN tool. The presented tool is an extension of the X-MAN tool [DCTL15]. The tool can reverse engineer components from Java API's, applications or libraries. The tool also facilitates the deposition, retrieval and code-independent composition of deposited components.

Section 7.2 presents a brief introduction of the tool. Section 7.3 discusses the technology stack used for implementation. Section 7.4 explains the working and interface of the tool and Section 7.5 demonstrates the tool using the brake control system to show the deposition, retrieval, composition and re-deposition of extracted components.
7.2 Tool Overview

The RX-MAN tool is designed by integrating our reverse engineering approach with a customised version of the X-MAN tool [DCTL15]. The tool is implemented using model-driven technologies like Eclipse Modelling Framework (EMF), Graphiti https://www.eclipse.org/graphiti and CDO (a framework with development time model repository) [Ste]. A user can extract, deposit and compose components using a click and drag interface. The tool is built as an eclipse plug-in and hence can be integrated with other plug-ins.

Our tool can: extract components from Java code (single application, SDK, library etc.), deposit the extracted component into repository, retrieve the deposited component, instantiate the retrieved components for system construction/integration, compose the retrieved components (form composite components, select/de-select services) and re-deposit the composed components for further reuse. Unlike other reverse engineering tools that extract components [ARA⁺09, CKK08, AAAC07], the novelty of our tool is based on the following two features: 1) re-composition of the extracted components after deposition. 2) reuse of the extracted components without changing the code manually at required places.

Like all other tools of reverse engineering, the presented tool also has some limitations in terms of the constructs of programming language i.e., Java. Few important points about working of tool in terms of Java constructs are as follows:

1. The RX-MAN tool can handle all Java constructs based on relational operators and can map all such conditions in the source code to adapter and composition connectors. These operators include $\equiv, \not\equiv, \geq, \leq, ||$ etc.
2. The tool does not deal with dynamic constructs e.g., polymorphism in Java. Static reverse engineering cannot deal with dynamic constructs.
3. The tool does not deal specifically with clone based or instantiation based constructs e.g., instance of command in Java.
4. The tool deals with all the constructs based on branching (e.g., if, if-else, switch etc.) and looping (e.g., while, doWhile, for etc.).
5. The tool does not consider abstract classes as a special case. The approach only needs concrete implementation of an abstract method to construct components.
6. The tool does not deal with exception based statements e.g., throw or catch constructs in Java.
7. The tool can deal with all the defined data types in Java.
8. The approach does not differentiate between composition, aggregation and association.

9. The quality of reverse engineered components is also based on quality of the source code to be reverse engineered. Poor programming practices can cause duplication of a method in more than one components e.g., if a public method is being called from multiple private methods of different classes, the public method needs to be added in computation units of all the private callers.

### 7.3 Implementing the Technology Stack

The technology stack used for implementing RX-MAN is as follows:

- **EMF**: EMF stands for Eclipse Modelling Framework\(^1\). The fundamental part of EMF is Ecore (meta-model of EMF) for creating models and EMF objects \(^2\).

![Diagram](image)

**Figure 7.1: RX-MAN: Technology Stack**

- **Xcore**: Xcore\(^3\) fulfils the gap between modelling and programming. It is an extension of Ecore and used for describing behaviour of the entities of defined model along with the behavioural aspects e.g., data type casting rules, relation

---

\(^1\)https://www.eclipse.org/modeling/emf/

\(^2\)http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html#details

\(^3\)https://wiki.eclipse.org/Xcore
between different data type entities etc. The X-MAN meta-model has been defined using EMF and Xcore.

- **Graphiti**: Graphiti is an eclipse based framework, used for developing graphical editors that can be edited. Graphiti is used to instantiate X-MAN meta-model and for creating an editor that can compose and design X-MAN components, based on the Xcore model. The process of creating editable editors is further facilitated by SPRAY (a domain specific language) that defines a language to create and manage Graphiti objects more easily.

- **CDO**: CDO stands for connected data objects and it is a distribution of EMF for storing the EMF based models. It is a centralised repository that supports development (design phase in a component life cycle, Section 3.3) and run time persistent framework (for deployment of components).

- **AST Parser**: AST parser stands for abstract syntax tree parser provided by JDT (Java development Tools) to access and manipulate the Java code. For RX-MAN tool, a customisation of AST has been developed with a back end model that can store the re-structured nodes for mapping.

- **RX-MAN Plug-in**: The RX-MAN plug-in carries the whole implementation of reverse engineering approach as described in Chapter 5 and 6. The implemented plug-in is also integrated with CDO, Xcore and X-MAN meta-model to synchronise and integrate the reverse engineering with X-MAN tool.

Figure 7.1 is showing the technology stack used for developing the RX-MAN tool. In the Figure 7.1, red boxes are showing the back end development, teal boxes are showing interface and editor related frameworks, purple box is showing the repository and the yellow one is showing the developed approach as an eclipse plug-in.

### 7.3.1 Comparison With Previous X-MAN Tool

The RX-MAN tool is designed as an extension of previous X-MAN tool but there are several changes that take place to re-define many entities of X-MAN meta-model in order to integrate the reverse engineering. Some of the most prominent changes are as follows:

- Changes in the semantics of Loop to enable the multiple data types for iterator and termination condition (in a source code a loop can be terminated based on a condition of any data type). Along with these changes, iteration interval is also

---

4. [https://www.eclipse.org/graphiti/](https://www.eclipse.org/graphiti/)
5. [https://www.eclipse.org/cdo/](https://www.eclipse.org/cdo/)
added so an iterator can be iterated over a defined interval. The previous tool version only had an integer iterator, without termination condition and with a fixed interval of iteration i.e., \( i = i + 1 \).

- Changes in the semantics of \emph{Guard} and \emph{Selector} to add the conditional invocation, based on any data type.
- Addition of \emph{Exposed Methods} in the meta-model to define the semantics of composition for composing 2-n components with selected methods.
- Integration of RX-MAN plug-in with CDO repository in order to synchronise the centralised repository for both extraction and designing of components.
- Integration of back end model to map the re-structured RX-MAN nodes to X-MAN meta-model for depositing the components.

7.4 RX-MAN Tool: Features

The features of RX-MAN tool can be classified into three main categories:

1) Design phase composition of components i.e., construct components from scratch and deposit them into CDO repository.
2) Reverse engineering of components from Java legacy code by using the proposed approach.
3) Deployment phase composition of components i.e., retrieval of reverse engineered or constructed components from CDO for further composition, system construction or reuse.

7.4.1 Design Phase Composition

The design phase composition of X-MAN components includes the construction and deposition of atomic X-MAN components. The interface of design phase composition includes a computation unit, exposed methods and data elements. At least one exposed method with explicitly defined inputs/outputs needs to be specified to deposit a semantically valid X-MAN atomic component. Figure 7.2 is showing the option to construct and deposit atomic components in the tool. Figure 7.3 is showing an atomic component with one method in its computation unit and as the exposed method. The property dialogue-box in the figure is used for adding names and comments in computation units and exposed methods, and specifying input/output data elements along with their data types. On selecting the code generation option from Figure 7.3, the tool will generate
the code skeleton in which, implementation can be added. Such a skeleton will consist of an interface (exposed methods) and their implementation (computation unit). Figure 7.4 is showing the generated code from atomic component of Figure 7.3 (the body of method in the implementation was implemented manually). Before depositing this component, more exposed methods can be added and code skeleton can be generated again to make it compatible with new additions. Once the implementation logic is added in the skeleton and component is deposited into CDO, it can only be retrieved for deployment (in deployment phase for composition) and one cannot see the inner implementation of the component i.e., it will become a black box with specified functionality by provided exposed methods.

7.4.2 Reverse Engineering of Components

The designed approach takes all the Java files at a specified path to extract components. The process is automated and just needs two steps from the tool i.e., generation of restructured RX-MAN nodes (Section 6.2.1) and extraction of components. Extraction
7.4. RX-MAN TOOL: FEATURES

Figure 7.3: RX-MAN: Design Phase Composition

```java
package com.A.scenario;

// Java interface for exposing methods of Component A

public interface A {
    public Boolean SensorMonitor(Integer SensorData);
}
```

```java
// This is Java implementation of the atomic component A
package com.A.scenario;
import java.util.*;

public class AImpl implements A {
    public AImpl() {
        // Constructor of atomic component A
    }

    public Boolean SensorMonitor(Integer SensorData) {
        // TODO: auto-generated code, to be implemented by developers under this comment
        Boolean flag=true;
        if (SensorData<0)
            flag=false;
        return flag;
    }
}
```

Figure 7.4: RX-MAN: Code Generation of an Atomic Component

will also deposit the extracted components into CDO repository. Figure 7.5 and 7.7 are showing the nodes' generation and components' extraction interfaces respectively.
7.4.3 Deployment Phase Composition

The deployment phase composition of RX-MAN tool uses the reverse engineered or deposited component from CDO repository to reuse or compose them further. Two or more than two components can be retrieved with selected exposed methods to compose via composition/adapter connectors. Figure 7.6 is showing the option of composite design to compose components in deployment phase. The interface and usage of deployment phase composition will be demonstrated using an example in the succeeding section.

7.5 Implementing the Brake Control System by RX-MAN Tool

This section will use the brake control system example from Chapter 5 to demonstrate the RX-MAN tool.

The Figure 7.7 is showing the RX-MAN tool and components' extraction option for the brake control system. The given option will generate components from all the Java files in the specified path. The Figure 7.8 is showing the deposited components of brake control system in the CDO repository (BCS in repository explorer). In
7.5. IMPLEMENTING THE BRAKE CONTROL SYSTEM BY RX-MAN TOOL

Figure 7.6: RX-MAN: Selection of Atomic Design to Construct and Deposit Atomic Components

Figure 7.8, the first composite component is instantiated as SpeedControl with methods collisionTimeCalculator() and speedMonitoringValue() selected to be exposed for composition (tool representation of the first composite component of Figure 5.57). The Figure is also showing the retrieval option for the other component from the repository. The third component in the BCS is the utility component (as discussed in Chapter 5). The X-MAN components in the tool are like a black box. One does not need to know about the composition of atomic components inside a composite, but only needs to use
the provided methods along with their data elements and composition connectors to compose components together.

Figure 7.9 is showing the composition of extracted components of brake control system by Sequencer. In the given figure, the second composite component is instantiated as BrakeControl with methods collisionParametersActivation(), BrakeSystemActivation() and TimeTriggerValue() to be exposed for composition. The box in the centre is a reference box for setting the order of execution of exposed methods in a composition. In the given example, the Sequencer will invoke the SpeedControl component as this route has the lower path value i.e., 0. According to reference order of the exposed methods of this component, the method (collisionTimeCalculator()) will be executed first (as this method is at the top of reference box). After execution of collisionTimeCalculator() and assignment of collisionEstimate to speedMonitoringValue(), the method speedMonitoringValue() will be executed. As there are not any exposed methods left for this component, the Sequencer will invoke the other composite component i.e., route 1. According to reference values of the exposed methods in the box, method speedMonitoringValue() will be executed, followed by either BrakeSystemActivation() or looping of TimeTriggerValue (as there is a Selector and an Loop connector between the atomic components of the component BrakeControl) as shown in the Figure 5.57.

The inputs and outputs in the reference box are the data elements and they can be assigned with an order reference to provide data values after execution of a specific method e.g., reference value of method collisionTimeCalculator() is zero as it is the first method, selected for execution in component SpeedControl. After its execution,
all the data elements in the reference box with value 1 will transmit their input/output values (e.g., value of collisionEstimate should be provided to method speedMonitoringValue()). Such ordering of exposed methods and data elements execute all the methods, exposed for a composition in a defined order while providing the data inputs/outputs to the upcoming method to be executed. Exposed methods will always be executed in top down order, defined in the reference box whereas, order of data elements can be set by using the property window of the tool. The order value of data elements can range from 0-n where n is a positive number.
Figure 7.10 is showing the retrieved component after deposition (it is re-deposition of already deposited components after composition) of composite component of Figure 7.9. The X-MAN repository is now showing four components in BCS and one of them is \textit{BCS\_System}. On retrieval, the \textit{BCS\_System} is showing all the data elements of the composed components inside the new composite component. Now, the brake control system can be composed further with more components and can be deposited/reused as many times as a specific integration/reuse demands. Hence, one can achieve code independent composition after deposition i.e., re-composition by using composition connectors\textsuperscript{6}. The component \textit{BCS\_System} has one method in it (Brake Control System in Figure 7.10) that abstracts the exposed methods, used in the composition of Figure 7.9. The composition of components after retrieval from the repository is a deployment phase composition. In the whole process of composition and re-deposition, one does not need to change any code manually, but only needs the source code familiarity to achieve semantically correct composition. Deposition after each composition results in a composite component that has the enclosed functionality of all the previous compositions and can be composed further code-independently.

\textsuperscript{6}The red error signs on inputs/outputs of Figure 7.10 are showing that these data elements need an order value and data channel for further composition.
Figure 7.9: RX-MAN: Composition of Brake Control System
Figure 7.10: RX-MAN: Retrieval of Composed Brake Control System
Summary

This chapter explained the technology stack for the implementation of RX-MAN tool and presented the difference between previous version and the developed version of X-MAN tool for this research. The pros and cons of the tool are also discussed in terms of Java constructs. The example of brake control system, defined in the Chapter 5 was used to demonstrate the working of tool by showing the deposition, retrieval, composition and re-deposition of the extracted components from brake control system.
Chapter 8

Evaluation of RX-MAN

“Friend to Marx: Life is difficult!
Marx to Friend: Compared to what?”

— Groucho Marx

8.1 Introduction

This chapter presents the validation and evaluation of RX-MAN. The overall evaluation can be divided into following four parts:

1. **Empirical Validation and Evaluation.** Five legacy code bases will be used to reverse engineer X-MAN components. The validation will be conducted by showing the extracted components in CDO repository and the empirical evaluation will be demonstrated by statistics, a potential case of composition of 2-n components from each code base, re-deposition of the composed components after composition and instantiation of the re-deposited component in RX-MAN tool. Due to complex and large selected code bases, it is impossible to fully narrate the semantics of the source code. However, one possible case of composition will be identified from each code base to show the compliance with research aims and objectives.

2. **Evaluation as a Reverse Engineering approach using Gannod and Cheng’s Framework [GC99]:** The selected framework of evaluation is not only confined to component directed reverse engineering but covers the overall domain of software reverse engineering and recovery techniques. There is no such framework that exists specifically for component directed reverse engineering.
3. **Comparison with Component Directed Reverse Engineering Approaches:** This comparison will compare the proposed approach with other component directed reverse engineering approaches that produce explicitly defined components. The focus of this comparison will be on re-usability and code-independent composition.

4. **Tool Evaluation by Bellay and Gall's Capability Assessment.** This comparison revolves around the tools of reverse engineering and will use the Bellay and Gall's [BG98] defined parameters to compare the proposed tool with other relevant tools that offer component directed reverse engineering.

Hence, the overall evaluation covers code bases to demonstrate the results, Gannod and Cheng's framework to demonstrate the pros and cons of relevant approaches, a comparison based on the re-usability and code-independent composition, and Bellay and Gall's capability assessment to compare relevant tools of reverse engineering. Figure 8.1 is showing the relevant entities of the research (red boxes) related to this chapter.
8.2 Evaluating RX-MAN By Empirical Validation and Evaluation

To empirically validate and evaluate the approach, five code bases have been chosen. Few of the selected code bases are taken from Qualitas Corpus [TAD+10]. The Qualitas Corpus is a curated collection of Java code for empirical studies. The succeeding sections will explain each code base, its source, size, components’ extraction, deposition, code-independent composition and re-deposition in RX-MAN tool.

8.2.1 Code Base 1: Draw.io

*Draw.io* [https://github.com/jgraph/drawio] is an online graphical tool that offers a wide variety of options to construct diagrams like flow charts, sequence diagrams, circuits, class diagrams etc. It uses *mxGraph* library as a base of its functionality. Table 8.1 and 8.2 are showing the results of RX-MAN based on package and class level clustering respectively.

According to Table 8.1, there were 101 Java classes in the source code of *Draw.io*

<table>
<thead>
<tr>
<th>Package level Clustering</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Classes</td>
<td>101</td>
</tr>
<tr>
<td>Total Components</td>
<td>21</td>
</tr>
<tr>
<td>Non-Utility Components</td>
<td>13</td>
</tr>
<tr>
<td>Utility Components</td>
<td>8</td>
</tr>
<tr>
<td>Abstraction Ratio</td>
<td>7.7%</td>
</tr>
<tr>
<td>Total Methods</td>
<td>843</td>
</tr>
<tr>
<td>Non-Utility Methods</td>
<td>513</td>
</tr>
<tr>
<td>Utility Methods</td>
<td>334</td>
</tr>
<tr>
<td>% Code in Non-Utility Methods</td>
<td>60%</td>
</tr>
<tr>
<td>% Code in Utility Methods</td>
<td>39%</td>
</tr>
<tr>
<td>Execution Time</td>
<td>7 Sec</td>
</tr>
</tbody>
</table>

Table 8.1: Reverse Engineering of Draw.io by Package Level Clustering

and the proposed approach extracted 21 components from these 101 classes by using the package-level clustering. Out of these 21 components, 13 were non-utility (i.e., atomic or composite X-MAN components that offer a specific functionality of a source code and can also be reused) and 8 were utility components (i.e., 8 can provide re-usability to use any utility method in them for composition). The abstraction ratio between classes and non-utility components was 7.7% i.e., on average, one non-utility
component has code equal to almost 7.7 Java classes in the original source code. The

**Class level Clustering**

<table>
<thead>
<tr>
<th>Component Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Classes</td>
<td>101</td>
</tr>
<tr>
<td>Total Components</td>
<td>139</td>
</tr>
<tr>
<td>Non-Utility Components</td>
<td>91</td>
</tr>
<tr>
<td>Utility Components</td>
<td>48</td>
</tr>
<tr>
<td>Abstraction Ratio</td>
<td>1.10%</td>
</tr>
<tr>
<td>Execution Time</td>
<td>17 Sec</td>
</tr>
</tbody>
</table>

**Draw.io**

<table>
<thead>
<tr>
<th>Abstraction Ratio</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.10%</td>
<td>17 Sec</td>
</tr>
</tbody>
</table>

Table 8.2: Reverse Engineering of Draw.io by Class Level Clustering

total methods in the Java source code were 843. Out of these 843 methods, 513 were allocated to non-utility components and 334 were allocated to utility components. These numbers i.e., 513 and 334 made up 61% and 39% of the source code respectively. It means 39% of the original source code can be reused with newly constructed components independently (as utility methods do not call any other method). Whereas, 61% of the code fulfils functionality of the original source code and requires the familiarisation with the source code to achieve semantically correct composition to reuse the components. The overall processing time for the extraction and deposition of components was 7 seconds.

According to Table 8.2, by using the class-level clustering, 91 non-utility and 48 utility components were extracted i.e., the proposed approach extracted 139 components (number of components are greater than the number of classes due to presence of utility components with utility methods. Number of methods in extracted components can also be greater than the number of methods in the original source code because one method can be allocated to utility and non-utility component both, and one method can also be allocated to more than one components, based on the nature of interaction with its callers). The processing time for this granularity level was 17 seconds.

Figure 8.2 is showing a case of composition (using Sequencer) by instantiating 5 extracted components of draw.io. The validation diagnostics of Xcore model of X-MAN only allows the deposition of components if they are compliant to the semantics of the meta-model. Therefore, semantically wrong components that do not

---

1All cases of composition are based on class-level clustering. Components with package-level clustering are much bigger and require a lot of component’s instantiations to show a semantically correct composition. Components extracted by class-level clustering are less granular and easier to show cases of compositions.
fulfil the compliance to the defined meta-model cannot be deposited. The instantiated components are deployed with following methods: component 1 with exposed method `addEdgeData`, component 2 with exposed method `setDataKey`, component 3 with exposed methods `setText` and `setStyle`, component 4 with exposed method `setShapeDataShapeEdge` and component 5 with exposed method `setEdgeData`. According to the source code, the composition of Figure 8.2 adds edge data and set its data key, text, style, edge data and the edge of the shape. The `EdgeService` reference value table at the left is showing the order of data elements and exposed methods (as described in the Chapter 7).

Figure 8.3 is showing the extracted 21 (at package-level) and 139 (at class-level) components of Draw.io. The class-level extracted components are 140 now because one composite component (EdgeComposite) has been deposited (shown in the left panel). The retrieval window in the figure is showing that the deposited composite component (EdgeComposite) can be retrieved with its exposed method which abstracts the functionality of exposed methods that were used in the composition of Figure 8.2. This composite component can be involved in further composition (like Figure 8.2).
i.e., re-composition after re-deposition. The whole process of composition did not involve any code-level configuration or restriction of binding the services like ADLs.

Figure 8.3: Instantiation of Composite Component after Composition/Deposition of Extracted Components
CHAPTER 8. EVALUATION OF RX-MAN

8.2.2 Code Base 2: EverNote SDK

*EverNote* is a popular multi platform application, used for scheduling and taking notes. The selected source code {https://github.com/evernote/evernote-sdk-java} is the Java SDK of EverNote that can be used with a Java application to embed the EverNote application. The SDK includes wrapper code to access cloud API, client authentication code and EverNote integration code. By extracting components from this SDK, one can integrate, expand and reuse the SDK in component-based development. Table 8.3 and 8.4 are showing the results of RX-MAN based on package and class level clustering respectively.

According to Table 8.3, there were 106 Java classes in the source code of *EverNote SDK* and the proposed approach extracted 18 components from these 106 classes by using the package-level clustering. Out of these 18 components, 9 were non-utility (i.e., atomic or composite X-MAN components that offer a specific functionality of a source code and can also be reused) and 9 were utility components (i.e., 9 can provide re-usability to use any utility method in them for composition). The abstraction ratio between classes and non-utility components was 11.77% i.e., on average, one non-utility component has code equal to almost 11.77 Java classes in the original source code. The total methods in the Java source code were 3221. Out of these 3221 methods, 1471 were allocated to non-utility components and 1750 were allocated to utility components. These numbers i.e., 1471 and 1750 made up 45% and 55% of the

<table>
<thead>
<tr>
<th>Package level Clustering</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Classes</td>
<td>106</td>
</tr>
<tr>
<td>Total Components</td>
<td>18</td>
</tr>
<tr>
<td>Non-Utility Components</td>
<td>9</td>
</tr>
<tr>
<td>Utility Components</td>
<td>9</td>
</tr>
<tr>
<td>Abstraction Ratio</td>
<td>11.77%</td>
</tr>
<tr>
<td>Total Methods</td>
<td>3221</td>
</tr>
<tr>
<td>Non-Utility Methods</td>
<td>1471</td>
</tr>
<tr>
<td>Utility Methods</td>
<td>1750</td>
</tr>
<tr>
<td>% Code in Non-Utility Methods</td>
<td>45%</td>
</tr>
<tr>
<td>% Code in Utility Methods</td>
<td>55%</td>
</tr>
<tr>
<td>Execution Time</td>
<td>7 min, 11 sec</td>
</tr>
</tbody>
</table>

Table 8.3: Reverse Engineering of EverNote SDK by Package Level Clustering

Note SDK and the proposed approach extracted 18 components from these 106 classes by using the package-level clustering. Out of these 18 components, 9 were non-utility (i.e., atomic or composite X-MAN components that offer a specific functionality of a source code and can also be reused) and 9 were utility components (i.e., 9 can provide re-usability to use any utility method in them for composition). The abstraction ratio between classes and non-utility components was 11.77% i.e., on average, one non-utility component has code equal to almost 11.77 Java classes in the original source code. The total methods in the Java source code were 3221. Out of these 3221 methods, 1471 were allocated to non-utility components and 1750 were allocated to utility components. These numbers i.e., 1471 and 1750 made up 45% and 55% of the

---

2These statistics also include the interfaces and abstract methods. Methods in computation units are less than these numbers because a computation unit does not need/store interfaces/abstract methods during extraction.
8.2. EVALUATING RX-MAN BY EMPIRICAL VALIDATION AND EVALUATION

### Table 8.4: Reverse Engineering of EverNote SDK by Class Level Clustering

<table>
<thead>
<tr>
<th>Class level Clustering</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Classes</td>
<td>106</td>
</tr>
<tr>
<td>Total Components</td>
<td>154</td>
</tr>
<tr>
<td>Non-Utility Components</td>
<td>80</td>
</tr>
<tr>
<td>Utility Components</td>
<td>74</td>
</tr>
<tr>
<td>Abstraction Ratio</td>
<td>1.3%</td>
</tr>
<tr>
<td>Execution Time</td>
<td>5 min</td>
</tr>
</tbody>
</table>

Source code respectively. It means 55% of the original source code can be reused with newly constructed components independently. Whereas, 45% of the code fulfills functionality of the original source code and requires the familiarisation with the source code to achieve semantically correct composition to reuse the components. The overall processing time for the extraction and deposition of components was 7 minutes, 11 seconds. The relativity higher processing time as compared to Draw.io indicates the presence of 3221 methods versus 843 methods of Draw.io. The interaction and collaboration of methods is also responsible for the processing time as more interactions leads to complex RX-MAN code re-structuring that requires more time for indexing and clustering of methods.

According to Table 8.4, by using the class-level clustering, 80 non-utility and 74 utility components were extracted i.e., the proposed approach extracted 154 components. The processing time for this granularity level was 5 minutes.

Figure 8.4 is showing a case of composition by instantiating 4 extracted components of EverNote SDK (one component i.e., `GetClient` is instantiated twice). According to semantics of the source code, these composed components create a note, get the client, authenticate the client before saving the note, get the client again after authentication and then store the note. All the components are instantiated with one exposed method each (as needed for the specified functionality). The component with exposed method `getClient` is instantiated twice as the source code is using this method twice, for authenticating the client and for storing the note. The stated composition is deposited as `EverNoteComposition` into CDO repository.

Figure 8.5 is showing the extracted 18 (at package-level) and 154 (at class-level) components of EverNote SDK. The class-level extracted components are 155 now because one composite component (`EverNoteComposition`) has been deposited (shown in the left panel). The CDO window in the figure is showing that the deposited composite component (`EverNoteComposition`) can be retrieved with its exposed method which
CHAPTER 8. EVALUATION OF RX-MAN

Figure 8.4: Example of Code-Independent Composition in EverNote SDK

abstracts the functionality of exposed methods that were used in the composition of Figure 8.4.
8.2. EVALUATING RX-MAN BY EMPIRICAL VALIDATION AND EVALUATION

Figure 8.5: Instantiation of Composite Component after Composition/Deposition of Extracted Components
CHAPTER 8. EVALUATION OF RX-MAN

8.2.3 Code Base 3: JabRef

JabRef [https://github.com/JabRef/jabref] is an open source citation and reference management tool, majorly used by the academic researchers as a citation manager. The source code is open source and has been used by hundreds of developers according to statistics of its releases and commits. Extraction of components from JabRef can provide code-independent composition and further development of the JabRef using component-based approach. Table 8.5 and 8.6 are showing the results of RX-MAN based on package and class level clustering respectively.

According to Table 8.5, there were 935 Java classes in the source code of JabRef and the proposed approach extracted 205 components from these 935 classes. Out of these 205 components, 110 were non-utility (i.e., atomic or composite X-MAN components that offer a specific functionality of a source code and can also be reused) and 95 were utility components (i.e., 95 can provide re-usability to use any utility method in them for composition). The abstraction ratio between classes and non-utility components was 8.5% i.e., on average, one non-utility component has code equal to almost 8.5 Java classes in the original source code. The total methods in the Java source code were 5676. Out of these 5676 methods, 4646 were allocated to non-utility components and 1028 were allocated to utility components. These numbers i.e., 4646 and 1030 made up 82% and 18% of the source code respectively. It means 18% of the original source code can be reused with newly constructed components independently. Whereas, 82% of the code fulfils functionality of the original source code and requires the familiarisation with the source code to achieve semantically correct composition to reuse the components. The overall processing time for the extraction and deposition of

<table>
<thead>
<tr>
<th>Package level Clustering</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Classes</td>
<td>935</td>
</tr>
<tr>
<td>Total Components</td>
<td>205</td>
</tr>
<tr>
<td>Non-Utility Components</td>
<td>110</td>
</tr>
<tr>
<td>Utility Components</td>
<td>95</td>
</tr>
<tr>
<td>Abstraction Ratio</td>
<td>8.5%</td>
</tr>
<tr>
<td>Total Methods</td>
<td>5676</td>
</tr>
<tr>
<td>Non-Utility Methods</td>
<td>4646</td>
</tr>
<tr>
<td>Utility Methods</td>
<td>1030</td>
</tr>
<tr>
<td>% Code in Non-Utility Methods</td>
<td>82%</td>
</tr>
<tr>
<td>% Code in Utility Methods</td>
<td>18%</td>
</tr>
<tr>
<td>Execution Time</td>
<td>4 min, 26 sec</td>
</tr>
</tbody>
</table>

Table 8.5: Reverse Engineering of JabRef by Package Level Clustering
8.2. EVALUATING RX-MAN BY EMPIRICAL VALIDATION AND EVALUATION

Class level Clustering

<table>
<thead>
<tr>
<th>Category</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Classes</td>
<td>935</td>
</tr>
<tr>
<td>Total Components</td>
<td>1232</td>
</tr>
<tr>
<td>Non-Utility Components</td>
<td>801</td>
</tr>
<tr>
<td>Utility Components</td>
<td>431</td>
</tr>
</tbody>
</table>

**Abstraction Ratio:** 1.16%

**Execution Time:** 3 min

Table 8.6: Reverse Engineering of JabRef by Class Level Clustering

components was 4 minutes, 26 seconds.

According to Table 8.6, by using the class-level clustering, 801 non-utility and 431 utility components were extracted i.e., the proposed approach extracted 1232 components. The processing time for this granularity level was 3 minutes.

Figure 8.6 is showing a case of composition by instantiating two components. The first component is instantiated with three exposed methods whereas, the second one is instantiated with one exposed method. The stated composition also involves a **Guard** that has an input condition, based on which it will invoke the second component. The reference-order area in the composition is showing the order of execution of data elements and exposed methods that are set according to functionality and invocation of methods in the source code. The composition in the figure is taking inputs from an
outer source (or from another component if it yields the required parameters) and creates a custom format of references in JabRef. After reading preferences, if the required format already exists, the exposed method `createFormat` will be invoked and the `Guard` will not invoke the component `GetPreferences`. Otherwise, a key is generated with a conditional invocation of component `GetPreferences` after `createFormat` to create and store a new key against the newly defined format. The left panel of the tool in Figure 8.6 is showing the extracted and deposited components of JabRef in RX-MAN tool.
8.2.4 Code Base 4: JGraph5

Jgraph5 is another legacy code base (library) that is used for swing compliant implementation of graph components in Java. It is used for specifying objects and the relations between them and represent the relation in swing UI (user interface). This legacy code base is also managed by the Qualitus Corpus [http://qualitascorpus.com/docs/catalogue/20130901/index.html]. Table 8.7 and 8.8 are showing the results of RX-MAN based on package and class level clustering respectively.

According to Table 8.7, there were 171 Java classes in the source code of JGraph5 and the proposed approach extracted 52 components from these 171 classes. Out of these 52 components, 26 were non-utility (i.e., atomic or composite X-MAN components that offer a specific functionality of a source code and can also be reused) and 26 were utility components (i.e., 26 can provide re-usability to use any utility method in them for composition). The abstraction ratio between classes and non-utility components was 6.5% i.e., on average, one non-utility component has code equal to almost 6.5 Java classes in the original source code. The total methods in the Java source code were 2931. Out of these 2931 methods, 2094 were allocated to non-utility components and 836 were allocated to utility components. These numbers i.e., 2094 and 837 made up 72% and 28% of the source code respectively. It means 28% of the original source code can be reused with newly constructed components independently. Whereas, 72% of the code fulfils functionality of the original source code and requires the familiarisation with the source code to achieve semantically correct composition to reuse the components. The overall processing time for the extraction and deposition of components was 42 seconds.

<table>
<thead>
<tr>
<th>Package level Clustering</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Classes</td>
<td>171</td>
</tr>
<tr>
<td>Total Components</td>
<td>52</td>
</tr>
<tr>
<td>Non-Utility Components</td>
<td>26</td>
</tr>
<tr>
<td>Utility Components</td>
<td>26</td>
</tr>
<tr>
<td>Abstraction Ratio</td>
<td>6.5%</td>
</tr>
<tr>
<td>Total Methods</td>
<td>2931</td>
</tr>
<tr>
<td>Non-Utility Methods</td>
<td>2094</td>
</tr>
<tr>
<td>Utility Methods</td>
<td>837</td>
</tr>
<tr>
<td>% Code in Non-Utility Methods</td>
<td>72%</td>
</tr>
<tr>
<td>% Code in Utility Methods</td>
<td>28%</td>
</tr>
<tr>
<td>Execution Time</td>
<td>42 sec</td>
</tr>
</tbody>
</table>

Table 8.7: Reverse Engineering of JGraph5 by Package Level Clustering
CHAPTER 8. EVALUATION OF RX-MAN

Class level Clustering
Java Classes 935
Total Components 1232
Non-Utility Components 801
Utility Components 431
Abstraction Ratio 1.16%
Execution Time 3 min

Table 8.8: Reverse Engineering of JGraph5 by Class Level Clustering

According to Table 8.8, by using the class-level clustering, 140 non-utility and 126 utility components were extracted i.e., the proposed approach extracted 266 components. The processing time for this granularity level was 31 seconds.

Figure 8.7 is showing a simple case of composition by instantiating two components. The presented composition requires a document source from a data source or from another component. The component mxCodex is composite that has a Guard inside it. Depending on the value of DocSource the component will create a document by sequencing component CreateDocument with exposed method CreateDocument. The component CreateDocument will either create the document (if the document source
8.2. **EVALUATING RX-MAN BY EMPIRICAL VALIDATION AND EVALUATION**

is null) that holds the XML DOM value of the graph or does not execute the creation of document if the provided value states that provided document source is not null.

Figure 8.8 is showing the deposited composed composite component of Figure

![Graph showing deposited composite component](image)

Figure 8.8: Instantiation of Composite Component after Composition/Deposition of Extracted Components

8.7. The repository in the figure (on left hand side) is also showing the deposited component `jGraph5Composition` i.e., total JGraph components in the repository are 267 after depositing the composed composite component.
8.2.5 Code Base 5: TeamMates

TeamMates \{https://github.com/TEAMMATES/teammates\} is an online tool for managing peer evaluation and feedbacks in academia. It is a cloud based service that is being used by many universities. The selected code base is a standalone Java project of TeamMates that can be further developed to meet the individual's needs. Extraction of X-MAN components from this code base can facilitate the future development of TeamMates in component-based paradigm and extracted components can also be reused via code-independent composition. Table 8.9 and 8.10 are showing the results of RX-MAN based on package and class level clustering respectively.

According to Table 8.9, there were 815 Java classes in the source code of TeamMates and the proposed approach extracted 64 components from these 815 classes. Out of these 64 components, 34 were non-utility (i.e., atomic or composite X-MAN components that offer a specific functionality of a source code and can also be reused) and 30 were utility components (i.e., 30 can provide re-usability to use any utility method in them for composition). The abstraction ratio between classes and non-utility components was 24% i.e., on average, one non-utility component has code equal to 24 Java classes in the original source code. The total methods in the Java source code were 6871. Out of these 6871 methods, 5686 were allocated to non-utility components and 1185 were allocated to utility components. These numbers i.e., 5686 and 1185 made up 82.75% and 17.24% of the source code respectively. It means 17.24% of the original source code can be reused with newly constructed components independently. Whereas, 82.75% of the code fulfills functionality of the original source code and requires the familiarisation with the source code to achieve semantically correct

<table>
<thead>
<tr>
<th>Package level Clustering</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Classes</td>
<td>815</td>
</tr>
<tr>
<td>Total Components</td>
<td>64</td>
</tr>
<tr>
<td>Non-Utility Components</td>
<td>34</td>
</tr>
<tr>
<td>Utility Components</td>
<td>30</td>
</tr>
<tr>
<td>Abstraction Ratio</td>
<td>24%</td>
</tr>
<tr>
<td>Total Methods</td>
<td>6871</td>
</tr>
<tr>
<td>Non-Utility Methods</td>
<td>5686</td>
</tr>
<tr>
<td>Utility Methods</td>
<td>1185</td>
</tr>
<tr>
<td>% Code in Non-Utility Methods</td>
<td>82.75%</td>
</tr>
<tr>
<td>% Code in Utility Methods</td>
<td>17.24%</td>
</tr>
<tr>
<td>Execution Time</td>
<td>4 min, 28 sec</td>
</tr>
</tbody>
</table>

Table 8.9: Reverse Engineering of TeamMates by Package Level Clustering
8.2. EVALUATING RX-MAN BY EMPIRICAL VALIDATION AND EVALUATION

### Class level Clustering

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Classes</td>
<td>815</td>
</tr>
<tr>
<td>Total Components</td>
<td>1121</td>
</tr>
<tr>
<td>Non-Utility Components</td>
<td>725</td>
</tr>
<tr>
<td>Utility Components</td>
<td>396</td>
</tr>
<tr>
<td>Abstraction Ratio</td>
<td>1.12%</td>
</tr>
<tr>
<td>Execution Time</td>
<td>3 min, 8 sec</td>
</tr>
</tbody>
</table>

Table 8.10: Reverse Engineering of TeamMates by Class Level Clustering

According to Table 8.10, by using the class-level clustering, 725 non-utility and 396 utility components were extracted i.e., the proposed approach extracted 1121 components. The processing time for this granularity level was 3 minutes 8 seconds.

Figure 8.9 is showing a case of composition by instantiating three components. The presented composition takes a course id (from a data source or from another component) and finds the course institute based on the instructor of that course, associated to it via course id. The component GetCourseInstitute is composite and have Loop and Selector in it. All the composites are like a black box for the tool user therefore, one composition to reuse the components. The overall processing time for the extraction and deposition of components was 4 minutes, 28 seconds.
Figure 8.10: Instantiation of Composite Component after Composition/Deposition of Extracted Components

does not need to know about the composition of a component but only needs to know the order of execution of exposed methods and data elements in order to compose a semantically valid composition.

Figure 8.10 is showing the deposited composite component of Figure 8.9 in CDO repository and its instantiation.
8.2. EVALUATING RX-MAN BY EMPIRICAL VALIDATION AND EVALUATION

8.2.6 Discussion

Table 8.11 summarises the statistics of all five code bases by using the package-level clustering. The selected code bases include library (i.e., JGraph5), SDK (i.e., EverNote SDK) and applications (i.e., TeamMates, Draw.io and JabRef). The primary purpose of extracting such statistics is to show that our approach is successful in confining the size of the extracted components with higher abstraction yet manageable granularity. The percentage of code, allocated to utility components shows the extent of re-usability i.e., these utility methods can be reused in any required composition without changing their code whereas, the percentage of code that is allocated to non-utility components preserves the functionality of the original source code. However, one needs to understand a code base in order to achieve meaningful composition by setting the appropriate order of data elements and exposed methods of a component. Code-independent re-usability is another factor in case of non-utility components.

Another important thing worth mentioning is the processing time. The processing time is dependent on the number and nature of methods’ interaction in the source code. JabRef had 935 classes in the code base and its processing time was 4 minutes, 26 seconds. On the other hand, Evernote SDK had 106 classes, way less than the JabRef but its processing time was 7 minutes, 11 seconds. X-MAN is a model for computation, not a model for resource allocation. We can study the resource and memory allocation for RX-MAN, considering the fact that components will be stored only once, but reused many times. Therefore, we need less memory overall.

As compared to package-level clustering, the class-level clustering is summarised in Table 8.12. The class-level granularity usually requires less computation time than the package-level because package-level mostly requires more complex RX-MAN nodes restructuring due to larger chain of invocations and dependencies. The class-level approach however, extracts components with very low granularity and therefore, provides smaller components that are easier to compose but do not provide a higher

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Draw.io</td>
<td>101</td>
<td>21</td>
<td>13</td>
<td>8</td>
<td>7.7%</td>
<td>843</td>
<td>513</td>
<td>324</td>
<td>60%</td>
<td>39%</td>
<td>7 sec</td>
</tr>
<tr>
<td>EverNote SDK</td>
<td>106</td>
<td>18</td>
<td>9</td>
<td>9</td>
<td>11.77%</td>
<td>322</td>
<td>147</td>
<td>1750</td>
<td>45%</td>
<td>55%</td>
<td>7 min, 11 sec</td>
</tr>
<tr>
<td>JabRef</td>
<td>935</td>
<td>205</td>
<td>110</td>
<td>95</td>
<td>8.5%</td>
<td>5676</td>
<td>4646</td>
<td>1030</td>
<td>82%</td>
<td>18%</td>
<td>4 min, 26 sec</td>
</tr>
<tr>
<td>JGraph5</td>
<td>171</td>
<td>52</td>
<td>26</td>
<td>26</td>
<td>6.5%</td>
<td>2931</td>
<td>2094</td>
<td>817</td>
<td>72%</td>
<td>28%</td>
<td>42 sec</td>
</tr>
<tr>
<td>TeamMates</td>
<td>815</td>
<td>64</td>
<td>34</td>
<td>30</td>
<td>24%</td>
<td>6871</td>
<td>5686</td>
<td>1185</td>
<td>82.75%</td>
<td>17.24%</td>
<td>4 min, 28 sec</td>
</tr>
</tbody>
</table>

Table 8.11: Evaluation Statistics by Using Package-Level Clustering
abstraction ratio by enclosing multiple classes. In all the composition examples, class-
level granularity was used to avoid instantiation of multiple components that is neces-
sary to preserve the semantics of a composition at package-level.

One of the biggest benefits of reverse engineering X-MAN components is the much
simpler semantics of composition for reuse. In most of the cases of compositions, one
only needs a *Sequencer* because all the control statements (e.g., *if-else, loops etc.*) have
already been mapped in the extracted composite components using composition con-
nectors. One only needs to know the order of execution of data elements and exposed
methods to obtain a semantically valid composition.

Table 8.12: Evaluation Statistics by Using Class-Level Clustering

<table>
<thead>
<tr>
<th>Code Base</th>
<th>Java Classes</th>
<th>Total Components</th>
<th>Non-Utility Components</th>
<th>Utility Components</th>
<th>Abstraction Ratio</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Draw.io</td>
<td>101</td>
<td>139</td>
<td>91</td>
<td>48</td>
<td>1.10%</td>
<td>17 sec</td>
</tr>
<tr>
<td>EverNote SDK</td>
<td>106</td>
<td>154</td>
<td>80</td>
<td>74</td>
<td>1.3%</td>
<td>5 min</td>
</tr>
<tr>
<td>JabRef</td>
<td>935</td>
<td>1232</td>
<td>801</td>
<td>431</td>
<td>1.16%</td>
<td>3 min</td>
</tr>
<tr>
<td>JGraph5</td>
<td>171</td>
<td>266</td>
<td>140</td>
<td>126</td>
<td>1.22%</td>
<td>31 sec</td>
</tr>
<tr>
<td>TeamMates</td>
<td>815</td>
<td>1121</td>
<td>725</td>
<td>396</td>
<td>1.12%</td>
<td>3 min, 8 sec</td>
</tr>
</tbody>
</table>

showed that the proposed approach does not need any code-level configuration in or-
der to reuse the components. Re-deposition of the composed components also showed
that the composed components can be re-deposited for further composition or for com-
ponents' integration with other systems. Unlike ADL-based components, the extracted
components in our approach can resolve the defined research problem i.e., one can
achieve the following: 1) code-independent composition. 2) does not need to select all
the exposed methods (i.e., services in ADL). 3) re-deposition of a configured compo-
sition for further reuse.
8.3 Evaluating RX-MAN By Gannod and Cheng's Framework

Gannod and Cheng's Framework is designed to evaluate the software reverse engineering and recovery techniques, aimed at legacy systems by their by-product. "A by-product is an artefact that is constructed by a reverse engineering approach/tool as a result of analysing program code" [GC99]. The framework evaluates the approaches from informational and evaluational point of view. The informational criteria includes the classification of an approach, platform of availability and the language of use. The evaluational criteria includes the values of semantic dimensions, functional abstraction and structural abstraction of the by-product.

8.3.1 Informational Evaluation

Gannod and Cheng classify the reverse engineering approaches as formal or informal techniques. According to Gannod and Cheng's classification, informal approaches rely on pattern matching and syntactic structure of code, and the formal approaches are based on using some type of formal analytical method for deriving a specification from source code (i.e., using a formally specified language with well-known semantics).

The informal techniques are further classified into plan-based (pattern matching, textual analysis etc.) and parsing-based (RX-MAN etc.), and formal techniques are further classified into transformational (techniques that transform one notation of semantics into another by specifying a formal context) and translational (techniques that translate a program into an equivalent formal specification e.g., creation of a directed graph from source code to show feature locations).

Figure 8.12 (a) is showing the parameters of informational evaluation.

8.3.2 Evaluation Based on By-Product

The selected framework evaluates the reverse engineering approaches, based on their by-product. The framework evaluates the by-products by defining semantic dimensions. Semantic dimensions\(^3\) evaluates an approach based on the accuracy, precision, traceability and semantic distance of the by-product. These dimensions are explained as follows:

\(^3\)Each semantic dimension is classified into: Lowest, Medium Low, Medium High and Highest.
1. **Semantic Distance:** The semantic distance shows the abstraction difference (of number of levels) of input and output of a reverse engineering approach. Greater the semantic distance, the more abstract the by-product e.g., if an approach restructures the source code without adding any layer of abstraction, the semantic distance is non-existent. On the other hand, if an approach yields design concepts in natural language, the semantic distance is not zero. The value of semantic distance depends on the inter-step distance of an approach i.e., if an approach consists of three steps and each step produces a representation that is more abstract than the previous, the semantic distance would be equal to number of such inter-steps. Figure 8.11 is showing the units of semantic distance that are varied from lowest to highest, depending on the by-product.

2. **Semantic Accuracy:** ”The semantic accuracy describes the level of confidence that a specification is correct with respect to the input source code” [GC99]. The approaches that are based on syntactic code structure (e.g., parsing) are the most accurate i.e., they have high confidence whereas, the approaches that are based on semantic models are the least confident e.g., approaches based on natural language processing. It is due to the fact that accuracy gets lost while following a high level abstraction model of extraction rather than a parser that captures the whole code base as it is. Based on the by-product, Figure 8.11 is showing the assigned values of accuracy by the framework.

3. **Semantic Precision:** The semantic precision is based on the level of details of a specification and the degree that the specification if formal. The least precise
by-product is natural language as it has potential of ambiguity whereas, a more precise by-product is a UML diagram as it has formally specified semantics associated with its notation. Figure 8.11 is showing the semantic precision, assigned to different by-products.

4. **Semantic Traceability:** "The Semantic traceability describes the degree that a specification can be used to reconstruct an equivalent program" [GC99]. It is highly dependent on the semantic accuracy and precision of the by-product. An equivalent by-product can only be achieved if an approach is accurate enough to capture the input without loss, and precise enough to formally specified the notation of by-product (e.g., semantics of a component model). Figure 8.11 is showing the semantic traceability, assigned to different by-products.

It is clearly understandable from the defined semantics dimension that accuracy, precision and traceability of an approach are closely connected in terms of the by-product whereas, the semantic distance can impact the accuracy and precision of an approach, if the notation of by-product is not formally well-defined or loses the semantics due to lower accuracy. An ideal component directed reverse engineering approach should provide high accuracy, precision and traceability, and the semantic distance is dependent on the nature of the defined approach.

**Structural Abstraction** is based on the ability of an approach to enclose the source code in well-defined modular entities e.g., encapsulation of statements into a computation unit of X-MAN.
Functional Abstraction is a description of a software system that is based on the semantics of a program i.e., a functional abstraction describes the program's behaviour.

In short, assignment of the actual code to a higher abstracted entity (e.g., component) is structural abstraction and availability of the description of semantics to access and use that highly abstracted code is functional abstraction. The selected framework classifies both these abstractions into as-built (i.e., low-level) and abstraction (i.e., high-level). Figure 8.12 (b) is showing the parameters that evaluate the approaches, based on their by-products.

8.3.3 Evaluation and Discussion

Appendix E (Section E.1) is showing the abbreviation index that assigns an abbreviated letter to each approach for evaluation from informational and by-product point of view.

Table 8.13, 8.14 and 8.15 are showing the informational evaluation, proposed by the framework. In these tables, the Platform of an approach depends on the availability of the tool of that approach on a specific platform system. Platform availability is also dependent on the secondary tools, used by approaches to execute the proposed methodology. Language parameter is showing the language of use i.e., the programming language of the source systems used for evaluating and applying an approach.

The classification based on Technique will help in calculating the values of semantic dimensions as they are dependent on this classification.

An important factor to consider in informational evaluation is the selection of the framework. In these tables, the Platform\(^4\) of an approach depends on the availability of the tool of that approach on a specific platform system. Platform availability is also dependent on the secondary tools, used by approaches to execute the proposed methodology. Language parameter is showing the language of use i.e., the programming language of the source systems used for evaluating and applying an approach.

The classification based on Technique will help in calculating the values of semantic dimensions as they are dependent on this classification.

An important factor to consider in informational evaluation is the selection of

\(^4\)Parameters of all the tables are highlighted in bold and italics.
Table 8.15: Informational Evaluation (c)

<table>
<thead>
<tr>
<th>Language</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
<th>N</th>
<th>O</th>
<th>P</th>
<th>Q</th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>U</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COBOL</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OO</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Language Independent</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTML/Tag-Based</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Platform</td>
<td>PC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Macintosh</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SOLARIS/UNIX</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Technique</td>
<td>Plan-Based</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Parsing-Based</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Transformational</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Translational</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.16, 8.17 and 8.18 are showing the evaluation by by-product using Gannod

**Technique** for component directed reverse engineering approaches (that produce explicit components). All of these approaches are either parsing-based (e.g., index w, z, A, y, r, V) or plan-based (e.g., index p, m, l, n). Plan-based approaches use heuristics and define abstraction models to capture the source code (hence, they are less accurate than the parsing-based approaches). It is due to the fact that syntactic code capturing preserves the code without any loss and that is why the proposed approach (RX-MAN) followed static reverse engineering. In component directed reverse engineering, one cannot extract semantically correct components if there is a loss of functionality in the process (e.g., in **Translational** approaches).
CHAPTER 8. EVALUATION OF RX-MAN

and Cheng’s framework. In the stated tables, parsing-based approaches have the highest accuracy, shown by H (e.g., index q, r, s etc.) whereas, plan-based approaches have the lowest accuracy, shown by L (e.g., index a, b, c, d etc.). Approaches based on formal translation and transformation have accuracy values medium-high (MH) and medium-low (ML) respectively.

Approaches that produce the formally defined by-product have the highest precision (e.g., index r, V etc.) whereas, by-products like graphs/UML diagrams have precision value of medium-high (MH). Those approaches that produce code trace as an output have the lowest precision value (e.g., index a). In case of component directed approaches, those that produce inexplicit components have precision value medium-high (e.g., index B, C etc.) whereas, those that produce explicit components have the highest precision value (e.g., index V, w etc.). All the values of accuracy and precision have been assigned by following the Gannod and Cheng’s guidelines. Inexplicit component directed approaches have been assigned the lower precision value than the explicit component directed approaches because explicit one’s produce a by-product with formally defined semantics using a formal notation and a language (e.g., ADL). Inexplicit one’s produce clusters that are not formally defined in most cases. Approaches that produce restructured code have medium-low (ML) precision value.

Traceability is defined as the degree that a specification can be used to reconstruct

### Table 8.16: Evaluation by By-Product (a)

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>l</th>
<th>m</th>
<th>n</th>
<th>o</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>MH</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>Precision</td>
<td>L</td>
<td>L</td>
<td>MH</td>
<td>MH</td>
<td>L</td>
<td>MH</td>
<td>L</td>
<td>MH</td>
<td>L</td>
<td>MH</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>MH</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>Traceability</td>
<td>L</td>
<td>L</td>
<td>MH</td>
<td>MH</td>
<td>L</td>
<td>MH</td>
<td>L</td>
<td>ML</td>
<td>L</td>
<td>ML</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>ML</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>Distance</td>
<td>L</td>
<td>L</td>
<td>MH</td>
<td>MH</td>
<td>L</td>
<td>MH</td>
<td>L</td>
<td>ML</td>
<td>L</td>
<td>ML</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>ML</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>Functional</td>
<td>As-Built</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Structural</td>
<td>Abstraction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

### Table 8.17: Evaluation by By-Product (b)

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>q</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>u</th>
<th>v</th>
<th>w</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Precision</td>
<td>MH</td>
<td>H</td>
<td>L</td>
<td>ML</td>
<td>MH</td>
<td>L</td>
<td>MH</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>MH</td>
<td>MH</td>
<td>MH</td>
<td>MH</td>
</tr>
<tr>
<td>Traceability</td>
<td>MH</td>
<td>H</td>
<td>L</td>
<td>ML</td>
<td>MH</td>
<td>L</td>
<td>MH</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>MH</td>
<td>MH</td>
<td>MH</td>
<td>MH</td>
</tr>
<tr>
<td>Distance</td>
<td>MH</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>MH</td>
<td>L</td>
<td>MH</td>
<td>L</td>
<td>ML</td>
<td>L</td>
<td>ML</td>
<td>ML</td>
<td>ML</td>
<td>ML</td>
<td>L</td>
<td>MH</td>
</tr>
<tr>
<td>Functional</td>
<td>As-Built</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Structural</td>
<td>Abstraction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

...
an equivalent program. To achieve such an equivalency, one needs accuracy (to capture the semantics) and precision (to formally specify the equivalent notation). In the given approaches, those that have the H accuracy and precision have the highest (H) traceability. The semantic distance has values MH and L for graphical models (concept lattices etc.) and re-structured code respectively. In case of component directed approaches, those that yield inexplicit and explicit components have distance values of ML and L respectively. Components are different from the other graphical notation because they retain the functionality and semantics of the original source code though, wrap it in an abstract notation of semantics (i.e., component model). For these approaches, functionality and execution still exists at the code level but the way of accessing that implementation changes. Therefore, values ML and L have been assigned because components are at lower level of abstraction than the domain concepts and the lattices. Explicit components have the lower distance than the non-explicit because explicit one's also represent behaviour of methods' interactions as services or exposed methods.

Those approaches that yield graphical notations provide higher functional abstraction (e.g., index L, M, N etc.) whereas, those that produce code-trace provide as-built (at same level as code) functional abstraction (e.g., index G, P etc.). All the component directed approaches provide high level structural abstraction (e.g., index V, E, A etc.), whether they produce explicit or inexplicit components. Those approaches that produce code re-structuring or feature models do not provide any abstraction as they neither provides a description of behaviour nor a structural notation to abstract the source code (e.g., index t, G, H etc.).

From components point of view, the above tables show that the component directed reverse engineering provides high structural abstraction yet none of the approaches provide functional abstraction. Functional abstraction is needed to extract a blueprint that can facilitate the reuse of extracted components. Without any such help, it is inevitable to get familiarise with a code base in order to exploit the semantically valid
re-usability.

The proposed approach (RX-MAN) has the highest accuracy (as it is parsing-based), the highest precision (as the by-product is formally defined with semantics of composition), the highest traceability (due to the highest accuracy and precision) and the lowest distance (as the extracted code is structurally abstracted but preserves the same executable functionality as the input source code). The purpose of using this framework was to evaluate the overall domain of reverse engineering in terms of the by-products, and to justify the selected analysis type for component directed reverse engineering (as the selected methodology provides highest accuracy, precision and traceability). The succeeding section will confine this broader evaluation by discussing only those reverse engineering approaches that produce explicitly defined components. The focus of such comparison will be on re-usability and code-independent composition.

### 8.4 Component Directed Approaches Versus RX-MAN

As stated in the Section 3.4.3, RX-MAN provides code-independent re-usability due to separation of control and computation in the extracted components (as part of the reverse engineering algorithms), and because of the repository that is implemented as part of the approach to support design and deployment phase composition. Considering these factors, Table 8.19 is showing a comparison of RX-MAN with other approaches that produce explicitly defined components.

In the table, the term **Primary Tool** refers to availability of a tool that is designed specifically for an approach whereas, **Secondary Tool** means ready to use tool, used by an approach to implement the methodology. **Automation** shows whether an approach is automated or semi-automated. **Control Mapping** shows whether an approach uses endogenous mapping (reverse engineer control and computation together) or exogenous mapping (separate control from computation as part of the reverse engineering process). The term **Computation Mapping** shows whether an approach mixes computation with control in the reverse engineered notation. **Extracted Architecture** is data driven if control and computation are not separated in the extracted components (i.e., data values will determine the route of execution as they are part of the control flow). **Composition Mechanism** can be method calls (for component models based on objects), ports (for ADLs) or connector (in case of RX-MAN). The term **Code Generation** shows whether an approach

---

5 Abbreviation list of approaches (column names) is available in Appendix E.
has the ability to generate executable code from the extracted components and **Repository Implementation** means whether an approach implements and supports repository as part of its implementation.

Many approaches do not consider or implement repository as part of the reverse engineering and same reflects in their tools. Lack of repository reduces re-usability as re-composition of the deposited components is impossible. Unlike RX-MAN, all the component directed reverse engineering approaches cannot generate executable code from the extracted components but can only provide the code-level manual configurations to tailor the extracted components. On the other hand, RX-MAN can compose components to form composites, and can generate executable code of those composites (such code can be used outside of the developed tool and it enhances the re-usability of reverse engineered resources) in order to reuse the implementation to its maximum potential. Index V (RX-MAN) in Table 8.19 shows that ours is the only approach that can offer the following benefits: 1) separate control from computation (to facilitate code-independent composition) in the process of reverse engineering. 2) generate code from executable components in its tool. 3) support repository as part of the reverse engineering. 4) extraction of control-driven architecture i.e., unlike ADLs, execution of components is driven by the control structures (composition connectors), not by the data channels and data values.

<table>
<thead>
<tr>
<th>Tool Support</th>
<th>l</th>
<th>m</th>
<th>n</th>
<th>r</th>
<th>w</th>
<th>y</th>
<th>z</th>
<th>A</th>
<th>E</th>
<th>p</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Tool</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Secondary Tool</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Automation</td>
<td>Automated</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Semi-Automated</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Control Mapping</td>
<td>Endogenous</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Exogenous</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Computation Mapping</td>
<td>Mixed</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Separated</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Extracted Architecture</td>
<td>Data Driven</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Control Driven</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Composition Mechanism</td>
<td>Methods</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Ports</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Connectors</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Code Generation</td>
<td>No</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Yes</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Repository Implementation</td>
<td>No</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Yes</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 8.19: RX-MAN Versus Component Directed Approaches
CHAPTER 8. EVALUATION OF RX-MAN

<table>
<thead>
<tr>
<th>Approach</th>
<th>Re-Composition</th>
<th>Repository Deposit</th>
<th>Component Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVACompExt</td>
<td>✗</td>
<td>✗</td>
<td>UML</td>
</tr>
<tr>
<td>Antoun et al. Design</td>
<td>✗</td>
<td>✗</td>
<td>ArchJava</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subsystem Structure</td>
<td>✗</td>
<td>✗</td>
<td>ADL</td>
</tr>
<tr>
<td>Identification</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Choumbe et al.</td>
<td>✗</td>
<td>✓</td>
<td>EJB</td>
</tr>
<tr>
<td>Component Oriented Archi.</td>
<td>✗</td>
<td>✓</td>
<td>OSGi</td>
</tr>
<tr>
<td>Archimetrix</td>
<td>✗</td>
<td>✗</td>
<td>ADL</td>
</tr>
<tr>
<td>L2CBD</td>
<td>✗</td>
<td>✗</td>
<td>Not Defined</td>
</tr>
<tr>
<td>Quality Centric</td>
<td>✗</td>
<td>✗</td>
<td>ADL</td>
</tr>
<tr>
<td>Memory Constrained</td>
<td>✗</td>
<td>✓</td>
<td>JavaBeans</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alshara et al.</td>
<td>✗</td>
<td>✓</td>
<td>SOFA</td>
</tr>
<tr>
<td>RX-MAN</td>
<td>✓</td>
<td>✓</td>
<td>X-MAN</td>
</tr>
</tbody>
</table>

Table 8.20: Comparison With Approaches based on Explicit Components

Repository implementation is only possible if an approach is based on a component model that supports repository. Table 8.20 is showing the reverse engineering approaches that are based on explicitly defined components along with the component models they used. In the table, only four approaches (other than RX-MAN) are based on component models that support repository i.e., approaches of Chouambe et al., component oriented architecture, Memory Constrained Environments and Alshara et al. Though, out of these four, only Chouambe et al. have implemented repository as part of their reverse engineering approach, and their repository can neither provide re-composition nor the code-independent composition of extracted components.

Due to all these stated differences with other component directed reverse engineering approaches, RX-MAN has managed to achieve code-independent re-usability, which is not dependent on binding of all the ports of components being reused. Extraction and deposition of components, and evaluation of the approach shows that the proposed solution has solved the stated research problem. The end result also supports the well-structured reverse engineered notation as compared to original legacy code because the extracted notation preserves encapsulation in computation units of the X-MAN components.
8.5 Evaluating RX-MAN By Bellay and Gall's Framework

This section presents a comparison between RX-MAN tool and tools of other component directed reverse engineering approaches that produce explicitly defined components. For comparison, we use Bellay and Gall's capability assessment [BG98]. This assessment revolves solely around the capabilities of tools of reverse engineering. After empirical evaluation (Section 8.2), evaluation based on by-product (Section 8.3) and comparison with other component directed approaches (Section 8.4), the purpose of this assessment is to evaluate the proposed tool with tools of other approaches using well-defined parameters of assessment.

Bellay and Gall define four functional categories to assess reverse engineering tools i.e., 1) analysis 2) representation 3) editing/browsing and 4) general capabilities.

**Analysis** is classified into parser functionality (or functionality of plan-based heuristics to capture a code base) and parsing functionality (i.e., parameters that define the quality of code, captured by a parser or heuristics).

**Representation** is classified into textual and graphical. For both types, a set of parameters is defined to evaluate the quality of representation.

**Editing/Browsing** capabilities are evaluated using a set of parameters that determine the different properties of a tool from editing/browsing point of view e.g., availability of an integrated text editor in a tool, search functionality in a tool or availability of search functionality to efficiently use a tool.

**General Capabilities** include platform availability, multi-user support, tool extensibility, storing capability etc.

All the parameters that define these four categories are shown in the tables below. These tables sum up the evaluation of component directed reverse engineering approaches (that produce explicitly defined components) from Bellay and Gall's point of view.

The assessment for all the categories is done using one of the three methods.

1) An enumeration of possible type.
2) Yes or No to classify the availability of a functionality.
3) A four level scale, where the scale shows whether a tool provides a specific functionality. This four level scale is categorised as excellent (represented by ++), good (represented by +), acceptable (represented by o) and not at all (represented by −). In addition to this scale sign / is used if a functionality cannot be assessed or not relevant
to a tool.

<table>
<thead>
<tr>
<th>Supported Platforms</th>
<th>I</th>
<th>M</th>
<th>N</th>
<th>R</th>
<th>W</th>
<th>Y</th>
<th>Z</th>
<th>A</th>
<th>E</th>
<th>P</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-User Support</td>
<td></td>
<td></td>
<td></td>
<td>P/M</td>
<td>P</td>
<td>P/M</td>
<td>P/M</td>
<td>P/M</td>
<td>P</td>
<td>P</td>
<td>P/M</td>
</tr>
<tr>
<td>Tool Extensibility</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Composition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parser</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User Interface</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functionality</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Storing Capabilities</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parsed Code Storable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Edited Representation Storable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tool State Storable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selections Storable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Views Storable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output Capabilities</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Storing of Output</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output Formats</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intelligent Printing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output (Exporting)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output Capabilities</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Automatic Generation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Macros</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output Formats</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>History of locations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search Facility</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Online Help</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.21: Evaluation: General Capabilities

Table 8.21 is showing the general capability assessment of tools using the Bellay and Gall's framework. In Table 8.21, the approach at the index I is not extensible from integration point of view as it is implemented in a closed environment. It is extensible from functionality point of view as it can extend the generation of UMLs of the extracted components. It supports PC and Mac as supported platforms (P/M in table). This tool does not support any repository (Storing of Output) that can be used to store the extracted notation.6

Approaches at the index m and n are methodologies that do not discuss any particular tool as part of the implementation therefore, general capabilities of these indexes cannot be evaluated.

The approach at the index r supports PC, Mac and Unix platforms (P, M, U) and its tool is extensible as it can be integrated with the other plug-in based tools. Its output format is ADL-based component (C in table).

The tool at the index w supports PC and Mac platforms. It is not quite extensible as it is implemented in a close environment (not a plug-in that can be integrated easily). This tool also produces components as an output notation. This tool, like the previous ones does not support repository as part of its implementation and therefore, storing

---

6A repository to store design artefacts, extracted clusters or class diagram notation is not the same as a repository that can store components. In all given approaches, only the approaches at the index y, z and V have repositories that store actual extracted components.
capabilities are not good.

The approach at the index \( y \) depends on a secondary tool for most of its implementation and also used its primary tool to achieve the end result. It supports a repository (in the form of a database). The major tasks of the primary tool (repository and parsed notation) can be extended and integrated easily though, its dependency on a secondary tool does not make it very extensible for other implemented features.

The tool of the approach at the index \( z \) supports the repository to store design components (this approach uses an extension of UML components) but it is not extensible from Tool Extensibility point of view, as the implementation can be changed to support some other component model but cannot be extended to compensate such endeavours.

The approach at the index \( A \) does not support repository in its tool therefore, the storing capabilities of the tool are not there at all. This approach uses an open-source secondary tool \( \text{Rigi} \) (Rigi repository is not a component-based repository). As an open-source tool, \( \text{Rigi} \) is extensible, as one can customise or integrate it to reuse the available features accordingly (though it demands a lot of code changes).

The approach at the index \( E \) does not support repository neither it is extensible due to its closed architecture. Only possible way to change/add the functionality is by changing the existing implementation. For index \( p \), the approach used many secondary tools, but does not implement its own tool. Therefore, the extensibility is not there (it is not extensible if one has to work on many tools in order to extend the methodology).

The RX-MAN approach at the index \( V \) has excellent (++) tool extensibility because all the features in it are implemented as standalone plug-ins. Any of the implemented plug-in can be integrated or added with other Java projects. The Output Format of RX-MAN includes the executable code (EC in Table 8.21) in addition to the X-MAN components.

Almost all the component directed reverse engineering tools do not support Tool State Storable and View Storable etc. Such parameters are more relevant to those tools that are semi-automated and used to extract high level design or semantics. Overall, a field has value ++ (excellent) if that functionality of the tool is easy to extend without making any changes in the current implementation. The field value is + (good) if a functionality can be extended but requires changes in the current implementation. The filed value is − (not at all) if a specific extensibility or a capability is impossible due to nature of the research and its implementation.
### Table 8.22: Evaluation: Tool Analysis

Table 8.22 is showing the capability assessment from Analysis point of view. In the table, only the tools at the index m and V (RX-MAN) can take an application system, directory or a file structure as an input (i.e., Project Type) whereas, rest of the approaches are designed for single system applications only. Therefore, Ease of Project definition in the table is determined by the flexibility of a tool to take variety of inputs. Tools at the index r, w and y are capable of incremental parsing. None of the evaluated tools supports Re-parsing of an application (to reflect the new changes in the source code in extracted components). Every tool support Fault Tolerance to parse incomplete/syntactically incorrect source code but this comes at the price of unknown errors in the semantics of extracted notation.

**Define and Undefined** means if a tool can support defined compiler-based commands and Command Configurable means if a tool has the functionality to define customised compiler-based commands. The tools at the index p and V (RX-MAN) show errors while parsing (Quality of Parse Error Statements) whereas, such information is not available for rest of the tools. None of the tools supports Continuity at Error i.e., ability to continue after an error. **Point and Click** means if a tool has the ability to move between source code and respective parsed notation.

**Parsing Results** refers to the information after the parsing that shows some sort of relation between the source code and the reverse engineered notation (e.g., total number of methods in the source code VS methods distribution in the extracted components in RX-MAN) and **Parsing Speed** refers to the speed in terms of time between the start of parsing and the retrieval of end results. Those approaches that are not automated

---

<table>
<thead>
<tr>
<th>Source Language</th>
<th>l</th>
<th>m</th>
<th>n</th>
<th>r</th>
<th>w</th>
<th>y</th>
<th>z</th>
<th>A</th>
<th>E</th>
<th>p</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Type</td>
<td>A</td>
<td>F/A/D</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>F/A/D</td>
</tr>
<tr>
<td>Ease of Project definition</td>
<td>o</td>
<td>++</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>++</td>
<td></td>
</tr>
<tr>
<td>Incremental Parsing</td>
<td>N</td>
<td>/</td>
<td>/</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Re-parse</td>
<td>-</td>
<td>/</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>Y</td>
<td>/</td>
<td>/</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Define and Undefined</td>
<td>-</td>
<td>/</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Command Configurable</td>
<td>-</td>
<td>/</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Support for Additional Compiler</td>
<td>-</td>
<td>/</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Quality of Parse Error Statements</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>Parse Abort-able</td>
<td>N</td>
<td>/</td>
<td>/</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Errors Viewable</td>
<td>N</td>
<td>/</td>
<td>/</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Continuity at Error</td>
<td>N</td>
<td>/</td>
<td>/</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Point and Click</td>
<td>N</td>
<td>/</td>
<td>/</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Parsing Results</td>
<td>-</td>
<td>/</td>
<td>/</td>
<td>-</td>
<td>o</td>
<td>+</td>
<td>o</td>
<td>-</td>
<td>-</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>Parsing Speed</td>
<td>o</td>
<td>/</td>
<td>/</td>
<td>o</td>
<td>o</td>
<td>+</td>
<td>+</td>
<td>o</td>
<td>o</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

---

7Table 8.22 Legend: LI=Language Independent OO=Object-Oriented J=Java CB=COBOL Y=Yes N=No F=File D=Directory A=Application
(e.g., index \( w, E, p \)) have the lower parsing speeds because semi-automated extraction is dependent on the run time manual decisions. The automated approaches have been assigned the same scale value (i.e., +) because almost all the approaches are based on capturing the syntactic structure and every interaction in a source code that will take \( O(n^2) \).

Table 8.23 is showing the capability assessment from Tool Representation point of view. In the table, Textual Representation and Report Properties are not considered because none of the component directed approaches are relevant to these parameters of the framework. In Table 8.23, the tools at the index \( w, z \) and \( V \) have directory structure in them that can be used to navigate or create output resources. Build Map and call Graph mean the ability of a tool to show a visualisation of these entities in the source code being reverse engineered. Control flow means ability of a tool to show the control structure between entities of a source code in reverse engineered notation. RX-MAN tool (index \( V \)) is different in this category because it shows the control structure as composition connectors and properties windows to compose and deposit the components but it does not show the control structure of the original source code in a composite component. Rest of the terms are quite self-explanatory. Only two tools can provide combined views in their implementation i.e., tool at the index \( A \) (it combines the views of extracted sub-systems) and the index \( V \) (RX-MAN tool combines the code view, graphical view and repository view to build/retrieve/compose/deposit components).

<table>
<thead>
<tr>
<th>Graphical</th>
<th>l</th>
<th>m</th>
<th>n</th>
<th>r</th>
<th>w</th>
<th>y</th>
<th>z</th>
<th>A</th>
<th>E</th>
<th>p</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directory Layout</td>
<td>-</td>
<td>/</td>
<td>/</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Build Map</td>
<td>-</td>
<td>/</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Call Graph</td>
<td>-</td>
<td>/</td>
<td>/</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Control Flow Graph</td>
<td>-</td>
<td>/</td>
<td>/</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Data Flow Graph</td>
<td>-</td>
<td>/</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Data Structure Diagram</td>
<td>-</td>
<td>/</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Entity Relation Diagram</td>
<td>-</td>
<td>/</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>State Transition Graph</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>Combined Views</td>
<td>-</td>
<td>/</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 8.23: Evaluation: Tool Representation

Table 8.24 is showing the capability assessment from Editing/Browsing point of view. In the table, Integrated Text Browser shows the availability of an internal browser in the tool that can be used to fulfil functionalities of the tool. Intelligent Control means ability to point out a line, opening a file on command, properties view of an entity etc. RX-MAN tool (index \( V \)) has many intelligent controls e.g., property...
view of a component to show the order of execution of methods in it, drag and drop interfaces that are linked with the back end code generation, and centralised repository with the ability to annotate and comment the components for future reference. *Highlighting* means highlighting of source code and *Visualisation Functions* means browsing of source code from different aspects.

The *User Interface* means availability of short-cuts for browsing the code. This

<table>
<thead>
<tr>
<th>Integrated Text Editor/Browser</th>
<th>Y</th>
<th>/</th>
<th>/</th>
<th>Y</th>
<th>Y</th>
<th>Y</th>
<th>Y</th>
<th>Y</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>External Editor/Browser</td>
<td>N</td>
<td>/</td>
<td>/</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Intelligent Control</th>
<th>-</th>
<th>/</th>
<th>/</th>
<th>-</th>
<th>+</th>
<th>+</th>
<th>-</th>
<th>-</th>
<th>+</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Integrated Browser Functionality</th>
<th>I</th>
<th>m</th>
<th>n</th>
<th>r</th>
<th>w</th>
<th>y</th>
<th>z</th>
<th>A</th>
<th>E</th>
<th>p</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highlighting</td>
<td>-</td>
<td>/</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Visualisation Functions</td>
<td>-</td>
<td>/</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Speed</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>Search Functions</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>User Interface</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>History</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>Hypertext Capabilities</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.24: Evaluation: Editing/Browsing

specific field is dependent on the technology stack of implementation e.g., if a tool is implemented using eclipse plug-in, it will come with all the default short-cuts of eclipse environment. Therefore, the filed value / has been assigned to all the approaches. *Speed* refers to speed of the internal editor/browser and it is also dependent on the technology stack (such entities are not the priority of component directed reverse engineering tools). The term *Search functions* refers to a search functionality implemented by a tool for specific purposes. All the relevant approaches can use the built-in search functions of their respective environments of implementation but none have implemented a specific search-based functionality, unique to the tool. Therefore, this parameter is not classified and not considered in Table 8.24. The term *History* refers to ability to store browsed locations of the source code. None of the tools has this functionality as it seems more relevant to the tools that extract variability models or configurational matrices. The last term *Hypertext Capabilities* means ability to jump to a specific code element across any file. This field is also primarily dependent on the technology stack of implementation.

Bellay and Gall's capability assessment sums up the following primary benefits of RX-MAN tool over other component directed reverse engineering tools: 1) RX-MAN tool can produce executable code and components both. 2) Unlike most of the other tools, RX-MAN tool can reverse engineer single application, set of classes, directory
and libraries. 3) RX-MAN tool is designed by considering re-usability and extensibility. All the entities of the tool (e.g., repository, Xcore model, parser etc.) are implemented as standalone plug-ins and therefore, can be integrated with any plug-in project. 4) Repository and functionality of RX-MAN tool can also be used for the design phase composition (other tools can only work on reverse engineered output) to construct new components from scratch. 5) RX-MAN tool has many intelligent controls e.g., synchronisation between Graphiti based interface and properties dialogues of eclipse to synchronise back end code generation from the graphical components.
8.6 Summary

This chapter covered the comprehensive evaluation of the proposed approach. The presented evaluation consisted of following aspects: 1) empirical validation/evaluation, based on five code bases. 2) evaluation by Gannod and Cheng's framework to compare the traceability, precision and accuracy of each reverse engineering approach. 3) comparison with other component directed reverse engineering approaches that produce explicitly defined components. The focus of this evaluation was on re-usability and the code-independent composition. 4) evaluation of RX-MAN tool with tools of other component directed reverse engineering approaches.

Above evaluations proved that the proposed approach can deposit and reuse the extracted components with code-independent composition. The presented reverse engineering is different from other approaches because it separates the control and computation of the source code, and also extracts utility components to enhance re-usability. RX-MAN approach has the highest precision, accuracy and traceability according to Gannod and Cheng's framework. RX-MAN tool can be used for design and deployment phase of component's life cycle. Unlike other tools, the proposed tool can also generate executable code from the extracted components though, the implemented tool lacks the incremental parsing and functional abstraction (e.g., documentation and state diagrams to show the semantics of source code in the extracted components).
Chapter 9

Conclusions and Future Work

“Reasoning draws a conclusion, but does not make the conclusion certain, unless the mind discovers it by the path of experience.”

— Roger Bacon

In this thesis, a component directed reverse engineering approach is presented that can yield explicitly defined components. The proposed approach is different from other related approaches because it is based on an encapsulated component model. The proposed approach separates the control from computation to achieve code-independent composition and re-usability. Such separation can enable further development of the legacy systems with better cohesion and re-usability. Re-usability of the extracted components is way better than the ADL-based approaches because X-MAN components do not require code-level configurations and binding of all the required and provided ports of a component.

To demonstrate the validity and advantages of the proposed approach, the approach was evaluated empirically, by using a framework of reverse engineering, by using a framework of reverse engineering tools and by providing a direct comparison with other component directed reverse engineering approaches. The succeeding sections will sum up the research objectives, contributions, challenges, achievements, limitations, future work and also answer the stated research questions of Chapter 1.
9.1 Addressing the Research Questions

The Research problem stated that there is no such component directed reverse engineering approach that can: compose the reverse engineered components without changing the code at all required places, allow to reuse the components without binding all required and provided services and support the re-deposition of composed components for future reuse of the same integrated configuration.

Based on the research problem, the primary research question defined in the Chapter 1 was to explore the possibility of achieving code-independent re-usability of reverse engineered components along with the ability of re-deposition. Chapter 2 and 3 explained that to achieve the stated goals, one needs a component model that separates control from computation and defines exogenous composition. There was also a need of a methodology that should be based on static analysis and could capture the legacy code without losing any functionality. In other words, what was needed was an exogenous component model (X-MAN) and a well-defined approach that could capture and map the source code to X-MAN meta-model.

The secondary research question was to find out that how well a code base could be re-structured to reduce coupling (better re-structuring means better re-usability) as part of the reverse engineering process. For better cohesion in the extracted components, our approach did the follows:

1. the proposed approach was based on interactions and invocations of methods in an object-oriented legacy code. It used the modifiers and chain of invocations of methods to cluster them in a way that resulted in minimum coupling.
2. the approach reverse engineered atomic or composite X-MAN components (from extracted clusters that were based on methods’ invocations and modifiers) in which control was extracted out of the computation of methods. One such component could only interact with others via composition connectors. Such semantics provided better cohesion and loose coupling as no computation went out of a computation unit of an X-MAN component.

9.2 Fulfilling the Research Aims and Objectives

To achieve the primary aim (to investigate the feasibility of establishing a component directed reverse engineering approach that can provide code-independent re-usability
9.2. **FULFILLING THE RESEARCH AIMS AND OBJECTIVES**

and composition of re-deposited components) of the research, following objectives had to be fulfilled:

1. the proposed reverse engineering approach that mapped the legacy code to a higher abstraction notation (components) without losing any functionality in the process.
2. development of a reverse engineering tool, based on the proposed approach.
3. to evaluate whether or not the proposed research could provide code-independent re-usability and composition of the re-deposited components.

---

To achieve the first objective, realisation consisted of a rule base of mapping from source code to X-MAN components, based on static analysis. Every input of this mapping was a code scenario and output was one or more than one (atomic or composite) X-MAN components (Figure 9.1). Multiple scenarios based on methods' interactions, invocations and control structure of Java code (if-else, switch, loops etc.) were presented and mapped to their X-MAN equivalent output (Chapter 5).

For the second objective, the development of RX-MAN tool consisted of following parts: 1) a customised parser that was an extension of Abstract Syntax Tree (AST) parser to capture a code base. 2) a code mapper that implemented the rule base to map the re-structured code to a notation that could be mapped to components. 3) a component mapper that defined a rule base to map the source code to X-MAN meta-model. 4) a component repository for depositing and retrieving the components.

For the realisation of third objective, there was no evaluation framework that was designed specifically for the reverse engineering of components though, few evaluation frameworks closest to our approach were Gannod and Cheng's framework [GC99] that focused on the general reverse engineering, and Bellay and Gall's reverse engineering
tool capability assessment [BG98]. Due to empirical nature of the research, RX-MAN approach was evaluated in following ways:

1. by extraction and deposition of valid X-MAN components from five legacy code bases.
2. by conducting empirical evaluation of code bases.
3. by presenting possible scenarios of composition and re-deposition of extracted components.
4. by a comparison with the other component directed approaches using Gannod and Cheng’s framework and Bellay and Gall’s capability assessment.
5. by a comparison with the other component directed reverse engineering approaches, based on re-usability and code-independent composition.

To achieve the secondary aim (how well a code base can be re-structured to get minimum coupling), a component model i.e., X-MAN was selected after detailed analysis of the domain of component based development. To propose reverse engineering of X-MAN components, a customised version of X-MAN was used that did not involve any compositional services but provided encapsulation of computation in computation units of X-MAN components. Two reverse engineered components cannot be coupled in a composition as methods of one component can only interact with methods of the other component exogenously.

9.3 Research Contributions

It is now confirmed that this thesis has fulfilled the intended research contributions (Section 4.3) as it includes a customised version of X-MAN (Section 3.4.4.2), an ecosystem of reverse engineering and component based development (Chapter 7) and flexible re-usability of the extracted components (Section 8.2). Along with these intended contribution, the proposed research has made the following major contributions:

• A reverse engineering approach that provides code-independent re-usability: The proposed approach is different and provides better re-usability than other ADL-based approaches because it provides re-usability of the extracted components without demanding any code-level changes. RX-MAN also does not enforce the binding of all the ports of all the components involved in a composition but only requires the order of execution of exposed methods and data parameters by using click and drag interfaces. The reverse engineered components in RX-MAN can be re-deposited after code-independent composition to
preserve the semantics of a composition for future reuse. Such a re-deposition is very efficient and time saving in constructing and enhancing new systems by reusing the legacy code.

- **A taxonomy of Reverse Engineering:** The proposed research presented a taxonomy of reverse engineering (Figure 2.1). All the relevant approaches aimed at legacy systems were classified based on their analysis type, programming languages they could reverse engineer, output notation and the input notation each approach required. Such a taxonomy is practical and precise in deciding the suitability of an approach for reverse engineering a code base.

- **Classification and analysis of Component Models:** The proposed research presented a taxonomy of component models (Table 3.1 and 3.2). One of the major achievements of this research was to integrate the domain of component based development with reverse engineering to show the impact of one on the other and vice versa.

- **Research Publications:** This research led to the publication of three conference papers and one journal paper. The three papers covered the classification of reverse engineering [AL.16], reverse engineering of architectural notation [AL.17] and reverse engineering of encapsulated components from object-oriented legacy code [AL.18].

## 9.4 Research Challenges

The following challenges have been identified while developing the automated approach to extract X-MAN components:

- **The selection of development stack:** Most of the reverse engineering tools are closed for future development i.e., they require a lot of effort and changes at the code level to serve as a base tool for future purposes (Table 8.21). The real challenge was to come up with a technology stack that cannot only provide the re-usability of the components but the re-usability of the approach itself. Therefore, each major entity of the RX-MAN was developed as a standalone integrate-able plug-in. Future research in the domain of reverse engineering can take the advantage of reusing the developed repository, model, parser and back end by easily integrate the required plug-in.

- **Identification of the factors that cause code-dependent composition:** One of the
primary aims of the proposed research was to provide code-independent composition of the extracted components to enable re-usability. The real challenge was to analyse the current component directed reverse engineering approaches to identify the core reason behind the code-dependent composition. This analysis alone required a lot of understanding of the domain of component based development. Identification of the exogenous composition and its semantics as compared to the endogenous one was a key finding towards code-independent composition.

- Addition of Design Phase Composition: One of the major entities missed from the state-of-the-art component directed reverse engineering approaches was the design-phase composition of components as part of the reverse engineering. All such approaches emphasise on the reuse of the extracted components but do not discuss the creation of new components to enhance the system using component-based development (i.e., not enough emphasise on the semantics and use of the selected component model). To resolve this challenge, our approach presented the design-phase composition and the repository to support both design and deployment phase composition. Therefore, RX-MAN integrated component-based development and reverse engineering in one environment, not only for reverse engineering the components but also for constructing and enhancing the current/new systems using the X-MAN notation in the same tool.

- The evaluation of RX-MAN: One of the real challenges was to comprehensively evaluate the proposed approach. Most of the component directed reverse engineering approaches (e.g., [ASST+16]) do evaluation by showing the empirical statistics but do not consider important factors of reverse engineering i.e., accuracy, precision etc. Similarly, these approaches do not emphasise on the developed tools (which is important due to empirical nature of the research). To tackle all such shortcomings, RX-MAN was evaluated from four aspects i.e., empirically, tool centric (Bellay's assessment), quality centric (i.e., accuracy, precision etc. by Gannod's framework) and by a direct comparison with other component directed reverse engineering approaches (Section 8.4).

9.5 Limitations and Future Work

The results achieved by evaluation justified the completion of proposed aims and objectives however, the limitations of the proposed approach need to be defined explicitly.
Limitations can be summarised as follows:

- **Object-Oriented Relations:** One of the major shortcomings in RX-MAN is the absence of consideration of special relations in an object-oriented code e.g., aggregation, inheritance, composition etc. By considering such relations and defining specific rule base for them can yield better clustering and therefore, better and more cohesive components.

- **Limitations in Mapping the Java Constructs:** As explained in the Section 7.2, the current tool (implemented approach) cannot map the following Java constructs as no rules have been defined for them: 1) instantiation based constructs e.g., `instance of` command. 2) dynamic constructs e.g., polymorphism. 3) exception based statements e.g., `throw` or `catch` constructs in Java. 4) abstract classes.

- **Parameters of Clustering:** Few approaches (e.g., [DPB14]) extract directed graphs to map the method calls as edges and methods as vertices. Based on the number of edges between vertices, a rule base clusters the vertices together and later define components. RX-MAN only works by defining clustering on class-level or package-level i.e., it does not use any threshold to define allocation of methods to a cluster. Package-level clustering provides good abstraction ratio but requires more exposed methods (i.e., more familiarity with the source code) to achieve a semantically valid composition. On the other hand, class-level clustering enables easier composition from semantics point of view but extracts components with poor abstraction ratio. However, this shortcoming can be tackled by composing and re-depositing composites that can be reused later.

- **Lack of Functional Abstraction:** Like other component directed reverse engineering approaches, RX-MAN also does not provide any functional abstraction (Table 8.18) of the legacy code (i.e., blue print document or a graphical notation that can help in reusing the extracted components by explaining semantics of the source code). Lack of such abstraction demands the understanding and familiarity of the original source code to define the correct order of execution of exposed methods to achieve semantically valid composition.

- **Iterative Extraction:** Iterative extraction can help in maintaining the extracted components by keeping them updated with the new versions of a source code. At the moment, no such functionality is implemented in RX-MAN.

- **Intelligent Parsing:** Intelligent parsing can be defined as a semi-automation of specific scenarios that can help in defining the final product according to personalised requirements. It is not exactly a shortcoming but a good supportive feature.
There are many aspects of the proposed research that can define the direction for future endeavours. Few prominent ones are as follows:

- The X-MAN component model is algebraic ([LdC17a]), hierarchical and supports explicit variation points therefore, X-MAN can define the product families in the domain of product line engineering [PBVDL05, DCTL+16]. The proposed reverse engineering can be used in re-factoring the software product lines to automate the application engineering.

- The proposed tool and the research can further integrate the domains of reverse engineering and component-based development by defining an ecosystem of component directed reverse engineering. Such ecosystem can generate functional models, requirement specification documents, state charts and re-usability blue prints to integrate the whole development under one tool.

- The rule base can be refined and re-factored to add threshold-based cohesion parameters, and instead of components, a full working system in X-MAN notation can be extracted from legacy applications. Such system will consist of hierarchical X-MAN components, composed by adapter and composition connectors. However, intelligent parsing and some semi-automation is desirable for such extraction.
Bibliography


[AMHS+14] R Al-Msie’Deen, Marianne Huchard, A-D Seriai, Christelle Urtado, and Sylvain Vauttier. Reverse engineering feature models from software


[LVDH] Chris Lüer and André Van Der Hoek. Composition environments for deployable software components (technical report uci-ics-02-18 department of information and computer science, university of california). Internet Citation.


[MKM09] Sameer Kumar Mishra, Dharmender Singh Kushwaha, and Arun Kumar Misra. Creating reusable software component from object-oriented


[PM07] Denys Poshyvanyk and Andrian Marcus. Combining formal concept analysis with information retrieval for concept location in source code.


Appendix A

Reverse Engineering: Prerequisites and Required Expertise

Table A.1 shows the prerequisites for implementing an approach. Prerequisites have classified into approach centric process, i.e., *macro constant's selection, landmarks method selection, domain concepts, corpus extraction* and *profiling*. Profiling is the most common prerequisite. RECoVar [ZB13] is an approach that requires the selection of the macro constants before it can be applied. It shows code based variability by extracting a model from the code. Users have to define the macro constants in the code to use them in conditional compiling while generating the model. Such macro constants can be *if-def* blocks or anything that can define a variation in pre-compilation and they are called variation points. Another approach Landmarks and Barriers [WRW07] demands selection of *landmark* methods. Landmark methods are those that have a key role in execution of a feature. Hence, in order to select landmark features one must have to know that feature composition in terms of code. *Barrier* methods are those methods that do not have major importance from a feature point of view and they have to be selected in order to decrease the size of generated variability graph. Combining FCA with IR [PM07] demands generation of the document corpus by latent semantic indexing (LSI). Document corpus is the generation of the part of the code that matches the user queries and it should be in vector space form which is a well known form in LSI. Formal Context Analysis (FCA) uses this notation to start matching and producing the output in the form of concept lattices. Dependence Graph [CR10] needs identification and selection of the nodes that should be included in the search graph in order to search the implementation of a feature. The relevant code parts cannot be
selected unless one has the knowledge and some familiarity with the domain and composition of the features in terms of code. Therefore, code understanding and domain knowledge is essential before executing this approach. In case of Reverse Engineering Feature Models [SLB+11], domain knowledge is needed because domain expert have to select the parent of each feature at each step and correct decisions require code and domain knowledge.

Table A.2 shows that required expertise are classified as FCA, LSI, natural language processing (NLP), profiling, vector space modelling (VSM) and domain knowledge. Product Variants [XXJ12], Concept Analysis [EKS01], Combining FCA with IR [PM07] and Locating Features in Source Code [EKS03] require the knowledge of FCA. FCA demands the designing of a formal context in which objects are defined in order to generate the model. Product Variants [XXJ12], Cerberus [EAAG08], Combining FCA with IR [PM07] and Heuristic-Based Approach [ADPAG10] require the knowledge of LSI. Natural language Parsing [AT10] requires Natural Language Processing which is a well established research domain on its own. Dynamic Feature Traces [EDV05], Scenario Driven Dynamic Analysis [SMADP06], Trace Dependency Analysis [Egy03], Featureous [OJ10], Locating Features in Source Code [EKS03],
Static and Dynamic Analysis [RHLR08], Cerberus [EAAG08], STRADA [EBG07], Call-Graphs [BD06], Focused views on Execution Traces [BVD08], Concept Analysis [EKSO1], Heuristic-Based Approach [ADPAG10] and Software Evolution Analysis [GDG05] require profiling. SNIAFL [ZZL+06] requires the knowledge of VSM. VSM is a special kind of LSI. Finally, Dependence Graph [CR10] and Reverse Engineering Feature Models [SLB+11] need the domain knowledge.

Table A.2: Reverse Engineering: Required Expertise

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Required Expertise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Variants [XXJ12], Concept Analysis [EKSO1], Combining FCA with IR [PM07], Locating Features in Source Code [EKSO3]</td>
<td>FCA</td>
</tr>
<tr>
<td>Product Variants [XXJ12], Cerberus [EAAG08], Combining FCA with IR [PM07], Heuristic-Based Approach [ADPAG10]</td>
<td>LSI</td>
</tr>
<tr>
<td>Natural language Parsing [AT10]</td>
<td>NLP</td>
</tr>
<tr>
<td>Dynamic Feature Traces [EDV05], Scenario Driven Dynamic Analysis [SMADP06], Trace Dependency Analysis [Egy03], Featureous [OJ10], Locating Features in Source Code [EKSO3], Static and Dynamic Analysis [RHLR08], Cerberus [EAAG08], STRADA [EBG07], Call-Graphs [BD06], Focused views on Execution Traces [BVD08], Concept Analysis [EKSO1], Heuristic-Based Approach [ADPAG10], Software Evolution Analysis [GDG05]</td>
<td>Profiling</td>
</tr>
<tr>
<td>SNIAFL [ZZL+06]</td>
<td>Vector Space Modelling</td>
</tr>
<tr>
<td>Dependence Graph [CR10], Reverse Engineering Feature Models [SLB+11]</td>
<td>Domain Knowledge</td>
</tr>
</tbody>
</table>

All the component based reverse engineering approaches are based on static analysis therefore, they do not need profiling. Unlike the above stated approaches, component based ones construct and define their own mechanism from code capturing to clustering. Hence, one does not need any pre-requisite implementation or specific expertise like FCA, LSI etc., though such approaches demand the comprehension of component based development (CBD) and composition mechanism of the selected component model.
Appendix B

X-MAN Semantics

B.1 X-MAN Services

The X-MAN component model our approach uses does not include any services. The new version of X-MAN also includes the concept of compositional services. A compositional service is an entity that does not exist in a legacy code (or in a code base) and cannot be retrieved or mapped by reverse engineering (services can be placed in reverse engineered components by forward engineering but they will not always reflect the appropriate behaviour of the legacy code). The composition of services means that services of all the components involve in a composition should also be composed together i.e., if a component A has n methods in its service S1 and a component B has m methods in its service S2 then the composition of A.B will also produce service S3 with n+m methods. We use the X-MAN model in which only those methods of computation units will be involved in a composition that are selected to be exposed before instantiating the components in deployment phase e.g., if two components A and B are instantiated with one exposed method each, then there will be only two methods that will be involved in this composition.

B.2 X-MAN Meta-Model

Figure B.5 is showing the X-MAN meta-model generated by E-Core [Mer17] using EMF (Eclipse Modelling Framework). This meta-model covers both the design and
deployment phases of component-based development. According to this model, Component\(^1\) can be specialised as an atomic or composite. An Atomic Component has one Computation Unit that is connected with all the methods implemented in it via Methods. A component is connected with zero to n exposed methods (E.Method in the Figure) where n is the number of methods in its computation unit and each one fulfils a contract via its Input and Output Parameters. The order of execution of E.Methods in an atomic components is set by E.MethodsRef. The interface Provider enables one or more exposed methods or zero or more Data Elements. Data elements are abstracted by Data. Connector abstracts the concept of exogenous connectors and can be Adapter Connector or Composite Connector. These connectors are specialised by their concrete types i.e., Sequencer, Selector etc. Composable in inherited by Connector and provides composition between Data Channels and Composition Connectors via Composite and Connection.

Control flow is abstracted by Connection and data flow is abstracted by Data hence, both are separated from each other unlike ADLs. Composable provides the link between design and deployment phase as it is inherited by both Component instance and Connector. To summarise, each component in the meta-model is either atomic or composite. Each component is connected with its exposed methods and their parameters. In case of composite component, Co-ordination Connection accesses the Composite connector and Connection accesses the Adapter Connection in case of an atomic component. Data abstracts data flow and Connection abstracts control flow. Composable enables the instantiation as it is inherited by both Connection and component instance.

### B.2.1 Adapter Connectors

Adapter connectors are applicable on atomic X-MAN components. These connectors are used for looping or conditional invocation of an atomic X-MAN component. These connectors are classified into Guard and Loop.

#### B.2.1.1 Guard

The Guard is used for conditional invocation of an atomic component. Guard can take a conditional statement as an input and only invokes the component if the condition is validated. Figure B.1 is showing an atomic component with Guard.

\(^1\)All the entities in meta-model are referred in italics here.
B.2.1.2 Loop

The Loop is used for conditional looping of an atomic component. It takes an iterator with an initial value, looping interval (increment by x or decrement by x if x is an integer) and a condition of termination. Figure B.2 is showing an atomic component with Loop connector.

B.2.2 Composition Connectors

These connectors are used to compose 2-n atomic components. Components can be executed in a defined order by Sequencer or conditional invocation of 2-n components is possible by Selector.
B.2.2.1 Sequencer

As the name suggests, this connector executes 2-n atomic/composite components in the specified order. The route with the lowest number have the highest priority of execution. Figure B.3 is showing two atomic components with Sequencer.

B.2.2.2 Selector

A Selector can take many conditional inputs to invoke one of the components out of 2-n. Figure B.4 is showing two atomic components with Selector.
APPENDIX B: X-MAN SEMANTICS

Figure B.5: X-MAN Meta-Model Generated by E-Core EMF
Appendix C

Code Mapper Implementation

C.1 AST Tree Creation and Compilation

```java
private void analyseMethods(IProject project) throws JavaModelException {
    IPackageFragment[] packages = JavaCore.create(project)
        .getPackageFragments();
    // parse(JavaCore.create(project));
    for (IPackageFragment mypackage : packages) {
        if (mypackage.getKind() == IPackageFragmentRoot.K_SOURCE) {
            createAST(mypackage);
        }
    }
}

private void createAST(IPackageFragment mypackage) throws JavaModelException {
    Methods_Extraction startExtraction = new Methods_Extraction();
    for (ICompilationUnit unit : mypackage.getCompilationUnits()) {
        // now create the AST for the ICompilationUnits
        MethodVisitor visitor = new MethodVisitor();
        CompilationUnit parse = parse(unit);
        parse.accept(visitor); // call sends one package at a time
        startExtraction.ExtractMainMethodToStart(visitor);
    }
}

private static CompilationUnit parse(ICompilationUnit unit) {
    ASTParser parser = ASTParser.newParser(AST.JLS3);
    // ...
}
```
APPENDIX C. CODE MAPPER IMPLEMENTATION

// Map options = JavaCore.getOptions();
// JavaCore.setComplianceOptions(JavaCore.VERSION_1_5, options);
// parser.setCompilerOptions(options);
parser.setKind(ASTParser.K_COMPILATION_UNIT);
parsed.setSource(unit);
parsed.setBindingsRecovery(true);
parsed.setResolveBindings(true);
return (CompilationUnit) parser.createAST(null); // parse

C.2 RX-MAN Visitors to Locate Nodes

package xman.ast.handlers;

public class MethodVisitor extends ASTVisitor {
// This HashMap is used to track invocation methods of each method
// you can see all the info by using invocationsForMethods.keySet()
// and can get invocations of a certain method by using
// visitor.invocationsForMethods.get(method) in Methods_Extraction Class
final HashMap<MethodDeclaration, ArrayList<MethodInvocation>>
    invocationsForMethods = new HashMap<MethodDeclaration,
                                                   ArrayList<MethodInvocation>>();

private MethodDeclaration activeMethod; // It is needed for HashMap storage
// of active method against all
// invocations

// List that stores and generate all the methods in all the packages
List<MethodDeclaration> methods = new ArrayList<MethodDeclaration>();
// List that stores all packages and their imports in terms of class imports
List<CompilationUnit> units = new ArrayList<CompilationUnit>();
// List that stores all simpleNames
List<SimpleName> simpleName = new ArrayList<SimpleName>();
// List that stores all VariableDeclarationStatement
List<VariableDeclarationStatement> variablesDeclaration = new ArrayList<VariableDeclarationStatement>();

// List that stores all invocated methods
List<MethodInvocation> methodInvocation = new ArrayList<MethodInvocation>();

// List that stores all classes
List<TypeDeclaration> classDeclaration = new ArrayList<TypeDeclaration>();

// List that stores all fields of classes
List<FieldDeclaration> fieldDeclaration = new ArrayList<FieldDeclaration>();

// List that stores positions of simple statements in class
List<Statement> Statements = new ArrayList<Statement>();

// List that stores positions of if statements in class
List<IfStatement> ifStatements = new ArrayList<IfStatement>();

// List that stores positions of switch statements in class
List<SwitchStatement> switchStatements = new ArrayList<SwitchStatement>();

// List that stores positions of do statements in class
List<DoStatement> doStatements = new ArrayList<DoStatement>();

// List that stores positions of For statements in class
List<ForStatement> forStatements = new ArrayList<ForStatement>();

// List that stores positions of while statements in class
List<WhileStatement> whileStatements = new ArrayList<WhileStatement>();

@override
public boolean visit(MethodDeclaration node) {
    activeMethod = node;
    methods.add(node);
    return super.visit(node);
}

public List<MethodDeclaration> getMethods() {
    return methods;
}

@override
public boolean visit(CompilationUnit node) {
    units.add(node);

    // System.out.println("Compilation unit: ");
APPENDIX C. CODE MAPPER IMPLEMENTATION

// node.getPackage().getName());
// System.out.println("Imports: " + node.imports());
return super.visit(node);
}
public List<CompilationUnit> getCompilationUnits() {
    return units;
}

@Override
public boolean visit(SimpleName node) {
    simpleName.add(node);
    return super.visit(node);
}

public List<SimpleName> getsimpleNames() {
    return simpleName;
}

@Override
public boolean visit(TypeDeclaration node) {
    classDeclaration.add(node);
    return super.visit(node);
}

public List<TypeDeclaration> getclasses() {
    return classDeclaration;
}

@Override
public boolean visit(FieldDeclaration node) {
    fieldDeclaration.add(node);
    return super.visit(node);
}

public List<FieldDeclaration> getFields() {
    return fieldDeclaration;
}

@Override
public boolean visit(MethodInvocation node) {
    // different from other visitor methods because we need Hash here
    if (invocationsForMethods.get(activeMethod) == null) {
        invocationsForMethods.put(activeMethod,
        new ArrayList<MethodInvocation>();
    }
}
invocationsForMethods.get(activeMethod).add(node);
return super.visit(node);
}

public List<MethodInvocation> getMethodInvocation() {
return methodInvocation;
}

public boolean visit(VariableDeclarationStatement node) {
for (Iterator iter = node.fragments().iterator(); iter.hasNext();) {
System.out.println("------------------");
VariableDeclarationFragment fragment = (VariableDeclarationFragment)
iter.next();
IVariableBinding binding = fragment.resolveBinding();
variablesdeclaration.add(node);
// System.out.println("binding: " +binding);
}
return super.visit(node);
}

public List<VariableDeclarationStatement> getVariables() {
return variablesdeclaration;
}

// for CONDITIONAL STATEMENTS
public boolean visit(IfStatement node) {
ifStatements.add(node);
return super.visit(node);
}

public List<IfStatement> getIfStatements() {
return ifStatements;
}

public boolean visit(SwitchStatement node) {
switchStatements.add(node);
return super.visit(node);
}

public List<SwitchStatement> getSwitchStatements() {
return switchStatements;
}

public boolean visit(DoStatement node) {
doStatements.add(node);
return super.visit(node);
APPENDIX C. CODE MAPPER IMPLEMENTATION

```java
    public List<DoStatement> getDoStatements() {
        return doStatements;
    }

    public boolean visit(ForStatement node) {
        forStatements.add(node);
        return super.visit(node);
    }

    public List<ForStatement> getForStatements() {
        return forStatements;
    }

    public boolean visit(WhileStatement node) {
        whileStatements.add(node);
        return super.visit(node);
    }

    public List<WhileStatement> getWhileStatements() {
        return whileStatements;
    }

    // class ends
```
C.3 X-MAN Intermediate Model to Store Code

```java
public class X_MAN_Component_Structure {
    public ArrayList<String> ExposedMethods = new ArrayList<String>();
    public ArrayList<DataTypeParameters> inputs = new ArrayList<DataTypeParameters>();
    public ArrayList<DataTypeParameters> outputs = new ArrayList<DataTypeParameters>();
    public ArrayList<String> Methods = new ArrayList<String>();
    public String computationUnit;
    public String SourceCode;
    public String componentName;
    public bool utilityComponent = false;
    public ArrayList<Connector_Stack> adapters = new ArrayList<Adapter_Connector_Stack>();
    public ArrayList<Connector_Stack> composites = new ArrayList<Composite_Connector_Stack>();
}
```

C.4 Captured Code of Brake Control System in RX-MAN notation
Figure C.1: Captured Code of Brake Control System
Appendix D
RX-MAN Tool

D.1 Selection of Exposed Methods Before Instantiation of A Deposited Component

Figure D.1: RX-MAN: Selection of Methods for Composition
D.2 Re-deposition of Composed Components

Figure D.2: RX-MAN: Deposition of Composed Components for Further Reuse (Deployment Phase Deposition)
# Appendix E

## Evaluation

### E.1 Abbreviation Index for Reverse Engineering Approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Allocated Index Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natural language Parsing [AT10]</td>
<td>a</td>
</tr>
<tr>
<td>Source Code Retrieval [LKE08]</td>
<td>b</td>
</tr>
<tr>
<td>Reverse Engineering Feature Models [SLB+11]</td>
<td>c</td>
</tr>
<tr>
<td>Call-Graphs [BD06]</td>
<td>d</td>
</tr>
<tr>
<td>Trace Dependency Analysis [Egy03]</td>
<td>e</td>
</tr>
<tr>
<td>Using Landmarks and Barriers [WRW07]</td>
<td>f</td>
</tr>
<tr>
<td>CERBERUS [EAAG08]</td>
<td>g</td>
</tr>
<tr>
<td>Systematic method approach [KC04]</td>
<td>h</td>
</tr>
<tr>
<td>Heuristic-Based Approach [ADPAG10]</td>
<td>i</td>
</tr>
<tr>
<td>Locating Features in Source Code [EKS03]</td>
<td>j</td>
</tr>
<tr>
<td>Shatnawi et al. [SSSA17]</td>
<td>k</td>
</tr>
<tr>
<td>Memory-constrained environments [WF14]</td>
<td>l</td>
</tr>
<tr>
<td>Quality centric approach [KSCC12]</td>
<td>m</td>
</tr>
<tr>
<td>Al-Shara et al. [ASST+16]</td>
<td>n</td>
</tr>
<tr>
<td>CORE [MKM09]</td>
<td>o</td>
</tr>
<tr>
<td>Antoun et al. [AAAC07]</td>
<td>p</td>
</tr>
<tr>
<td>RECoVar [ZB13]</td>
<td>q</td>
</tr>
<tr>
<td>Archimetrix [DPB14]</td>
<td>r</td>
</tr>
<tr>
<td>Static and Dynamic Analysis [RHLR08]</td>
<td>s</td>
</tr>
</tbody>
</table>
APPENDIX E. EVALUATION

Language Independent Approach [ZHP+14] t
Concern Identification [Tri09] u
Automatic Generation [Rob05] v
JAVACompExt [ARA+09] w
RECAST [EM93] x
Chouambe et al. [CKK08] y
Design Components [KSRP99] z
Subsystem Structure Identification [MOTU93] A
Favre et al. [FDE+01] B
Bunch Tool [MS06] C
Erdemir et al. [ETB11] D
Component-oriented Architecture [ASSF11] E
MoDisco [BCDM14] F
Evolutionary Algorithms [LHGB+12] G
Product Variants [XXJ12] H
Burd and Munro’s approach [BM98] I
Software Configurations using FCA [AMHS+14] K
Combining FCA with IR [PM07] L
Concern Graphs [RM02] M
Dependence Graph [CR10] N
Focused views on Execution Traces [BVD08] O
Scenario Driven Dynamic Analysis [SMADP06] P
Featureous [OJ10] Q
Dynamic Feature Traces [EDV05] R
Concept Analysis [EKS01] S
Software Evolution Analysis [GDG05] T
SNIAFL [ZZL+06] U
RX-MAN V
Appendix F

Glossary

**ADL-based Components**: Components that are based on architecture description language. These components offer composition by indirect method calls (ports) that can require or provide a functionality.

**Re-deposition**: Ability of an approach to retrieve the extracted components after reverse engineering and deposit them again after composition.

**Re-composition**: Composition of components after re-deposition.

**Exogenous Composition**: Composition of components that does not mix control and computation of methods.

**Endogenous Composition**: Composition of components that cannot separate control from computation.

**Component-directed Reverse Engineering**: Reverse engineering that yields components instead of feature locations or graphical models.

**Utility X-MAN Component**: An atomic X-MAN component that have utility methods i.e., methods in the computation unit do not call any other method.

**Design Phase Composition**: Construction of components from scratch by following semantics of a component mode.

**Deployment Phase Composition**: Composition of components after their instantiation from a repository in deployment phase.

**Semantic Dimensions**: Parameters to evaluate semantic accuracy, precision, traceability and distance of a reverse engineering approach.

**Iterative Parsing**: Ability to re-parse the new version of a source code and reflect the new changes in the extracted components.

**Intelligent Parsing**: Ability to empower the user to make decisions during reverse
engineering that can yield the required semantics in the extracted components. Semi-automation of reverse engineering is the way to provide intelligent parsing.