Feature-based Approach to Bridge the Information Technology and Business Gap

PhD Thesis

Fayez Eid Alazemi

This thesis is submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy.

Software Technology Research Laboratory
De Montfort University
Leicester - United Kingdom

2014

Dedication

To my parents, my wife and my children

for their love, support and encouragement

during this time of challenges

Abstract

The gap between business goals (problem domain), such as cost reduction, new business processes, increasing competitive advantage, etc., and the supporting Information Technology infrastructure (solution domain), such as the ability to implement software solutions to achieve these goals, is complex and challenging to bridge. This gap emerges for many reasons; for instance, inefficient communication, domain terminology misunderstanding or external factors, e.g. business change.

As most business and software products can be described by a set of features, a promising solution would be to link both the problem and solution domains based on these features. Thus, the proposed approach aims to bridge the gap between the problem and the solution domains by using a feature-based technique in order to provide a quick and efficient means for understanding the relationships between IT solutions and business goals.

The novelty of the proposed framework emanates from the three characteristics of the business-IT gap: the problem domain, the solution domain and the matching process. Besides the proposed feature-based IT-business framework, other contributions are proposed: a feature extracting method and feature matching algorithms.

The proposed approach is achieved in three phases. The first phase is to decompose business needs and transform them into a feature model (presented in UML diagrams); this is represented as a top-to-middle process. The second phase is a reverse engineering process. A system program code is sliced into modules and

transformed into feature-based models (again, in UML diagrams); these are represented as a bottom-to-middle process. The third phase is a model-driven engineering process. It uses model comparison techniques to match the UML feature models of the top-to-middle and bottom-to-middle phases.

The presented approach in this research shows that features elicited from the business goals can be matched to features extracted from in the IT side. This proposed approach is feasible and able to provide a quick and efficient means for improving feature-based business IT matching.

Two case studies are presented to demonstrate that the feature-oriented view of features from the users' perspective can be matched to the feature-oriented view of features in the IT side. This matching can serve to remove any ambiguities that may cause difficulties in the cases of system maintenance or system evolution, in particular when there are changes in requirements, which is to be expected when there is any business change.

Declaration

I declare that the work described in this thesis is original work undertaken by me for the degree of Doctor of Philosophy, at the software Technology Research Laboratory (STRL), at De Montfort University, United Kingdom.
No part of the material described in this thesis has been submitted for any award of any other degree or qualification in this or any other university or college of advanced education.
This thesis is written by me and produced using LaTeX.

Fayez Alazemi

Publications

F. Alazemi and M. Alawairdhi. Feature-based Approach to Bridge the Information Technology and Business Gap. September 17-19, 2013, page 87 pp . Baltimore, MD, USA, 2013. Recent Researches in Telecommunications, Informatics, Electronics and Signal Processing.

Acknowledgments

First and foremost, my deepest and humblest gratitude goes to Almighty Allah, who gave me the patience, health and strength that I needed to complete this research.

I was very lucky to have Professor Hussein Zedan, the Director of the Laboratory, as my supervisor. His continual guidance, scientific support and ideas have greatly helped me in developing my thinking, technical writing and widening my vision. Without his guidance, support and encouragement, this thesis would not have been possible to accomplish.

Also, many thanks go to my second supervisor Dr. Amelia Platt for her encouragement and support.

I also would like to thank all of my colleagues, especially Abduallah Almarshid and Saad Almutairi, in the Software Technology Research Laboratory at De Montfort University for their valuable suggestions and discussions.

I am grateful to all members of the STRL for creating the academic and homelike environment and for their continued support, especially Mrs. Lynn Ryan and Mrs. Lindsey Trent.

I am also thankful to the staff in the Research Office at De Montfort University for their excellent management.

I am deeply indebted to my friend Dr. Mohammed Alawairdhi for his concern and encouragement through all my study years.

I would like to express my gratitude to my dear brothers and sisters for their

support, concern and encouragement.

I would like also to express my sincere gratitude to my mother and father, who gave their unending love and support, and for everything they have given in their life for me. Without their care, prayers and support, it would have been very difficult for me to accomplish this project.

I would like to express my deepest love and gratitude to my wife, who has stood by me throughout all these difficult years, and who has offered me her constant support, patience, encouragement and unconditional love. Finally, my love goes to my children Khaled, Salma, Fares, Maha and Saud for the hope and encouragement they have given to me, without knowing it, to complete this work.

Contents

Dedication **I**

Abstract **II**

Declaration **IV**

Publications **V**

Acknowledgments **VI**

Table of Contents **XIII**

List of Figures **XVI**

List of Tables **XVII**

List of Abbreviations **XVIII**

1 Introduction **1**

1.1 Overview 2

1.2 Motivation 4

1.2.1 Business-IT Gap 4

1.2.2 Research Problem 4

1.3 Thesis Scope and Research Question 5

1.4 Research Methodology 6
1.5 Measure of Success 8
1.6 Contribution to Knowledge 9
1.7 Thesis Structure 9

2 Feature-based Approach: State of the Art **12**
2.1 Features and Features-Oriented Software Development (FOSD) 13
2.1.1 The Concept of Feature 13
2.1.2 Feature-Oriented Software Development (FOSD) 14
2.1.3 Feature Modelling 15
2.1.4 Feature Interaction 16
2.1.5 Feature Implementation 18
2.2 Software Evolution and Software Re-engineering 19
2.2.1 Software Engineering 19
2.2.2 Software Re-engineering 20
2.2.2.1 Re-engineering Classification and Software Abstraction 21
2.2.3 Reverse Engineering 23
2.2.4 Software Evolution 24
2.2.5 Laws of Software Evolution 25
2.2.6 Legacy Software System 26
2.3 Software Architecture 29
2.4 UML 31
2.4.1 Introduction 31
2.4.2 UML Concepts 32
2.5 Program Slicing 36
2.6 Requirement Engineering 38
2.7 Model Comparison 42

2.7.1 Model Comparison Phases 42
2.7.2 Model Versioning 43
2.7.3 Model Clone Detection 44
2.7.4 Model Comparison Approaches 44
2.8 Conclusion 46

3 Proposed Approach **47**
3.1 Framework Overview 48
3.2 Business Feature Elicitation 55
3.2.1 Business Analysis 56
3.2.2 Requirement Engineering 57
3.3 IT Feature Extraction 57
3.3.1 Program Understanding 58
3.3.2 Program Slicing 58
3.3.3 Program Dependence Graph 59
3.4 Feature Model Matching 59
3.5 Conclusion 60

4 Business Feature Elicitation **62**
4.1 Overview 63
4.2 Business Needs and Business Analysis 64
4.3 System Features 66
4.4 Requirement Engineering 68
4.4.1 Requirements Elicitation 69
4.4.2 Requirement Analysis 70
4.4.3 Requirement Specification 71
4.4.4 Requirement Validation 71
4.5 Requirement Elicitation Methods 72

4.5.1 Scenarios 73
4.5.1.1 Scenarios and Requirement Elicitation 74
4.5.1.2 Scenarios 76
4.5.2 Story Cards 79
4.5.3 Analysis to Derive Features 81
4.6 UML-based Feature Modelling 82
4.7 Conclusion 85

5 IT Feature Extraction 86

5.1 Importance of the Phase 87
5.2 Program Understanding 89
5.3 Program Slicing Step 90
5.4 Slices-to-UML Step 93
5.4.1 IT Feature Representation 96
5.5 Conclusion 99

6 Feature Model Matching 101

6.1 Introduction 102
6.2 The Matching Problem 103
6.3 Model Comparison and Approaches 104
6.4 Proposed Model Matching Approach 107
6.4.1 Model Matching Example 111
6.5 Conclusion 115

7 Case Study 116

7.1 Overview 117
7.2 Tool Support 117
7.2.1 Eclipse 117

7.2.2 Indus Java Program Slicer 118
7.3 An ATM System 119
7.3.1 The ATM Software System Application 120
7.3.2 Business Feature Elicitation Phase 125
7.3.2.1 Scenarios 125
7.3.2.2 Story Cards 127
7.3.2.3 Feature Presentation in UML Models 129
7.3.3 IT Feature Extraction Phase 130
7.3.3.1 Program Slicing Step 131
7.3.3.2 Program Dependency and Control Flow Graphs . . . 137
7.3.3.3 UML Model Feature Representation 138
7.3.4 Feature Model Matching Phase 140
7.3.4.1 Matching Algorithm 141
7.3.5 Matching Results 142
7.4 A Library Management System 142
7.4.1 The Library Management Software System Application 143
7.4.2 Business Feature Elicitation Phase 144
7.4.2.1 Scenario 144
7.4.2.2 Story Cards 145
7.4.2.3 Feature Presentation in UML Models 146
7.4.3 IT Feature Extraction Phase 147
7.4.3.1 Program Slicing Step 147
7.4.3.2 Program Dependency and Control Flow Graphs . . . 150
7.4.3.3 UML Model Feature Representation 151
7.4.4 Feature Model Matching Phase 152
7.4.4.1 Matching Algorithm 152
7.4.5 Matching Results 153

7.5 Evaluation and Discussion 153

7.6 Conclusion 156

8 Conclusion And Future Work 158

8.1 Summary of the Thesis 159

8.2 Research Questions Revisited 160

8.3 Future Work 161

Bibliography 181

A An ATM Case Study Source Code 182

List of Figures

2.1 Problem Domain and Solution Domain 13
2.2 Phases of the FOSD Process 15
2.3 An Example of Feature Modelling with a Feature Diagram Notation . 16
2.4 Feature-Oriented Programming 18
2.5 Re-engineering Classification and Software Abstraction 23
2.6 4+1 View Model 29
2.7 UML Diagrams 34
2.8 Levels of Requirements Engineering 40

3.1 An Overview of The Proposed Framework 49
3.2 Feature-Oriented IT-Business Framework 51
3.3 Framework Stages and Steps 54

4.1 Needs, Features and Requirements 63
4.2 System Software Lifecycle 64
4.3 Business Analysis Process 65
4.4 ATM Use Case. 67
4.5 Components of Requirement Engineering Domain. 69
4.6 Iterative Requirement Elaboration Process 75
4.7 ATM Scenario 77
4.8 Goals, Scenario and Features 79

4.9 Class Diagram - Display a Welcome Message. 83
4.10 Class Diagram - View Account Balance. 83
4.11 ATM User Authentication Activity Diagram 84

5.1 Level of Abstraction 87
5.2 IT Feature Extraction Process 89
5.3 Backward Slicing Example 91
5.4 Forward Slicing Example 92
5.5 Program Dependence Graph 93
5.6 Control Flow Graph 94
5.7 CFG and Program Slicing 95
5.8 Potential System Features in a CFG 98
5.9 UML Class Diagram 99

6.1 IT-Business Framework and Matching Stage 102
6.2 A Scholarly Teacher in USA and UK Academic System 104
6.3 Intersection Shape for $\Delta(v_1, v_2)$ 106
6.4 Class Converting for a Matching Algorithm 108
6.5 Feature Model Mapping 109
6.6 Class Model Matching Example 111
6.7 Proposed Algorithm Activity Diagram 112

7.1 Indus Slicing Result 119
7.2 A User Interface for an ATM Software System 120
7.3 An ATM User Welcome Screen 121
7.4 User Authentication Screen 122
7.5 Invalid Data Entry Screen 123
7.6 Main Menu Screen of the ATM Software System 124
7.7 ATM Scenario 126

7.8 Class UML Representation of the Welcome Feature 129
7.9 Activity UML Representation of 'authenticated user' Feature 130
7.10 ATM Software Classes and Methods 132
7.11 Program Slices Across the System 133
7.12 Snapshot of the Indus Slicing Tool 134
7.13 Part of The ATM Source Code Shows Sliced Statements 137
7.14 CFG and Statements Both Inside and Outside the Slice. 138
7.15 Welcome UML Representation 139
7.16 UML Activity Diagram of 'authenticate user' Representation 140
7.17 : Library Management System Scenario 144
7.18 : UML Class for Confirm Add Book Confirmation Message 147
7.19 Part of the Library Management System Source Code 150
7.20 CFG Includes Statements Both Inside and Outside the Slice. 150
7.21 UML Class for add Book Confirm Message 151

List of Tables

2.1 Legacy System Maintenance Options 28

4.1 Story Card for Bank Customer - Welcome Screen 80

4.2 Story Card for Login 80

4.3 Story Card for Main Menu 81

4.4 Feature 1 - Welcome Screen 82

4.5 Feature 2 - User Authentication 82

7.1 ATM System Source Code Statistics 124

7.2 Story Card for Customer - Welcome 127

7.3 Story Card for Customer - Login 128

7.4 Feature 1 - Welcome Screen 128

7.5 Feature 2 - User Authentication 129

7.6 Library Management System Source Code Statistics 143

7.7 Story Card for Library Management System - Add Book And Confirm 146

7.8 Feature 1 - Confirm Add Book 146

List of Abbreviation

CFGs	Control Flow Graphs
CFG	Control Flow Graph
RE	Requirements Engineering
IT	Information Technologies
FOSD	Feature-Oriented Software Development
FODA	Feature-Oriented Domain Analysis
FM	Feature Modeling
UML	Unified Modelling Language
PDG	Program Dependence Graphs
SRS	Software Requirement Specification
BA	Business Analysis
SBRE	Scenario Based Requirements Elicitation
PV	Program visualisation
IT	Information Technology
MDE	Model-Driven Engineering

ATM	Automated Teller Machine
OCL	Object Constraint Language
VCS	Version Control Systems
EMF	Eclipse Modeling Framework
VCS	Version Control Systems
AI	Artificial Intelligence
PIN	Personal Identification Number
IBM	International Business Machines
IDE	Integrated Development Environments
RSA	Rational Software Architect
SBRE	Scenario Based Requirements Elicitation
IEEE	Institute of Electrical and Electronics Engineers

Chapter 1

Introduction

Objectives:

- Present an introduction and motivation for this research
- Identify the research questions
- Present the research methodology
- Present the thesis structure

1.1 Overview

In the last two hundred years, technological tools as well as management methods have dramatically improved; from steam engines to electricity, from motorcycles to aeroplanes, from telexes to computers. Organizational methods have also improved; from small factories to assembly lines. These rapid changes have created many challenges and complexities for business and industry.

In the past fifty years, information technology (IT) has emerged rapidly and has become a key provider for (and a key factor in) the business world. It has improved productivity, opened up and created new business opportunities, minimized delivery times and allowed for flexibility in location decisions. Many businesses need guidance in creating and developing new technology, and many new technologies create new business opportunities. Many technological inventions (from mainframes to the web and mobile computing) have helped to increase business profits, ultimately delivering benefits to society. Nowadays, modern businesses cannot exist without IT.

IT providers are currently facing the challenge of rapid business change, and they have to respond quickly to the business changes, and do so in an appropriate manner. Even though there are many successful business-IT stories, there are also many project failures within the domain of IT-business cooperation. Some analysts report that the failure rate may exceed 50% of all projects [99]. Others have identified that the major factors determining whether a project succeeds or fails are requirement-related [66].

Software requirements, in general, describe the problem to be solved and the solution to this problem. These requirements utilize stakeholder goals, needs and agreements to capture system features. A feature can be described as a specific service that a product provides, but it also can be described as an attribute of the whole product, such as 'fault tolerant' or 'user friendly'.

Thus, the concept of feature has been defined in various ways. These definitions can be grouped into two main categories: abstract and technical. Some of the abstract definitions are:

1. "a prominent or distinctive user-visible aspect, quality, or characteristic of software system of systems" [75].

2. "a coherent and identifiable bundle of system functionality that helps characterize the system from the user perspective" [142].

3. "a product characteristic from user or customer views, which essentially consists of a cohesive set of individual requirements" [32].

4. "distinguishable characteristic of software (functional or non-functional) that is relevant to some of its stakeholders" [41].

On the other hand, there are the technical feature definitions. For example:

1. "an increment of program functionality" [12].

2. "a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirements, to implement and encapsulate a design decision, and to offer a configuration option" [8].

3. "a triplet, f = (R,W,S), where R represents the requirements the feature satisfies, W the assumptions the feature takes about its environment and S its specification" [37].

Since this research aims to match features from a business's perspective to IT feature perspective, in this research a feature is defined as:

A required user service that can be used as a distinctive characteristic of software system.

1.2 Motivation

1.2.1 Business-IT Gap

Business-IT gap or business-IT communication gap is mainly caused by a lack of understanding between business and IT professionals [120]. This lack of business IT alignment is considered to be a major factor of high cost and missed deadline projects [120][99]. Moreover, this gap is a major cause for business to lose time, budget, and even business reputation. To the business and IT professionals, this gap remains a challenge to have true service excellence. Therefore, it is important that IT and business professionals to understand each other needs and challenges [103].

In general, there are two main approaches to bridge this business IT gap. First approach is helping IT professionals with information needed to understand business needs and goals. In this approach IT is trying to get closer to business [9]. Second approach is helping business professionals with information needed to understand IT challenges and abilities. In this second approach business is trying to get closer to IT[127].

1.2.2 Research Problem

Stakeholders (e.g. users, customers and developers) think about and describe a software system in terms of the features it provides. A feature of a software system is any distinguishable characteristic of that software that is relevant to some of its stakeholders; these characteristics are either functional or non-functional [41]. The feature-oriented view of features from the users' perspective is not reflected in the software development process during product production. In other words, the connections between these features and the system software elements are not

obvious. This ambiguity cause difficulties in the case of system maintenance or system evolution, especially when there are changes in requirements, which is to be expected when there is any business change.

Software artefact traceability is well known as being an important factor in software development, evolution, maintenance and testing. Software artefact traceability has been defined as "the ability to follow the life of a requirement in a forward and backward direction" [61].

This research investigates tracing features from the users' perspective and from a technical viewpoint. Researchers in software traceability trace software requirements downstream from initial requirements down to the software system source code. Other researchers trace these requirements upstream, from the software system source code up to the software requirements. In this research, this linkage should flow downstream and upstream at the same time, downstream from business needs and upstream from software system source code, where both paths meet in the middle at a specific development stage. The UML software modelling stage for software system development is selected as being such a meeting point.

1.3 Thesis Scope and Research Question

The main focus of this thesis is to establish a framework and methodology in order to match business goals and needs to corresponding software systems features. The proposed feature-based business IT framework considers a system feature as a cornerstone for every stage of the proposed framework. The proposed framework follows systematic steps in order to match business needs to software systems features. There are three main steps, which are: business feature elicitation, IT feature extraction and feature model matching.

The first stage of the proposed framework focuses on business analysis and in

particular on software requirement engineering. The second stage uses a software reverse engineering process, aiming to extract features from software code, which are then represented as UML. Finely, the third stage focuses on model comparison, a technique in model-driven engineering. The scope of this thesis is represented in the issues covered in each stage of the framework.

Based on the above pertaining to the scope of this thesis, the main research work is to answer following question:

Is it possible to trace business goals to the software features at the source code level?

The main question opens the door to answering other sub-questions:

- Does the software satisfy the stakeholders' requirements?
- Can the efficiency of the software system be improved during the software evolution process?
- How can the software features be extracted from the source code?
- How can the software features be matched?
- How can the software features be represented?

1.4 Research Methodology

The research methodology for this thesis incorporates three approaches; the first is classified as formulative, as proposed by Morrison and George in their research approaches to software engineering [106]. The formulative approach involves the "development and refinement of theories, models, or frameworks that govern research

activities and support scientific progress through paradigm shifts" [106]. The second is constructive research. The main thrust of constructive research is the construction of new theory, algorithms, methods, models or frameworks based on the existing body of knowledge [40]. The constructive approach is "a research procedure for producing innovative constructions, intended to solve problems faced in the real world and, by that means, to make a contribution to the theory of the discipline in which it is applied" [29]. Finally, to evaluate the available tools and techniques, an empirical research approach is required. Empirical research is a "research that is based on experimentation or observation, i.e. evidence. Such research is often conducted to answer a specific question or to test a hypothesis" [83].

Accordingly, this research is conducted in four stages; the first one concerns comprehending the research problem and identifying the most relevant questions. The second stage concerns constructing a solution process. The third stage concerns validating the proposed methodology. Finally, the last stage concerns deriving conclusions.

1. **Identifying the Research Problem and Questions**

   Problem-related literatures were reviewed and studied in order to gain a full overview of the research problem. In addition, literatures related to the research problems, such as algorithms and the tools adopted in this research, were studied and analysed. To narrow down the research problem, a set of research questions was suggested and structured to clarify all the problem issues.

2. **Constructing a Solution**

   Constructing a solution is a mixture of reverse-engineering and forward-engineering processes. Its framework is composed of three main stages; firstly the business features elicitation stage, secondly, the IT features extraction stage, and

finally, the feature modelling mapping stage. Algorithms and rules for feature extraction and feature modelling matching were also developed.

3. **Validation**

   To demonstrate and validate the proposed approach, a case study is presented. The case study methodology is well known to be suitable for many kinds of software engineering research [124]. An ATM case study is adopted because it is a representative example, targeting the business and IT domains. The case study is designed to demonstrate whether the methodology works is capable of producing useful and meaningful results.

4. **Deriving Conclusions.**

   At the end of this research, conclusions are drawn, and the methods adopted and tested are discussed. Because a research is an iterative process in nature, new (related) research questions and suggestions are proposed to encourage future research work in the area.

1.5 Measure of Success

The criteria for measuring the success of the work conducted in this thesis are as follows:

- The research questions mentioned at the beginning of this research must be answered.
- The research's proposed architecture must be shown to be identifiably different from others.

- A study of why 'software system features' was chosen among other tracing methods must be done.
- The advantages of using feature-based mapping must be demonstrated.

1.6 Contribution to Knowledge

This research aims to investigate the possibility of tracing and matching features from the users' perspective with those from a technical viewpoint. Various aspects of both the problem and the solution domains are investigated. Therefore, the main contributions of this thesis are:

1. Developing a feature-based framework to link the features of both the business side and IT side, which provides a quick and efficient means for understanding the relationships between IT solutions and business goals.
2. Proposing a comparison algorithm to compare two UML diagrams. The algorithm is used to match the two models from both sides (business features and IT features).
3. Developing a method to extract features from a software source code based on PDG and CFG.

1.7 Thesis Structure

The following paragraphs briefly give a description of the remaining chapters of the thesis. At the end of each chapter, a summary is provided.

Chapter 2: Feature-based Approach: State of the Art

This chapter provides an overview, giving background information and describing other works related to the research topic. The background information relates to

the basic concepts of the research area, and the related works are those topics that are considered relevant to this research work. The chapter topics are: The concept of Features, Features-Oriented Software Development (FOSD), Software Evolution, Software Re-engineering, Unified Modelling Language (UML), Program Slicing, Requirement Engineering and Model Comparison.

Chapter 3: Proposed Approach

This chapter introduces the research framework for feature-oriented business and IT mapping. The three stages of the proposed framework are explained briefly. These stages are: Business feature elicitation, IT feature extraction, and Business-IT feature mapping.

Chapter 4: Business Feature Elicitation

The fourth chapter describes the first stage of the research's proposed framework, which is mainly the requirement engineering process. In this research, it is represented as the top-to-middle stage, and it describes business needs, business analysis and software requirement elicitation. This chapter starts from the business analysis process and goes though the requirement engineering process to discover the business needs and software requirements. Finally, features are presented as UML modelling.

Chapter 5: IT Feature Extraction

This chapter explains the bottom-to-middle stage of the research framework, which is mainly a program understanding process. The program understanding is achieved though software reverse engineering. Program static slicing is used to extract software-related source code statements. These sliced statements are trans-

formed into a visual representation form, which is later transformed into UML modelling.

Chapter 6: Feature Model Matching

The last stage is business IT feature matching, which is located in the middle of the proposed framework. It maps the outputs of the business feature elicitation stage and the IT feature extraction stage. Comparison methods are described and a new algorithm is proposed.

Chapter 7: Case Study

An ATM case study is presented to demonstrate the utility of the research approach, i.e. of the feature-oriented business IT framework. This chapter provides a description of an ATM system and then demonstrates the process of applying all the framework stages to the ATM system.

Chapter 8: Conclusion and Future Work

This chapter present a summary of the work conducted in this thesis. It then highlights the significance of the proposed research and draws a number of conclusions. Also, it discusses possible future work and makes suggestions.

Chapter 2

Feature-based Approach: State of the Art

Objectives:

- To present an overview of software feature and related work
- To discuss software evolution, software re-engineering and related terminology
- To define basic concepts of related to software understanding
- To explore UML and model-driven engineering

2.1 Features and Features-Oriented Software Development (FOSD)

2.1.1 The Concept of Feature

The definition of feature, whether abstract of technical (as described and defined through examples in the previous chapter), changes depending on the various software development phases. From the top of the software development process, the definition of feature is more abstract. However, during the software development process, the definition of feature becomes less abstract (problem-orientated) and more technical (solution-orientated), i.e. feature definitions become less abstract and more technical the more they describe the lower levels of the software development process. Thus, in the problem domain and during the software requirement engineering process, the definition of feature is abstract; for example, 'characteristic of software system'. On the other hand, during the design and performance processes, the feature definition is much more technical; for example, "structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirements" [8].

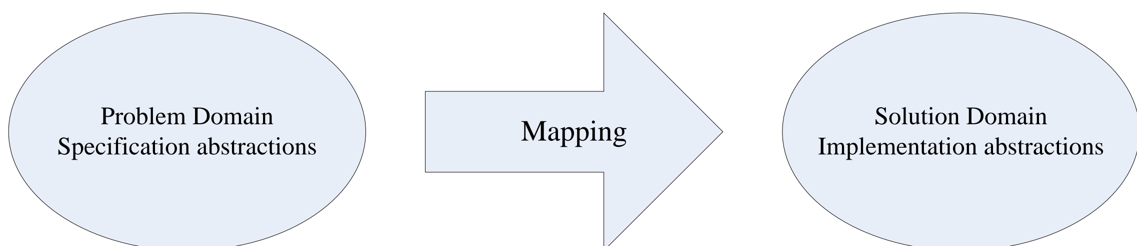

Figure 2.1: Problem Domain and Solution Domain [41].

Figure 2.1 shows the problem domain and the solution domain, as illustrated by

Czarnecki and Eisenecker [41]. The problem domain describes the problem in the real world, the software system requirements and the system behaviour. The solution domain describes the solution to the problem in the virtual world (the computer world), and how the system requirements and behaviours are satisfied. Feature definitions can be categorized as either feature definition from the perspective of the problem domain or feature definition from the solution domain. Feature definitions from the problem domain perspective describe features as services or what is expected from the software system. On the other hand, feature definitions from the solution domain perspective describe how features are implemented, i.e. the desired system functionality.

2.1.2 Feature-Oriented Software Development (FOSD)

In software systems, the concept of feature greatly assists in identifying similarities and differences in analysis, design and implementation of the software system phases. Feature-oriented software development (FOSD) is an approach to using the feature concept in the software development phases during the software development lifecycle. In FOSD, feature is the main focus during all the software development lifecycles: analysis, design, implementation, testing and software evolution. Features in other software development phases are derived from the structure and behaviour of the whole software system. However, in FOSD, features are systematically and explicitly used and defined. Features are defined as blocks within the structure of the software system and they facilitate software reusability. FOSD represents a very important property in the mapping of the features within the software requirements to all the software lifecycle phases; it creates a path from system requirements in the analysis phase, through the design phase, and down to the implementation level.

The FOSD process consists of four phases [7]; these are domain analysis, domain design and specification, domain implementation, and product configuration and

generation. In all these phases, the concept of feature is at the core. The first phase is the analysis domain, which determines all the features of the software system. Besides this, analysts include how features are related to each other, e.g., some features require the absence or presence of other features. The second and the third phases are the design and the implementation phases. In design and implantation, a feature is encapsulated as a feature artefact. From these sets of feature artefacts, the software system can be generated, and this is the fourth phase. Figure 2.2 shows the four phases of FOSD.

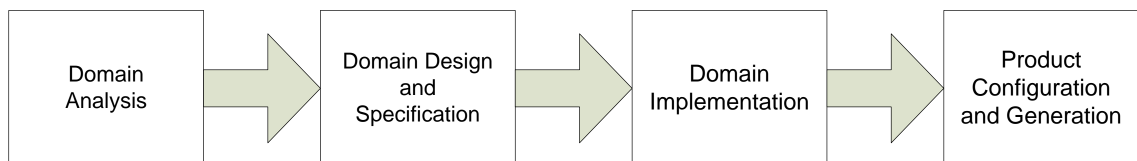

Figure 2.2: Phases of the FOSD Process [7]

2.1.3 Feature Modelling

The concept of feature modelling was first used by Kang et al. during their work on feature-oriented domain analysis (FODA) [75]. An FODA domain "is defined as a set of current and future systems sharing common capabilities. Domain analysis aims at discovering and representing commonalities and variabilities among them" [42]. Kang et al. used the concept of feature in an attempt to describe commonalities in and differences between software systems [75]. In their work to describe the relationships and dependencies within a set of features, they introduced the feature modelling concept. Feature modelling identifies the features from the end-users' perspective as well as the system's functionalities or capabilities.

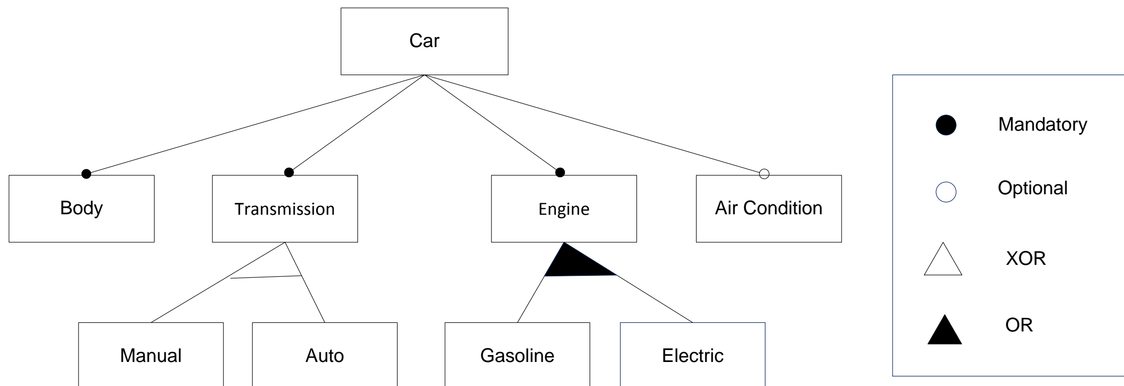

Figure 2.3: An Example of Feature Modelling with a Feature Diagram Notation [41].

Figure 2.3 shows a feature model that describes the variability of a simple car, i.e., different cars can be produced in the domain of the model 'car'. In the feature modelling, the root of the tree is the concept that is modelled. Other boxes are features, where a child depends on its parent features. The connectors or arcs are constraints on the ways in which the features are included in the model. In the figure, the filled circles indicate that every car must have a body, a transmission and an engine. However, the empty circle indicates that not every car must have air conditioning. Unlike FOSD, the early work of FODA feature modelling did not explicitly represent the features in the design or the code level.

2.1.4 Feature Interaction

The term 'feature interaction problem' was first used in the telecommunications industry, early in the 1980s [7][94][78]. A feature interaction emerges when a feature behaves unexpectedly because of adding or removing another feature or features. To illustrate the concept of feature interaction, two examples are given.

- Example 1: Phone call forwarding and call waiting

> A telephone company can provide many features in a phone system; e.g., caller ID, call forwarding, call waiting, three-way calling, etc. Each feature of the telephone system should be able to work independently. The call forwarding feature is activated for any incoming call. The call waiting feature is activated when a call is coming in and the receiver phone is busy with another call. In the second case, the incoming, the busy and the call forwarding features should also be activated. However, it is unclear which feature should work and which one should not.

- Example 2: A lift system

> Consider these features of a lift system:
> When the lift door has been opened, it will close automatically after 10 seconds.
> When the lift is overloaded, the door will not close.
> It is clear that the second feature conflicts with the first one. The first feature tries to close the door after 10 seconds while the second feature stops the door from closing until some passengers get out.

In feature-oriented software development, feature interaction is a major problem and it should be detected and resolved [18][28][79][76]. Researchers have proposed many techniques to detect and handle feature interaction [117][95][7]. The two main

feature interaction detection and handling methods are [82][34]:

1. Off-line detection: Feature Interaction detections are applied before the features are executed. An example is Interaction filtering.

2. On-line detection: Feature Interaction detections are applied after the features are executed. An example is Observers mechanism.

2.1.5 Feature Implementation

Software system developers traces features from the problem space to the features in the solution spaces. This tracing process has been a longstanding problem [61]. An approach to solve this problem is to make features and feature interactions explicit even in the programming language level [117]. A feature is built by a set of programming units. Building a feature must follow a feature model. Features merged together to group a feature packages where interactions among them are resolved. In this case, feature packaged are reusable. Feature package can be seen a single feature in other software system. Features implementation uses approaches similar to the one used for FOSD. It implements features so they can be composed in different combinations as illustrated in Figure 2.4.

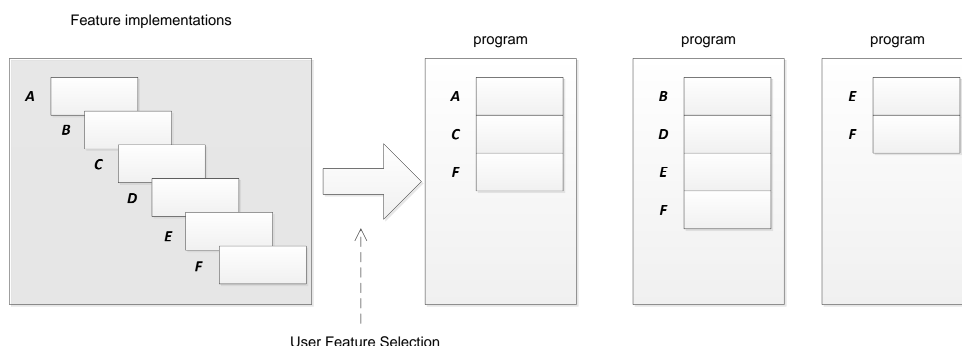

Figure 2.4: Feature-Oriented Programming [7].

2.2 Software Evolution and Software Re-engineering

This section presents an overview of software evolution and software re-engineering. Subsection 2.2.4 gives a description of software evolution. Subsection 2.2.6 describes legacy systems. The laws of software evolution are provided in Subsection 2.2.5. Software engineering is described in Subsection 2.2.1. Subsection 2.2.2 gives an overview of software re-engineering. Finally, Subsection 2.2.3 is an introduction to software reverse engineering.

2.2.1 Software Engineering

Engineering is the use of science and mathematical principles in designing, building and making things work. However, this definition of engineering is best understood in its relation to other disciplines [62]. Software or computer software is a term used to describe a computer program, which is a collection of computer data and instructions; other terms such as programs, applications, scripts and computer instructions are also used. It is difficult to describe software because it is virtual, i.e. it cannot be touched or seen; it is not physical like computer hardware. Computer software consists of computer instructions, which are lines of codes written by a computer programmer; they are compiled and stored in computer memory and they serve to manipulate computer behaviour. Building a software system is a highly complex; in fact, some believe that it is the most intricate and complex of human activities [22].

Software engineering is "a systematic approach to analysis, design, assessment, implementation, test, maintenance and reengineering of software, that is, the application of engineering to software. In the software engineering approach, several models for the software lifecycle are defined, and many methodologies for the dentition [sic] and assessment of the different phases of a lifecycle model" [88].

The need for software engineering emerged as an engineering solution for the

so-called 'software crisis' of the 1960s, 1970s and 1980s [25]. A software crisis occurs when many software projects fail, or run over time and budget.

2.2.2 Software Re-engineering

Software re-engineering utilizes a 'forward engineering' technique to re-implement a completely new system based on an original system's requirements and specifications [141]. Re-engineering can be the re-documenting of a legacy system, organising, restructuring, or translating the system to a more modern programming language. In general, the functionality of the system is not changed. In other words, the software re-engineering process takes an existing legacy system, which has become expensive or unable to update, and redesigning it. The output is higher performance, and the software should be easier to maintain and more reliable [2]. However, the major challenge is to fully comprehend the legacy system.

Thus, the main aims of software re-engineering are to understand the system under review and to redevelop it in order to achieve better functionality and performance. Software re-engineering objectives depend on the organisation's goals but, generally, there are four re-engineering objectives [108]:

1. Prepare for enhanced functionality
2. Improve maintenance
3. Access to the new platform
4. Improve reliability

Many terms are used in the software re-engineering domain. Some of these terms are clarified by Yang and Ward as follows [159]:

- Forward engineering is the well-known transformation process from high-level abstractions and logical, implementation-independent designs toward the physical implementation of a system.

- Reverse engineering is the process of understanding a system to (1) identify the system's components and their interrelationships, and (2) create representations of the system in another form or higher level of abstraction.

- Design recovery (or reverse design) is a subset of reverse engineering. Design recovery recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about a problem or application domains.

- Program understanding (or program comprehension) is a term related to reverse engineering. Program understanding always implies that understanding begins with the source code, while reverse engineering can start at a binary and executable form of the system or at high-level descriptions of the design. Program understanding is comparable with design recovery because both of them start at the source code level.

2.2.2.1 Re-engineering Classification and Software Abstraction

The software re-engineering process starts at the implementation level and ends at the implementation level again, passing through the whole software development lifecycle backwards (reverse engineering) and forwards (forward engineering). Therefore, software re-engineering can be classified into two main operations: reverse engineering and forward engineering. On the path of the software re-engineering process, there are four levels of software abstraction: conceptual, requirements, design and implementation.

The top level is the conceptual one. The conceptual level relates to the software's *raison d'être*, i.e. the purpose of the software. It explains the features expected from the software in general terms. The requirements level of abstraction comes under the conceptual level. Here, there is a description of what the system must do. However, it need not say anything, at this level, about how the system performs the work required of it. The requirement level of abstraction deals with defining, modelling, extracting, gathering and documenting the software requirements in order to better understand the problem [133]. The software design level of abstraction is a framework that describes and guides the implementation in order to achieve the requirements. This description is a set of representations describing the data structure, architecture and algorithmic procedure [159]. The software implementation level of abstraction is the lowest level. Here, a software design in translated into artificial language, which is executed by the computer. Figure 2.5 shows these four levels and the forward and reverse engineering processes.

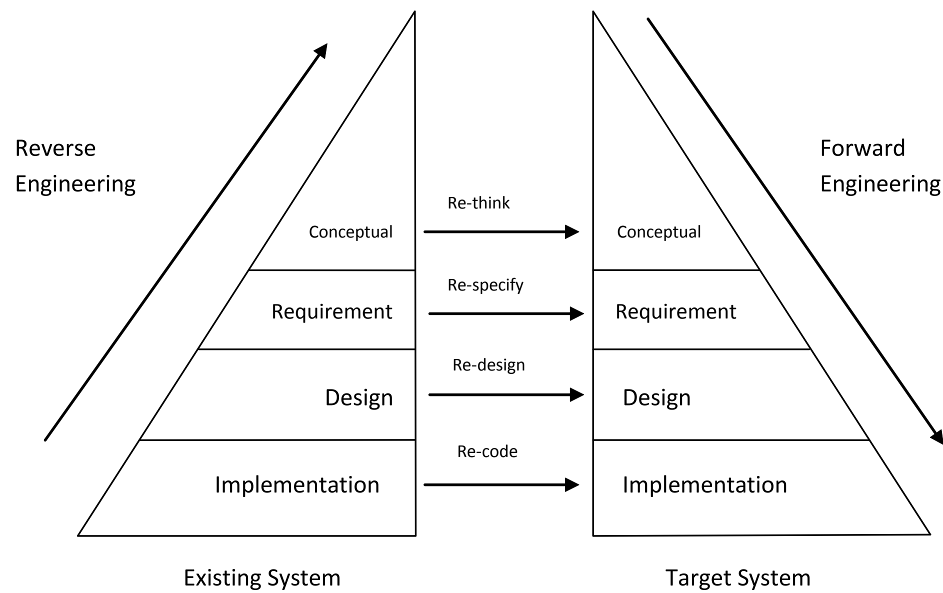

Figure 2.5: Re-engineering Classification and Software Abstraction [132].

2.2.3 Reverse Engineering

Reverse engineering is an important process in extracting useful information from the software (to be used in software maintenance or software reuse) [125]. The process of reverse engineering does not change or modify anything; however, it is a process that collects information for understanding in reverse order the traditional software development process.

Reverse engineering has been defined as "analysing a subject system to identify its current components and their dependencies, and to extract and create system abstractions and design information" [35]. The main aim of reverse engineering is to

transform a software system into a higher level of abstraction to reveal the software system components and their interrelationships [130]. The higher the abstraction of representation, the more easily the software system can be understood [158].

2.2.4 Software Evolution

'Evolution' as a term has been used in many domains to describe a common phenomenon, which is caused by continuing change in the evolving entity. Organisms, cities, ideas, concepts and almost everything is subject to the phenomenon of evolution within its own context [96].

In the context of software, evolution generally entails a slow process of incremental change, driven by the changing requirements of a software system [154]. Software evolution generally passes through a long process of discrete steps throughout the software lifecycle. This process consists of the execution, usage, enhancement, extension and updating of the software system in question. There are many and various causes in software evolution; they can be corrective actions (e.g., fixing defects), adaptive actions (e.g., adapting the changes in the operating environment) or perfective actions (e.g., improving performance) [133].

The term evolution is used in contrast to software maintenance. Software maintenance has negative connotations, i.e. the software is declining in terms of optimal functionality. Changes in the software environment or changes in stakeholder needs make it important for the software to adapt to the new situation [102].

To keep pace with rapid changes in the environment, new business needs and demands for new features, software systems must be in a state of continuous change if they are to remain useful [133].

In general, there are four cause of software change; these are corrective, preventive, adaptive and perfective. Corrective changes are needed to correct software defects. Preventative changes are needed to change the software to prevent potential

malfunctions. Adaptive changes are needed to adapt the software to changes in the hardware or system environment. Finally, perfective changes are needed to change the software system to improve its performance [159].

2.2.5 Laws of Software Evolution

Belady and Lehman were the first to study software evolution in a systematic manner (in the late 1960s). Their works in software evolution continued for more than a decade and they defined a set of laws pertaining to software evolution [13] [92] [93]. They identified three main laws in software evolution: the law of continuing change, the law of increasing complexity and the law of self-regulation. A further two laws were added to describe the limitation of software growth: the law of the conservation of organizational stability, and the law of the conservation of familiarity [91] [11]. By the late 1990s, another three laws were proposed in software evolution: the law of continuing growth, the law of declining quality, and the law of the feedback system [90].

Thus, the laws of evolution in software systems, which are usually called Leham's laws, are:

1. Law of Continuing Change: a software system must adapt to changes in the real-world environment or it becomes less useful and continues to fall below satisfactory performance.

2. Law of Increasing Complexity: as the system evolves to adapt to environment change, it become more and more complex. More work and resources are therefore needed to maintain simplicity.

3. Law of Self-Regulation: software system evolution is a self-regulating process, i.e. the system adjusts its growth over time.

4. Law of the Conservation of Organizational Stability (invariant work rate): during a system lifetime, the rate of productive output stays constant.

5. Law of the Conservation of Familiarity: the incremental growth of the system is constant. New software releases are always followed by smaller releases to fix problems in the initial release.

6. Law of Continuing Growth: software is in a state of continuing growth in order to satisfy the continuing change of user requirements.

7. Law of Declining Quality: over time, the software system quality declines unless it is continually under maintenance to adapt to new environment changes.

8. Law of Feedback System: in order to achieve improvements in software evolution, the system must be treated as a process of multi-level and multi-loop feedbacks.

2.2.6 Legacy Software System

'Legacy system' is a well accepted and a clear term in the software community nowadays, unlike a couple of decades ago. The software community is aware that new software systems quickly become legacy systems because of rapid changes in software requirements and the environment [159]. Legacy systems are generally those large software systems upon which a great deal of time and money has been spent by organisations. However, these systems remain critical to the organisation's main business. Unfortunately, these systems often resist software evolution processes. Thus, a legacy system may be an information system that resists change or evolution processes in order to meet new requirements [21].

Most legacy software systems are associated with old and large systems that

were developed and coded in early versions of 'third generation' languages such as COBOL or FORTRAN. Moreover, legacy systems were generally designed and built on inflexible architectures, and adaption to change or being able to update them were not in the mind of designers during software development [96].

The reason for a software application becoming a legacy system usually stems from software characteristics such as complexity [21], and the major factors in an organisation's decision to upgrade or maintain their legacy system are the costs entailed and the extent to which they can benefit from the investment.

In general, there are four options organisations in dealing with legacy systems [133]. The first one is to scrap the whole system and replace it with a totally new one. This first option is usually chosen when the system is not essential and the cost of the maintenance process can no longer be justified. The second option is keeping the system and maintaining it. This option is chosen when the cost of purchasing a new one cannot be justified; however, there are some benefits to keeping it for a small number of organisations. The third option is re-engineering the legacy system. This option is chosen when the cost of rebuilding the whole declining system justifies the cost of investment. The fourth and last option is replacing part of the legacy system. This option is chosen when only part of the system is causing the defect (possibly due to some change in the system environment). Table 2.1 shows and illustrates these options.

Legacy System Maintenance Strategy	Description
Scrap the system completely	The system is not essential to the organisation's business, or the cost of building a new system is less than the system maintenance.
Keep the system and maintain it	The system is stable and the need for it is limited to a small number of users. In this case, the organisation needs the system but the upgrade cost is not worth the investment.
Re-engineer the system	The system quality is declining due to regular changes. However, more changes are needed and the cost of rebuilding the system is worth the investment.
Replace parts of the system	Only parts of the system are causing the system defects (because of environment change, e.g. hardware upgrade). Sometimes, compromising some of the system's features is wiser than adopting costly maintenance.

Table 2.1: Legacy System Maintenance Options

2.3 Software Architecture

It is easier to understand a system once that system's individual parts have been decomposed, visualized and presented, revealing how those individual parts interact and function together. This problem of precisely how to present a system as a set of individual parts in such a way that it does reveal how they interact has been researched by several authors; however, each author tends to address only certain aspects of this problem [59][1]. Nevertheless, this high level view of a software system structure is generally described in the literature as software or system architecture [123].

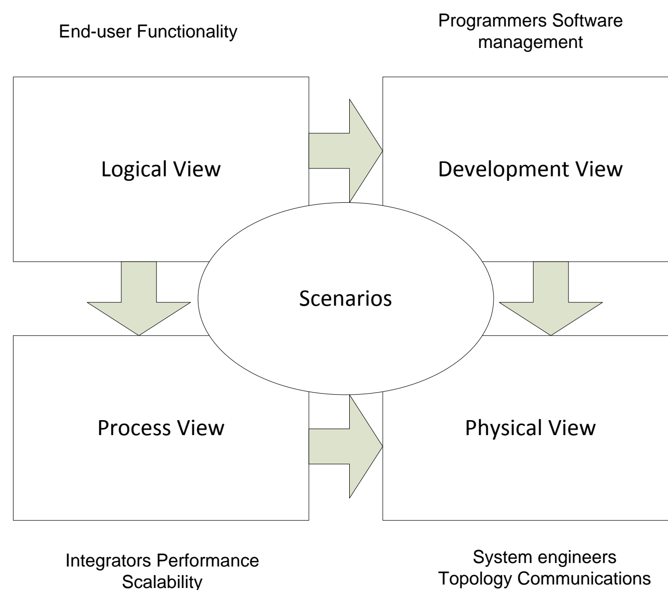

Figure 2.6: 4+1 View Model [87].

In Figure 2.6, the "4+1 view" model shows that system architecture consists of five main views [87]. These five main views are:

1. Logical view:

The logical view identifies the functional requirements of the system. The design model is the core of the logical view, where a description of the functional behaviour of the system is given.

2. Physical view:

   The physical view maps the software to the hardware. It contains non-functional requirements such as performance and availability.

3. Development view:

   The development view is a description of the actual software module organisation.

4. Process view:

   The process view represents all the processes and emphasizes the concurrency and synchronization aspects of the design.

5. Scenarios:

   The scenarios view integrates and validates the other four views by using use-cases.

The 4+1 view model provides a birds-eye view of a software system. It distinguishes and highlights all the software components as separate parts. This architectural view of a software system provides a better understanding of a software system as a whole but, when considering the 4+1 view model, the main target is to extract the logical view.

2.4 UML

2.4.1 Introduction

A model is a blueprint description of a system [17]. In software systems, the definition of a model refers to the modelling of a software process. Models simplify complex systems and make them easier to understand.

UML stands for Unified Modelling Language. It is a modelling language written mainly for software systems; however, the full scope of UML has always been the subject of debate. The UML specification document states, "The objective of UML is to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software based systems as well as for modelling business and similar processes" [39].

UML has been used for modelling system structure and behaviour. It is a language for visualising, specifying, constructing and documenting the artefacts of a software system. Nowadays, UML is considered the standard modelling language for writing software blueprints and it is commonly used for modelling a wide range of systems, from enterprise information systems to Web-based applications and embedded systems.

UML is popular because of its impressive ability to address all the perspectives needed to develop a system. When modelling large and complex systems, UML has proven to be the most successful engineering practice [57] [80]. The best-practice approach in UML is to model both a system's data and its processes, and this done by many professionals in the industry.

Because of the increasing popularity of the UML, many different CASE tools have been introduced to support it. These tools can generate source code in many different programming languages [105][30].

2.4.2 UML Concepts

What is now known as Unified Modelling Language (UML) is the product of integrated software modelling approaches conducted by Booch, Rumbaugh and Jacobson in 1995 [53]. UML is a visual language for modelling systems, and it describes in simple terms the meaning of the system's rules and the system design; it is not a stage within the system development lifecycle.

The Object Management Group (OMG) is a non-profit computer industry group started in 1989. It works on developing enterprise integration standards for a wide range of technologies for many industries. OMG states, "UML is a graphical language for visualising, specifying, constructing, and documenting the artefacts of a software-intensive system as well as for business modelling and other non-software systems. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components." The latest version of OMG UML, UML Version 2.4.1, was published in August 2011 [100].

Diagrams are not the only part of UML; however, they are viewed as the core part of the language. UML diagrams deliver the graphical representations of a system. There are two main groups in UML diagrams: structure diagrams and behaviour diagrams [146]. These two groups model the software system structures and behaviours. Structure diagrams represent the static architecture of the system's physical or conceptual elements. These structures can be classes, objects, interfaces, system physical components or the relationships among them. On the other hand, behaviour diagrams describe the software system's dynamic model. They represent the activities and interactions of the software system.

When UML was introduced in the late 1990s, there were nine diagrams for de-

scribing a software system. However, describing a software system was fraught with difficulties, and therefore UML 2.0 was developed, which had thirteen diagrams. Some of the UML 1.0 diagrams were abandoned and new diagrams were introduced [39]. UML 2.0 uses six of its thirteen diagrams to describe the static system structure. These diagrams are: Class diagram, Object diagram, Package diagram, Component diagram, Composite structure diagram and Deployment diagram. The other seven diagrams are for modelling the system behaviour. These behaviour diagrams are: Use case, Activity diagram, State machine, Communication, Interaction overview, Timing diagram and Sequence diagram. Four of the seven behaviour diagrams are sometimes referred to as interaction diagrams [3]. Figure 2.7 shows all the UML diagrams [115].

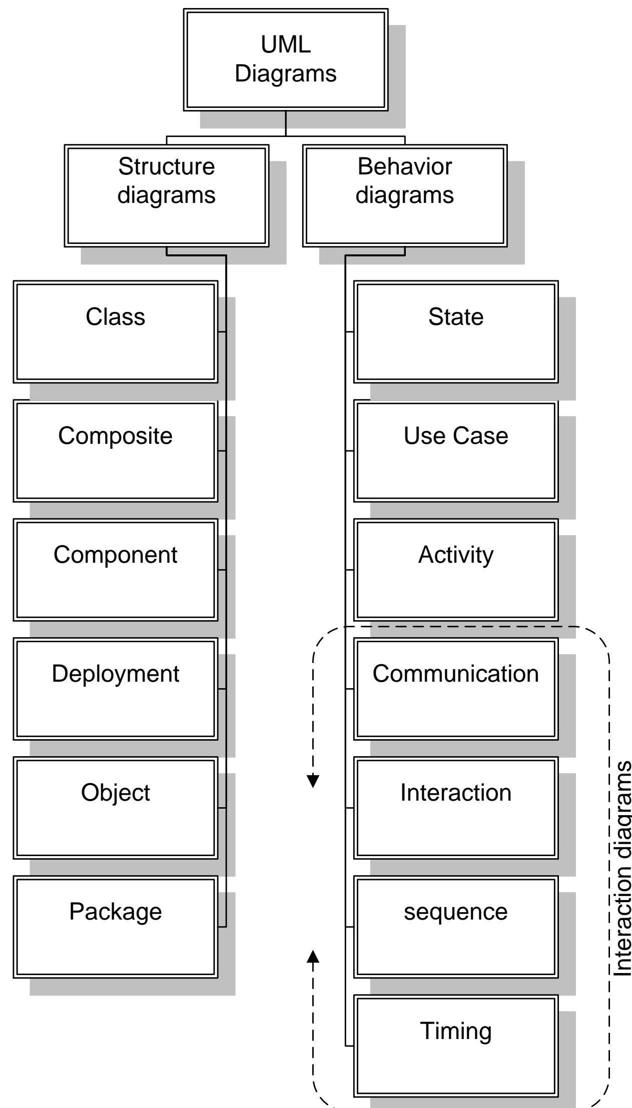

Figure 2.7: UML Diagrams

UML is mostly used in the design phase of a software development lifecycle. However, some of the UML models can be used in the early stages of software development. UML graphically shows a bird's-eye view of a system, and this big picture of the system allows the UML diagrams to serve as a universal communication medium for everyone in the software development team [112]. In most projects,

UML is used very early in the software development project; however, it can be used in other phases of the software lifecycle. An example is using UML in a software reverse engineering process, such as extracting UML diagrams from software source code [26]. Software engineers must understand the software system before they can re-engineer or integrate it [27]. Extracting UML diagrams from an old system can provide a great deal of assistance in fully understanding the system.

In summary, UML represents a system from two points of view: static and dynamic. It is important to know that UML is neither a system development lifecycle, nor a software process model; it is simply a notation. UML is a language for the visual representation of a project's requirements and desired design. This visual representation of a system ensures that there is one consistent model for a targeted system, and that all stakeholders can understand and provide feedback on a project in reference to the same model.

2.5 Program Slicing

Since Mark Weiser proposed the notion of program slicing in 1979, many researchers have conducted their work in this area [150] [157]. Program slicing is a program analysis and transformation technique. It is a technique for decomposing a program to extract statements with respect to some particular computation. It uses dependence information pertaining to program statements in order to identify all statements that are affected by (or affect) a specific point in the program. This specific point is called a slice criteria point [118].

When Weiser proposed program slicing twenty years ago, he defined it as "method for automatically decomposing programs by analysing their data flow and control flow. Starting from a subset of a program's behaviour, slicing reduces that program to a minimal form which still produces that behaviour. The reduced program, called a slice, is an independent program guaranteed to represent faithfully the original program within the domain of the specified subset of behaviour" [151].

A software contains thousands of lines of code, which can be sliced into small manageable segments to simplify their maintenance tasks.

In general, a slice consists of a set statements and a variable, $< s, v >$, where every statement of s has a direct or indirect affect on the value of the variable v [104].

Software slicing is used as a reverse engineering technique. It intensively used in analysing legacy software systems to extract codes that are related to a targeted software system. It is considered a particularly useful reverse engineering technique [63]. Software slicing in not limited to software reverse engineering tasks. Software engineers use it in many other software-related activities, such as software testing, software measurement, program comprehension, and software debugging [51] [148].

Since Weiser described the concept of program slicing, many slicing algorithms

have been proposed in the literature. These slicing algorithms can be classified into two main categories: static slicing and dynamic slicing [161][131]. Static slicing does not consider any particular input. It was called static because it does not depend on program execution. Dynamic slicing, on the other hand, depends on the program execution and on a given program input. Static slicing is ideal for understanding the impact of changing an existing system, e.g. in software maintenance [119]. Alternatively, dynamic slicing is ideal for software debugging tasks, as it depends on program data input.

In the context of static program slicing, there are three main approaches. The first uses data flow equations; this was first introduced by Weiser [151]. In this approach, slices are computed based on processes over a Control Flow Graph (CFG). This process uses data dependencies to compute sets of relevant variables for each node in the CFG [149].

The second approach uses information-flow relations [15]. To obtain slices, relational calculus is applied to several kinds of information flow in a syntax-directed bottom-up fashion.

The third approach uses Program Dependence Graphs (PDG); this is the most popular one. In this approach, a Program Dependence Graph (PDG) is constructed. A slice is produced by applying an algorithm to the PDG, and different slices can be produced based on the different PDGs constructed [55].

Program dependencies can be traversed forwards or backwards from a slice criterion; this is known as forward slicing or backward slicing, respectively. A forward slicing is a set of all statements that can be influenced by a particular slice criterion. In contrast, backward slicing is a set of all statements that could influence a particular slice criterion [51]. As explained in Chapter 5, forward slicing is the technique adopted in this research, and it will be discussed in greater depth in that chapter.

2.6 Requirement Engineering

Lack of common definitions for the terms used in the software industry represents an ongoing problem, and it causes confusion or misunderstandings. Many researchers may describe the same statement with different terms [153], and the term 'requirement' is no exception. It has been described in various ways; the IEEE (Institute of Electrical and Electronics Engineers) Standard Glossary of Software Engineering Terminology defined the term in 1990 thus [71]:

1. A condition or capability needed by a user to solve a problem or achieve an objective.

2. A condition or capability that must be met or possessed by a system or system components to satisfy a contract, standard, specification or other formally imposed documents.

3. A documented representation of a condition or capability as in (1) or (2).

This definition of requirement combined both user and developer perspectives. However, the term stakeholder should be used instead of user, as not all stakeholders are users. A stakeholder is "an individual, group of people, organization or other entity that has a direct or indirect interest (or stake) in the system" [67].

Requirement can be viewed at a high level of abstraction as the services that a system should provide [133]. On the other hand, at low level of abstraction, requirement can be viewed as detailed, formal descriptions of the system functions [44].

In any software development process, the first step is to identify and elicit the software requirements. This step is not only the first step, it is the most important step in the software development lifecycle. A correct, comprehensive and accurate

requirement specification is a must because the whole process is based on model transformation. Therefore, if the original model requirements are less than satisfactory, the final product will be so as well, and the project is doomed [128]. Unfortunately, understanding and designing a software system is a difficult and complex task.

Brooks, in his famous article No Silver Bullet: Essence and Accidents of Software Engineering, illustrates why developing a software system is so hard [23]. He classified the software development process into two main classes: essential qualities and accidental qualities. The essential qualities are extremely difficult; these are: complexity, invisibility, changeability and conformity. The accidental qualities relate to implementing and testing, and the problems relating to these tasks are likely to be solved with greater ease, given sufficient technical expertise [98].

As with many terms in the software industry, there are many definitions for the term 'requirement engineering' [110]. A clear definition of software requirement engineering was given by Zave [160]: "Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behaviour, and to their evolution over time and across software families." Requirement engineering refers to this first and most important stage of the software development lifecycle.

The term 'system requirements' is sometimes used to mean software requirements. However, these two terms are not the same. Firstly, the term 'system' originates from the Greek *systēma* (Latin systema), which means 'to place together', 'to combine' or 'to organize as a whole' [16]. Therefore, the term system requirement should be used to describe high level requirements, which contain software, hardware, other subsystems or even people. In the software requirement process, there are three separate levels that need to be extracted and analysed [153]. These

three levels are the business level, the user level and the functional level, as shown in Figure 2.8.

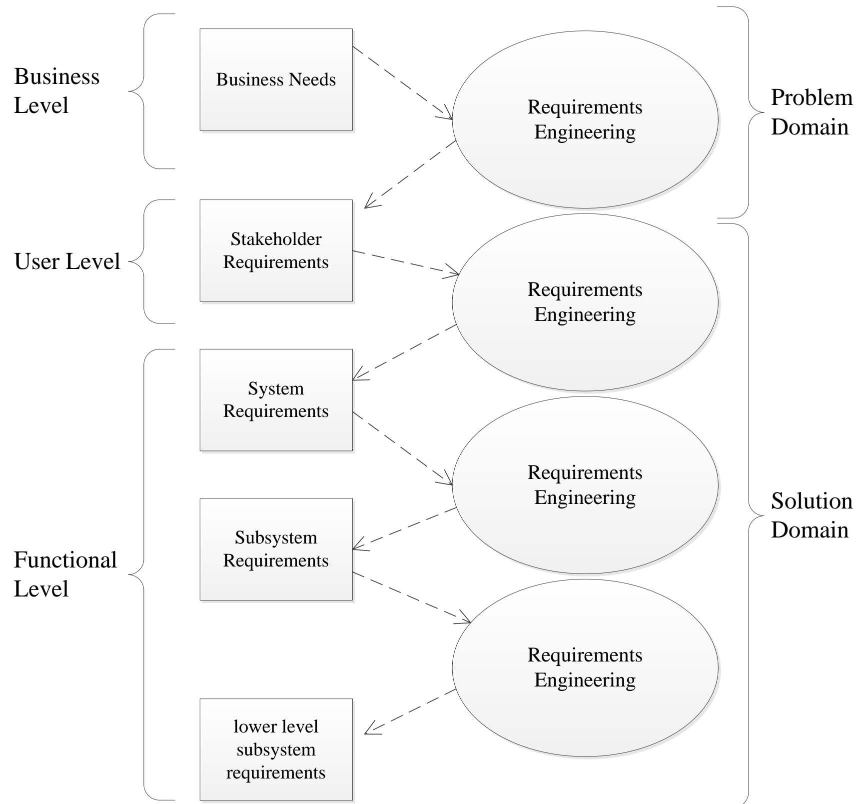

Figure 2.8: Levels of Requirements Engineering [66].

The business level represents the high level objectives of the users or organization that needs the system. The business level describes the business needs and what it is that the system is expected to achieve in order to satisfy these needs. User requirements represent the tasks or goals that the system user must be able to perform. User requirements are usually expressed in natural language, such as scenarios and story cards, or as diagrams such as 'use case'. The third level represents the functional requirements. These requirements are all the functionality that the

system must include to enable the users to achieve their goals in order to satisfy the business requirements.

In addition to these levels, every system has a collection of non-functional requirements. These non-functional requirements include performance goals and quality attributes. These attributes include portability, performance, security, usability or robustness [36].

The functional and non-functional requirements are documented in a software requirement specification (SRS) package. The SRS thus contains a full description of the system and all the behaviour expected from the system. Therefore, the SRS is used in the whole software system development lifecycle.

The process of requirement development is subdivided into four stages, which are elicitation, analysis, specification and validation [52]. Inside each of these, many and various techniques can be utilised; the choice of technique depends on the resources and time available to the software engineers. Every requirement elicitation technique has its strengths and weaknesses [77].

2.7 Model Comparison

Model comparison is a technique in Model-Driven Engineering (MDE). MDE is the systematic use of high-level software models as the main artefacts during a software engineering process [68]. Model comparison is a technique used to identify the similarities or differences between any two models. It is also used for model versioning and model clone detection (described below). Moreover, it is used in many other MDE areas and it is the basis of other modelling techniques such as model composition and model transformation testing [134].

Kolovos, Paige and Polack describe model comparison as a process that distinguishes elements into four groups [84]:

1. Elements that match and conform
2. Elements that match and do not conform
3. Elements that do not match and are within the domain of comparison
4. Elements that do not match and are not within the domain of comparison

Matching means that elements have the same idea or artefact, while conformance refers to additional matching criteria. For example, a non-conformance UML class diagram is when a class in two models has the same name but one is abstract; although they probably represent the same artefact, they do 'sufficiently match' or conform to one another [84].

2.7.1 Model Comparison Phases

To solve the problem complexity in model differentiation, Brun and Pierantonio decomposed model comparison into three phases [24]:

1. Calculation: this is the initial step in model comparison. Calculation can be an algorithm, method or procedure that is able to find similarities or differences between two models.

2. Representation: this phase takes place during the calculation phase, i.e. when the differences and similarities are detected. It is a form of presentation relating to the outcome of the calculation phase. One approach used in representation is the notion of edit scripts [4] [101]; these are an operational representation of the changes needed to make one model equivalent to another, such as add, edit or delete.

3. Visualization: model similarities and differences need to be visualized in an end-user, human-readable notation that allows the designer to see the reasons behind the differences or similarities between models; for example, visualizing model-based representation through colouring [109]. Visualization is closely related to representation; however, recent visualization approaches attempt to distinguish visualization from representation [144][152].

2.7.2 Model Versioning

As in traditional software projects, teamwork in MDE projects is crucial. Version Control Systems (VCS) is used to achieve such cooperation. VCS enables software developers to keep previous versions in what is generally called a repository. The repository keeps files and project structures that are under development stored in order to be able to retrieve past versions.

In MDE projects, it is necessary that developers are able to work separately but at the end able to reintegrate updated versions into the final project. Unfortunately, models in traditional VCS do not work well [6].

Model versioning passes through different phases. The number of phases is different for different researchers [4][6]; however, it is clear that there are needs for:

1. Comparison or matching that identifies which model elements match other model elements.

2. Detection of model differences or similarities.

3. Representation of any model differences and model merging that highlight changes or conflicts.

2.7.3 Model Clone Detection

Model clone detection is another practice in model comparison. Model clone detection is related to a traditional software maintenance problem called code clone or source code cloning. "A code clone is a code block in source files which is identical or similar to another code block" [107]. Many approaches, techniques and tools have been proposed and developed to deal with code clones [122].

Model cloning is similar to code cloning; however, it involves a higher level of abstraction. Model clone is a term referring to groups of model elements that share certain similarities [135].

2.7.4 Model Comparison Approaches

Existing model comparison methods can be categorized by the types of models they compare; yet, some methods claim they can work with different types of models. Some of the categories in model comparison approaches are [135]:

1. Methods for Multiple Model Types: this first category of model comparison approaches represents those approaches that are able to work with more than

one type of model, such as structural models and behavioural models. Examples of these models are: UML Models, EMF Models and Metamodel-Agnostic Approaches [5].

2. 2. Methods for Behaviour/Data-Flow Models: this second category of model comparison approaches deals behaviour or data-flow models. Examples of these models are: Simulink and Matlab Models, Sequence Diagrams and Statechart Diagrams [45].

3. Methods for Structural Models: these types of models represent the structure of a system; for instance, UML Structural Models and Metamodel-Agnostic Approaches [156].

4. 4. Methods for Product Line Architectures: this approach to comparison is mainly for product line models (for merging). This approach assumes that the comparison is being done between two different versions of the same artefact [33].

5. 5. Methods for Process Models: this approach deals with the differences between software development process models and outlines; for example, node matching, which compares process element labels and attributes, structural similarity, which compares labels and topology, and behavioural comparison, which evaluates labels in conjunction [49].

2.8 Conclusion

This chapter (Feature-based Approach: State of the Art) is a literature review, which has studied researches related to the proposed feature-oriented framework. The feature-oriented framework makes connections between the features relating to business needs and the features relating to implementation. The proposed framework focuses on software system understanding and software system re-engineering. Therefore, most of the concepts covered and studied in this chapter relate to these concepts.

The first section provided a background to the concept of feature in the software development context. It proceeded to illustrate the concept of feature-oriented software development (FOSD). Feature in software development has various definitions, and most these definitions were explained and discussed in this first section. The FOSD process consists of four phases: domain analysis, domain design and specification, domain implementation, and product configuration and generation. Following this, feature modelling and feature-oriented domain analysis (FODA) were addressed. Feature interaction and feature implementation were covered in this section as well.

The second section covered software evolution and software re-engineering. This section and all the sections that followed it studied program understanding. These studies included program reverse engineering and program slicing. UML conceptual system modelling was discussed in the UML section, and finally, requirement engineering and model comparison were studied in the last two sections.

Chapter 3

Proposed Approach

Objectives:

- To introduce the stages of the feature-oriented business IT framework
- To give an overview of the steps within each stage
- To highlight the focus of this research in each stage of the framework

---

3.1 Framework Overview

The term 'framework' has been used in many different ways in the literature. Dictionaries can provide a basic understanding of the meaning as well as synonyms of this word, and two definitions have here been taken from two different dictionaries: Dictionary.com and TheFreeDictionary.com.

Framework: [48]

1. A frame or structure composed of parts fitted and joined together.
2. A skeletal structure designed to support or enclose something.

Framework: [139]

1. A set of assumptions, concepts, values and practices that constitute a way of viewing reality.
2. A particular set of beliefs, ideas or rules referred to in order to solve a problem.

In the discipline of computer science and particularly in its sub-discipline software engineering, a framework presents an abstraction or a bird's-eye view of a solution to a number of problems that have similarities. It guides and assists in drawing a map of the steps needed in order to find a solution without going into the details of these steps or the activities they entail.

The framework introduced in this research can be seen as a roadmap for software engineers targeting a linkage between technical-oriented software features and problem-oriented domain features. This framework approach is comparable to the first dictionary definition "composed of parts fitted and joined together".

The proposed framework consists of three main phases. The first is a top-to-middle process and the second is a bottom-to-middle process. These two processes meet at the third phase, which is in the middle of the framework. Each phase consists of a set of steps, which must be satisfied in order to achieve the final goal of the phase's purpose. In other words, these phases map views of the software features from the business perspective, which is the top-to-middle process, and the technical perspective, which is the bottom-to-middle process. Figure 3.1 provides an overview of the proposed framework.

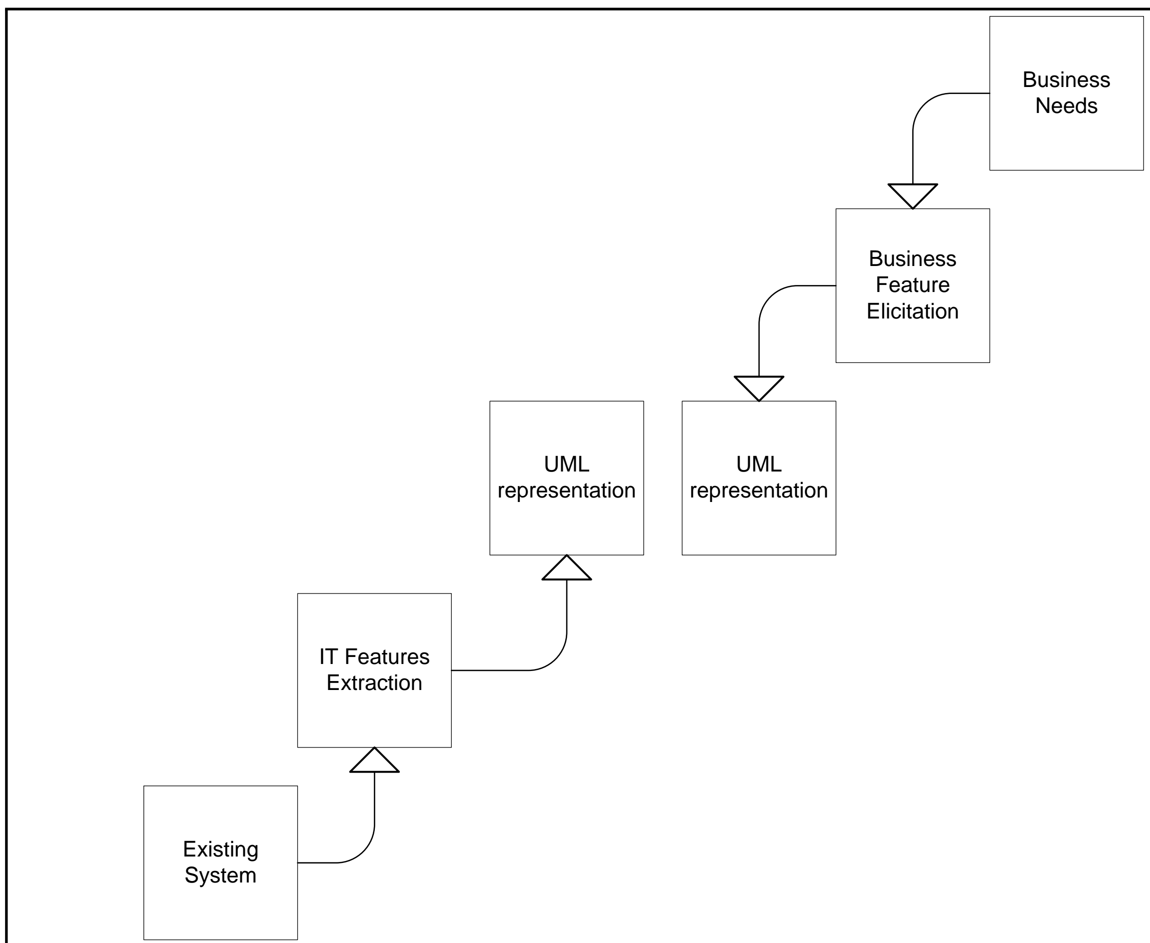

Figure 3.1: An Overview of The Proposed Framework

This section now gives an overview of the steps within the phases of the proposed feature-oriented IT business framework. As shown above in Figure 3.1, it has two main phases (bottom-to-middle and top-to-middle); however, a third phase is evident, and this can be seen as the mapping of the first and the second phases.

Generally, the first phase is a forward engineering software development phase. It begins with the first step of any software development process; namely, identification of business needs (which is a business analysis process). It proceeds to identify the features needed to satisfy these needs, and finally represents these features using a conceptual data model language.

On the other hand, the second phase is a reverse engineering process. It begins at the lowest level of the software development lifecycle; namely, the software source code. It is well known that information from the source code is the most reliable information, as all the system features must be presented in the final software system source code [64]. Program slicing, which is a reverse engineering technique, is used to extract all statements that are related to a specific software feature. At this lower level of software development, a software feature is defined as "an increment of program functionality" [143]. Figure 3.2 shows the feature-oriented IT-business framework in more detail.

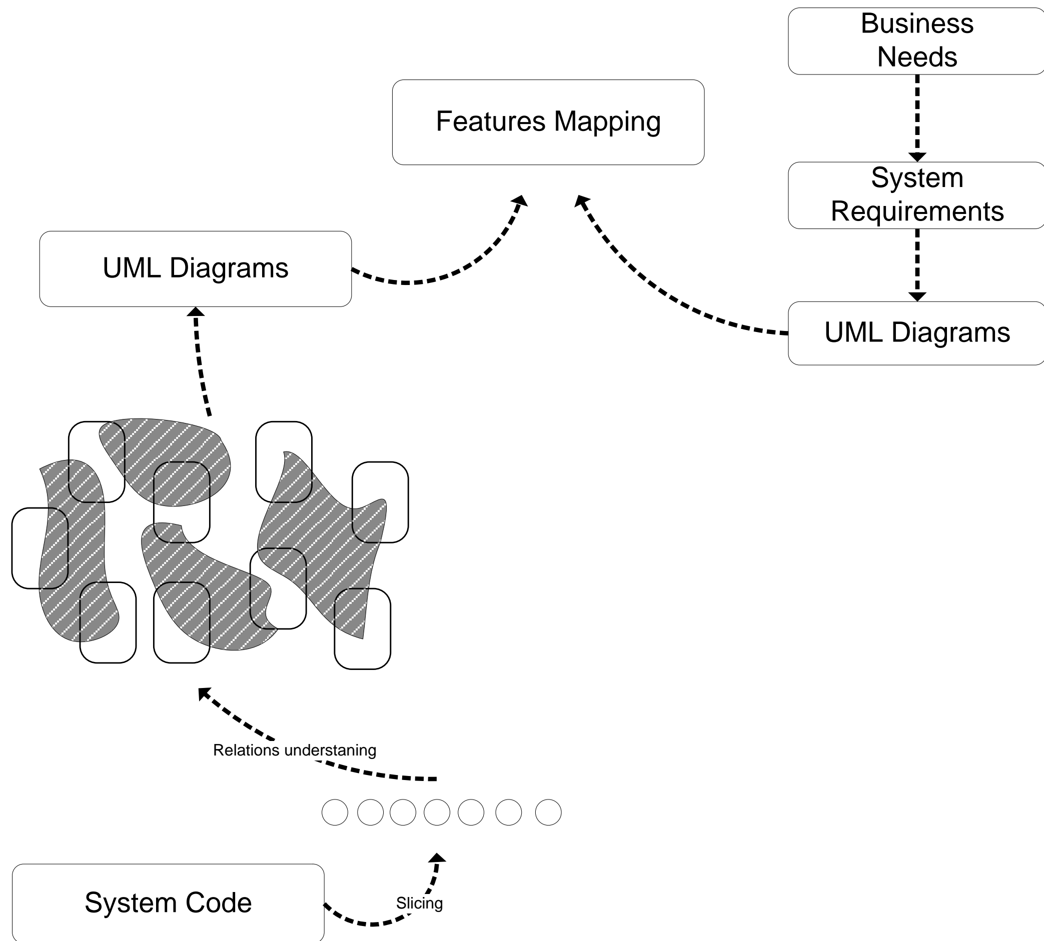

Figure 3.2: Feature-Oriented IT-Business Framework

From the top-to-middle phase and the bottom-to-middle phase, conceptual models of the features are produced. These models are mapped to find feature similarities in the third phase, as shown in the figure above. Thus, the proposed feature-oriented IT-Business framework consists of three main phases:

1. The Business Feature Elicitation Phase:

   The business feature elicitation phase is the top-to-middle process, as Figure 3.2 presents. It begins with the process of business analysis. This process deals with identifying business needs or clarifying the business problems; it then provides solutions to these needs or problems. These solutions are doc-

umented as the business requirements or the features needed to resolve the problems. Once the business needs have been identified, the next step is to pass these to the requirement engineer. Requirement engineering elicits the system requirements and documents these requirements or necessary features in a 'technical requirements document' called the Software Requirement Specification (SRS) file. These technical requirements are transferred to the next level in the software development lifecycle process, wherein these requirements are represented as data and behaviours in the form of conceptual modelling.

2. IT Feature Extraction Phase:

   The IT feature extraction stage is the bottom-to-middle phase, as shown in Figure 3.2 It begins with the system source code and then extracts software features, which are later represented in conceptual modelling presentations. The IT feature extraction phase is a reverse-engineering process. Program slicing, which is a reverse-engineering technique, is used on the software source code for program comprehension. Firstly, the software system code is sliced into a set of related statements. Secondly, the slices resulting from the program slicing technique are represented in program dependency graphs (PDG). These high abstractions of model representations contain potential system features or system functionalities. Finally, the PDGs are transformed into a conceptual modelling presentation.

3. Model Matching Phase:

   The final phase of the feature-orient business IT framework is located in the middle. It takes the product of the two other phases and links them together. The products of the two previous stages are sets of features presented in a modelling language. Therefore, model comparison techniques are used in this

phase to find similarities between the two models that were generated in the previous phases.

The novelty of the proposed framework emanates from the three characteristics of the business-IT gap: the problem domain, the solution domain and the matching process. The problem domain part is covered in the first stage of the framework, The Business Feature Elicitation Phase. The problem solution domain is covered in the second part of the framework, IT Feature Extraction Phase. And finally, the matching process is covered in the Model Matching Phase.

More details of these stages will be illustrated in next chapters. Chapter 4 will discuss the business feature Elicitation stage. The IT Features Extraction Stage is illustrated in chapter 5. Finally, chapter 6 illustrates the model matching stage.

Figure 3.3 provides detailed information on the phases of the proposed framework as well as the steps in each.

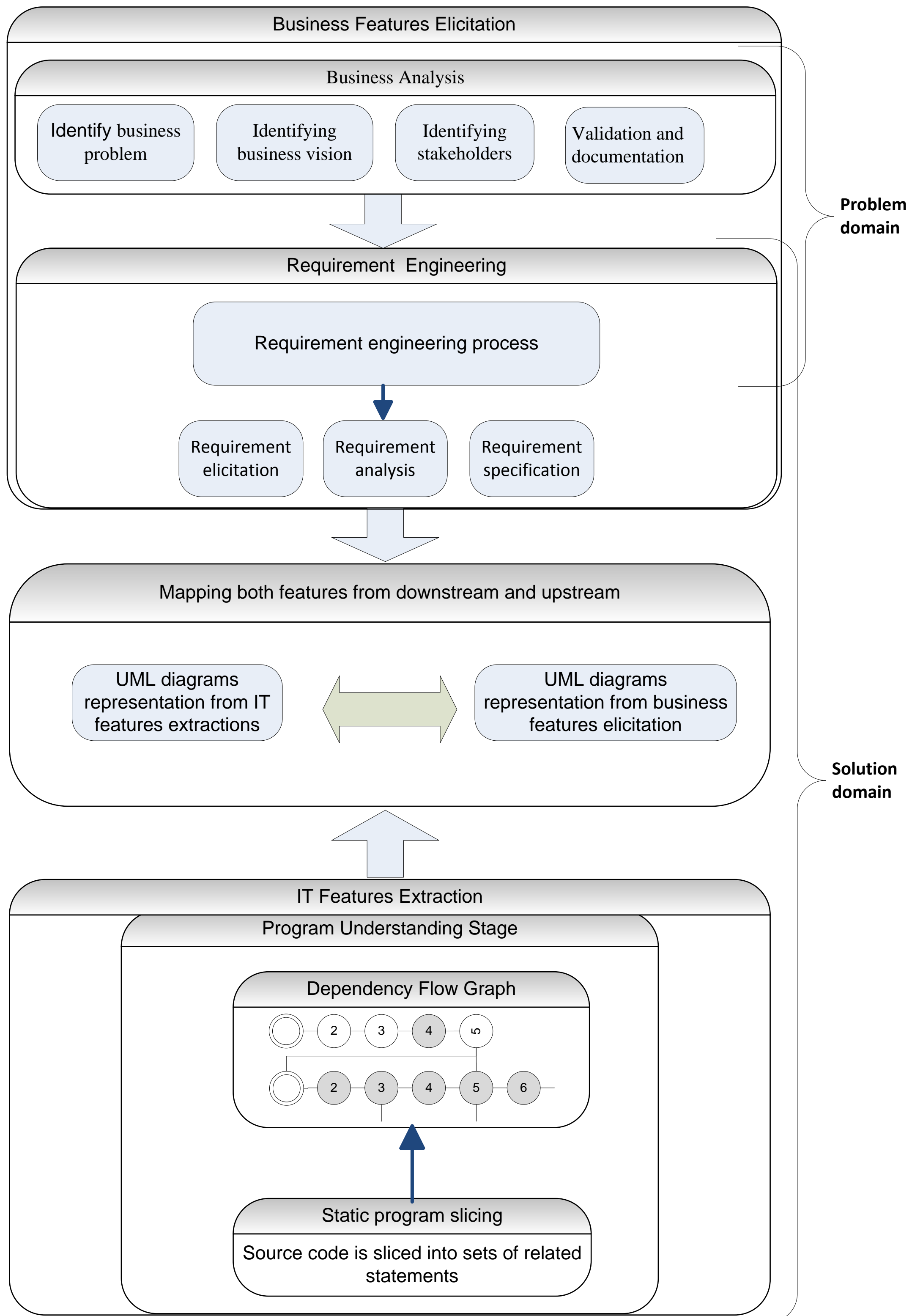

Figure 3.3: Framework Stages and Steps

3.2 Business Feature Elicitation

The first step for any business software system begins with identifying the business needs, problems or business change. The business analysis process seeks to fully comprehend the organization's current state as well as its particular business needs, the full range of current problem and the drivers of change [72]. Thus, this analysis process involves: clarifying the business problem, identifying the vision, identifying stakeholders and documentation. The business vision or strategic mission is interpreted as business goals or objectives. These business objectives are implemented as business functions and processes. All these activities are carried out by a set of processes within business analysis, all of which are within the context of the business domain.

In the context of software engineering (as covered in the literature review chapter), the process of identifying, modelling, communicating and documenting the software system requirements is a Requirement Engineering (RE) task. A software requirement task describes what is to be done but not how it is to be done [111]. Thus, RE is a set of processes that describes what needs to be done to solve the problem under investigation. The main aim of this step is to identify business needs and services, and then to decompose these services into a set of features that can be modelled; it is a decomposing process.

Business tends to view system services as set of system features that can satisfy business needs. These features are elicited as stakeholder requirements, which the system must contain in order to satisfy the business needs. The RE process elicits the business requirements and clusters all the features into sets of user and system requirements. Finally, all the sets of requirements (which represent features) pass through the software development process to be represented as conceptual model diagrams.

As evident in Figure ??, this phase can be divided into two main stages: the business analysis stage and the requirement engineering stage. Business analysis and part of the RE process are within the problem domain; however, the other part of the RE is within the solution domain. In accordance with the limitations of this research, most of the research work focuses on the requirement engineering stage only.

3.2.1 Business Analysis

Business analysis is set of tasks and techniques used to identify business changes, problems, opportunities or needs, and then it seeks to find solutions for them. A business analyst creates a network among business stakeholders in order to understand and analyse the business requirements needed for the business in question, and the best solutions available. Once these business requirements have been collected, they must be validated through the business processes, polices and information systems.

In general, the business analysis process passes through a set of steps, which are:

1. Understand how the organisation functions as well as its current state.

2. Define the organisation's goals and objectives, and determine how it achieves them.

3. Identify organisational units and any stakeholder interactions (within and outside).

4. Conduct validation and documentation.

3.2.2 Requirement Engineering

The Requirement Engineering (RE) process starts after the business needs and problems have been identified. It is a set of processes that are involved in developing the system requirements. RE consists of requirement elicitation, requirement analysis, requirement specification and requirement validation.

1. Requirement Elicitation:

   Elicitation is the process of interacting with stakeholders to capture their needs.

2. Requirement Analysis:

   The requirements collected from the stakeholders are analysed in detail to produce a collection of consistent, unambiguous and complete requirements.

3. Requirement Specification:

   A description of what must be done.

4. Requirement Validation:

   The requirement collection outcome must satisfy the stakeholders' needs.

3.3 IT Feature Extraction

The IT features extraction phase in this feature-oriented IT-business framework works to extract software system features from the software system source code. The process of extracting knowledge from software is a 'program understanding' process. It consists of source code program slicing, Program Dependency Graph (PDG) representation and, finally, transforming the PDG into conceptual modelling diagrams.

3.3.1 Program Understanding

The main concept of program understanding is building a knowledge base that represents the program. Rugaber's definition of program understanding is, "...the process of acquiring knowledge about a computer program. Increased knowledge enables such activities as bug correction, enhancement, reuse, and documentation" [38]. Software understanding is usually a reverse engineering process; it is about building knowledge from software code.

Reverse engineering practise usually starts at the system source code level. It takes software system code as an input and transforms it into higher level of abstraction. Reverse engineering techniques analyse and identify the system's components and the relationships among them. From these relationships, the reverse engineering process creates higher representations of the system based on these relationships.

3.3.2 Program Slicing

In the literature review chapter, program slicing was reviewed and it was described as a technique for decomposing programs by analysing their control flow and data. It is usually employed at the source code level as a reverse engineering and transformation technique. It is also well known to be one of the main applications in program comprehension.

Software slicing is used in this research (and in this phase) to extract statements related to program functionalities. This technique is considered to be a reliable source of information pertaining to a software system.

Program slicing has been widely studied since it was first presented. By definition, program slicing is a technique for locating all statements (called slices) that can affect or be affected by a critical point. The critical point is usually a variable or a statement within a program. Many researchers and programmers use the program

slicing technique as a tool to locate features implemented in a software system [10] [85] [86] [97].

Thus in this phase, program slicing is used as a method for extracting statements that are relevant to certain system behaviours of interest. It is a transformation method that takes the program source code as an input and extracts statements that have relationships among them. These groups of statements have the potential to represent a software system's features.

3.3.3 Program Dependence Graph

The second step of the IT feature extraction phase in the proposed framework is taking the product of the program slicing step into a higher level of abstraction. A Program Dependence Graph (PDG) is a graph in which each statement is represented as a node, and edges are represented as the possible flow of the PDG. More details about the PFG process are presented and explained in the IT feature extraction chapter.

The outcome from the previous step (the source code program slicing) is taken as an input, analysed and transformed into a visual representation as a PDG. Rules are proposed on producing candidate software system features. Some of these features are independent sets of program codes within a PDG.

From the PDG and the program slicing steps, all potential software system features are extracted. These extracted features are reverse engineered and represented using a conceptual data model presentation.

3.4 Feature Model Matching

The feature model matching phase is the final one; it is located in the middle of the proposed framework (as shown in Figure 3.3). It aims to match features represented

as UML diagrams from the previous two phases (business feature elicitation and IT feature extraction). This phase aims to reveal implemented features or missing features. Model comparison approaches are employed and a matching algorithm is proposed to perform the model matching. Model comparison is the practice of identifying the similarities and differences between two model's elements. As will be illustrated in the model matching chapter, a feature might be represented as a single model or a set of features might be embedded within a single model.

The matching algorithm proposed in this research is based on a UML class diagram. It takes all the elements of each class diagram and generates a set of class elements; for example, two classes represented as two sets where each set contains elements of a class. An element can be a class attribute or a class operation. Finally, the two sets are matched for similarities.

Thus, feature model matching involves Model Driven Engineering (MDE) and model comparison approaches, employing a matching algorithm.

The algorithm states that Model $v1$ is matched to model $v2$ if all element of $v1$ exist in model $v2$. It's also true the other ways around. More examples and details will be illustrated in chapter feature model matching.

3.5 Conclusion

In this chapter, firstly, an overview of the proposal framework was given. The feature-oriented IT business framework aims to address the need to link business needs features with IT software features. This framework organises this task in three main phases. The first is business feature elicitation, which is a forward engineering process. The second is IT feature extraction, which is a reverse engineering process. The third and last phase is feature model matching, which is a model matching process.

In every phase of the framework, a number of steps are followed in order to achieve the final goal of each phase, which is presenting software features as UML models.

Chapter 4

Business Feature Elicitation

Objectives:

- To highlight the importance of the business feature elicitation stage
- To define related terms
- To describe the business analysis steps
- To describe Requirement Engineering (terms and methods)
- To present the UML-based feature modelling

4.1 Overview

The business feature elicitation stage begins with the business analysis process. Business analysis deals with identifying business needs and seeks to provide solutions for these needs. The outcome of this process is some form of documentation, which states the business requirements or features needed for the problem solution. This documentation passes though a Requirement Engineering (RE) process. The RE process elicits system requirements and then documents these requirements or necessary features in a requirement document called a Software Requirement Specification (SRS) file. The SRSs are transferred to next level of the software's development lifecycle and are represented as data and behaviour conceptual modelling.

Figure 4.1 shows the decomposing of needs into the features needed and thence into requirements.

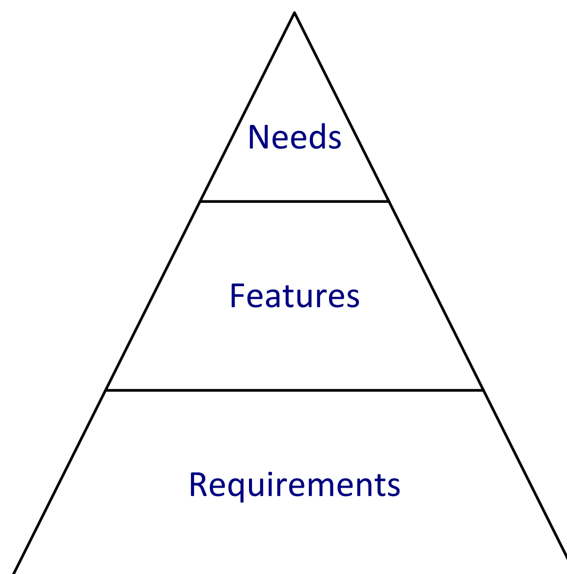

Figure 4.1: Needs, Features and Requirements [89]

4.2 Business Needs and Business Analysis

Business must be able to adapt to its current business environment if it is to achieve its main goal, which is generating (and increasing) business profits. Business uses a business analysis methodology to process and analyse any changes in the business needs or business environment. Business analysis is a set of processes and techniques used to study a organisation's policies and structures, and makes suggestions for achieving the business goals in a better manner [19].

One of the most important functions of a business analyst is to identify business changes and to determine what is needed to adapt to these changes. He or she needs to elicit new requirements and pass them to the IT department (if it is part of the solution). However, the job of the business analyst does not end with passing the new business requirements to the IT department; he or she should be involved in the process of identifying the most efficacious solutions until final implementation.

Figure 4.2 illustrates the relationship between the business analysis process and the software development process.

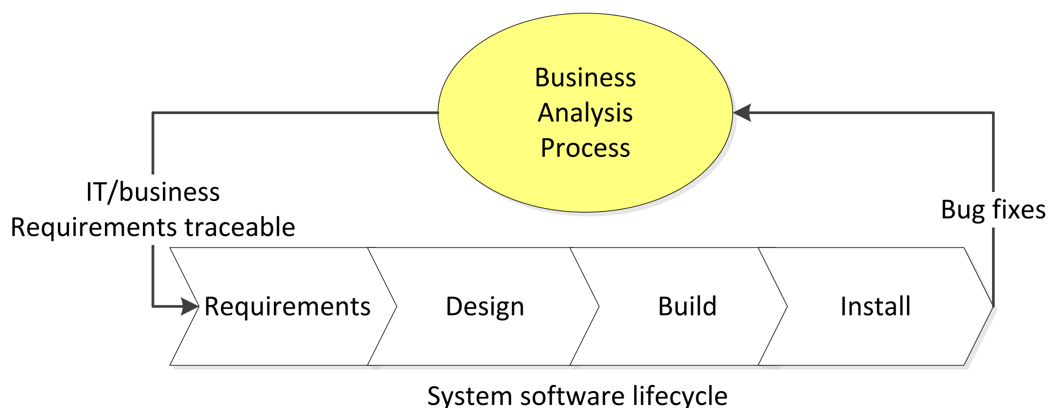

Figure 4.2: System Software Lifecycle

Thus, the first step in any software development is the process of identifying business needs. This process involves clarifying the business problem, identifying the organisation's vision or strategic mission and identifying stakeholders; this is followed by validation and documentation. All of these activities are business analysis activities. Business analysis is a set of tasks, involving the building of knowledge and employing techniques to comprehend the need for business change, identifying the impact of those changes, and generating solutions for the business problem [155]. Figure 4.3 illustrates the business analysis process.

Figure 4.3: Business Analysis Process [19].

From the figure above, identifying the business needs during business analysis process involves the following four steps:

1. Enterprise Analysis:

   Enterprise analysis describes the general business problems, opportunities or needs. It defines clearly all the needs and establishes a solution scope. The business needs should be aligned with the strategic mission of the business. If the new business needs or opportunities involve replacing part of the business system, the processes involved in that part must be clearly understood and form part of the requirement documents.

2. Elicitation:

Elicitation is a process of interacting with all stakeholders in order to explore, understand and define their needs and problems clearly and completely. These interaction activities must be all documented.

3. Requirement Analysis:

   Requirement analysis describes the steps needed to build the solution that is to satisfy the business and stakeholder needs. The requirements collected must be correct, validated and acceptable to all stakeholders.

4. Solution Validation and Documentation:

   All solutions must be validated to ensure that the requirements are acceptable and that the business needs will be satisfied.

4.3 System Features

Describing a product as a set of features that the system can provide is common in business and IT fields. Therefore, it is natural to think about the features needed in a system to fulfil business needs. Features fit in between the users' descriptions of real needs and the detailed descriptions of the system requirements for satisfying those needs, as shown in Figure 4.1 above.

One of the many advantages of describing a system as a set of features is that it provides initial system boundaries; this is because features describe a system at a very abstract level. Features describe what the system is capable of doing as well as what it is not able to do. Thus, most business analysts talk about a product's features in order to describe what that product is capable of achieving. In this sense, a feature is a set of related functions that enable a user to satisfy a business objective. At this level, feature definition is at the highest level of abstraction,

which can be defined as a service that that system provides to fulfil one or more stakeholder needs.

A product's features are used to describe the system in natural language with terms that are easy to understand for stakeholders. For example:

- The ATM system enables users to withdraw money.
- Bank customers are able to view their balance.
- The system displays a Welcome message to users.

To describe system features, software engineers use many techniques to present what a system is capable of, and the most famous of these techniques is the UML use case diagram. An example of a use case diagram is shown in Figure 4.4.

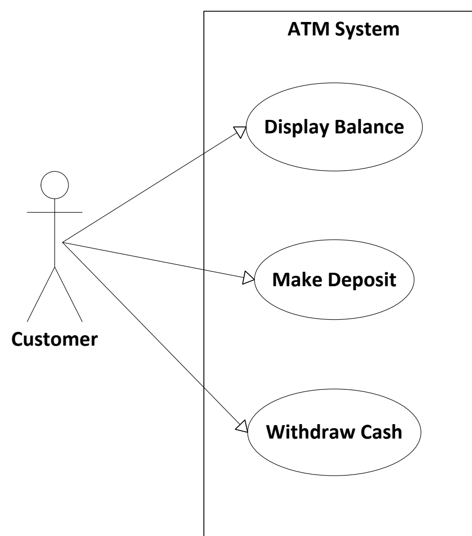

Figure 4.4: ATM Use Case.

The use case describes how the system's user interacts with that system to accomplish a particular task. It also describes the system's boundaries. The system boundaries define what is expected from the system and what the user can expect from that system.

The use case technique has been employed and developed in many researches, and it is now standard practice in UML methodology [73][17]. It can be used as a tool for explaining how the system features relate to the purpose and functions of that system in a brief and effective manner.

4.4 Requirement Engineering

Sommerville asserts that Requirement Engineering (RE) consists of four distinct steps [133]:

1. Requirement Elicitation
2. Requirement Analysis
3. Requirement Specification
4. Requirement Validation

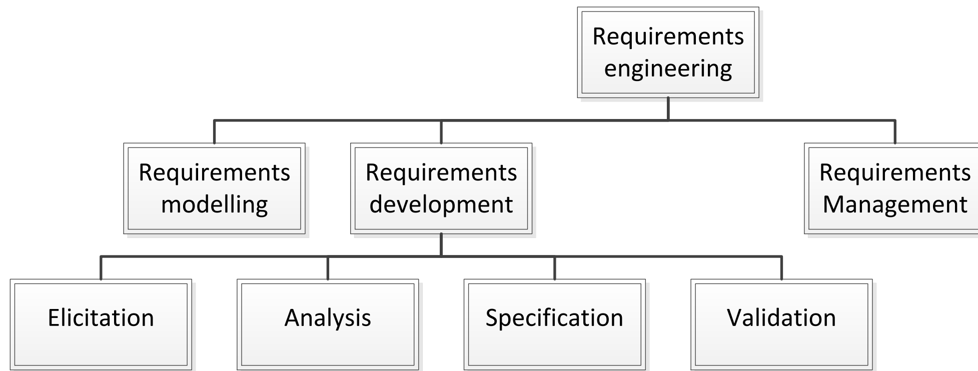

Figure 4.5: Components of Requirement Engineering Domain.

Two additional components are evident in the literature: requirement modelling and requirement management [153][20]. Figure 4.5 shows all the components of the requirement engineering domain. Requirement elicitation is briefly described in subsection 4.4.1. Subsection 4.4.2 is about requirement analysis. Requirement specification is briefly illustrated in subsection 4.4.3. Finally, requirement validation is in section 4.4.4.

4.4.1 Requirements Elicitation

The first step in the RE process is conducting requirement elicitation. Elicitation is the process of interacting with stakeholders to capture their needs. Requirements elicitation can be difficult, time consuming and represent a heavy demand on resources. The term requirement capturing is sometimes used as an alternative for requirement elicitation. However, the term 'elicitation' is usually preferred to 'capturing' because the latter implies that the requirements are already there and can easily be collected by asking questions.

Requirement elicitation concerns discovering the system's requirements by con-

tacting stakeholders, studying system documentation, analysing the market and exploiting domain knowledge.

The term 'system stakeholder' is used to refer to anyone (a person or a group) that might affect the requirements of the system under investigation whether directly or indirectly. The stakeholders' needs must be elicited in order to obtain the system requirements [58]. End-users, managers and software engineers are all typical stakeholders but other parts of the organisation (that may occasionally interact with the system) can also be considered stakeholders. Furthermore, external organisations can form part of the system stakeholders.

The first goal of the requirement elicitation process is to place the system boundaries under investigation. The extent of the effort that needs to be expended on any system development depends on the boundaries of that system. The system boundaries also determine who should be included in any considerations and who should not. The system stakeholders, scenarios, use cases, goals and system tasks are all determined based on the system boundaries.

Requirement elicitation works to identify the problem to be solved and to identify the business and technical feasibilities. This process also endeavours to identify all the people who could assist in collecting the system requirements.

4.4.2 Requirement Analysis

The requirement analysis task follows the requirement elicitation process. The requirements collected from the stakeholders are analysed in detail in order that the targeted stakeholder needs be satisfied. The requirement analysis begins with classifying the requirements and organising them into related groups. In other words, the analysis activities entail collating the sets of unorganised, unstructured requirements gathered from previous stage in order to investigate the relationships among them. The result of this stage must be a collection of consistence, unambiguous and

complete requirements.

Many stakeholders are involved in building system requirements, and therefore it is likely that some of the requirements will conflict. To resolve any such conflicts, negotiations among the system stakeholders should be conducted.

In brief, requirement analysis takes the stakeholder needs as an input and produces a formal specification product as an output.

4.4.3 Requirement Specification

Requirement specification is the product of the RE process; it describes what must be done, and what must be done is what the targeted system must do. It describes the services, performance and the system features. In other words, requirement specification describes the functional and non-functional requirements of a system.

The functional requirements are what the system must do, and the non- functional requirements reflect the anticipated performance of that system. Besides the functional and non-functional requirements of a system, the requirement specification process describes the data that flow into and out of a system.

A system specification file is an official legal (contractual) document that describes what must be done. It can take various forms, from simple hand-written documents, to prototypes and graphical or mathematical models; it may be a combination of these.

4.4.4 Requirement Validation

Requirement validation measures the validity of the product as a result of the RE process. It deals with demonstrating that the outcome of the requirement collection process satisfies the stakeholders' needs.

A formal technical review is often used as the main requirement validation mech-

anism; this includes the participation of the system stakeholders. Such a formal review assesses the system specifications to search for errors in content, to ensure understanding, to eliminate lack of information, to identify remaining requirement conflicts, and to determine whether or not the requirements are realistic.

Certain techniques are employed to ensure that the stakeholders' requirements have been clearly identified and that their needs are met. Some of these techniques are requirement traceability, verifiability, comprehensibility and adaptability. Thus, the ideal requirement specification file should be complete, correct, feasible, necessary, prioritized, unambiguous and verifiable [145][20].

4.5 Requirement Elicitation Methods

In the subsection 4.4.1, requirement elicitation was defined as the practice of collecting requirements by interacting with the targeted system stakeholders. At the beginning of the requirement elicitation process, the system needs are not yet understood. Therefore, the needs that are to be included as requirements have not yet been decided upon; not even which requirements must be included in the final product.

They are many elicitation techniques and selecting which ones to employ depends on various factors. These factors may be the time or resources available to the requirements engineer as well as the identified system boundaries. Some of the techniques used in requirement elicitation are:

1. Interviews::

   The interview technique is often adopted as a requirement elicitation method, which entails by meeting and interviewing a sample of system stakeholders. The interview questions can be predefined as open or closed (or both) and the interview itself conducted in a structured, semi-structured or unstructured

manner; this latter may be conducted in a largely unprepared manner, in which the interview takes the form of an open discussion.

2. Story Cards:

   A story card is a written description, containing conversations and details pertaining to a 'user story'; such a story identifies the functionality that is considered important to that user (stakeholder).

3. Inspection:

   Inspection is important when there is a large volume of data and formal documentation.

4. Observation:

   In the literature, the terms observation, social analysis and ethnography have largely the same meaning. Observation is a method of collecting requirements by watching (or sometimes participating with) people performing their normal duties.

Each elicitation method has its own advantages and disadvantages; however, this depends on the system domain. The requirement engineer should select the method or combination of methods that are appropriate to the domain. Because scenarios have been widely used as a method for eliciting and modelling system requirements [54], the scenario and story card methods are selected in this research for the purpose of requirement elicitation. More details are given in the section below.

4.5.1 Scenarios

A scenario is the instantiation of a generic task type or a sequence of generic tasks connected by transitions. It describes the characteristics and the social protocols

that should be in place, and it explains what the users should do (or try to do) at the requirements level, but not how they should do it [140].

The term 'scenario' has a number of definitions in software engineering. For example, a scenario is an interaction, and it describes a sequence of actions that relate to real-life events rather than abstract descriptions of the functions [137]. Sutcliffe has another definition for scenario; he describes it as "narratives that describe the usage or operation of a system, either drawn from experience of accidents or imagined future situations for system operation" [138].

Scenarios have been widely used in RE as a technique for requirement elicitation, requirement analysis, removing ambiguities, detecting missing features and conflict among features, and verifying and validating requirements.

4.5.1.1 Scenarios and Requirement Elicitation

Scenarios are considered to be a very useful technique for requirement elicitation, for two reasons; firstly, a scenario affords flexibility, as after each scenario session, the system designer can identify and analyse the requirements. Secondly, the stakeholders' responses after interaction with a scenario serve to improve requirement elicitation. Moreover, for a stakeholder, it is easier to relate to a scenario than to abstract statements on what is required of the system. Scenarios generally follow the specification of the initial requirements.

Figure 4.6 illustrates a scenario-based requirement elicitation, as presented by Potts. It demonstrates that a scenario is an iterative process consisting of three parts: document requirements, discuss requirements, and evolve requirements [116].

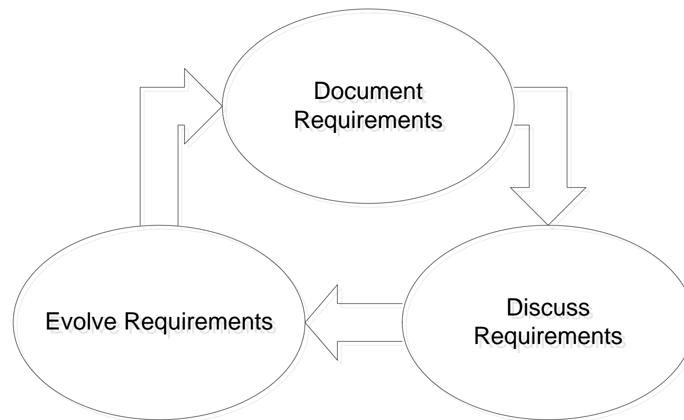

Figure 4.6: Iterative Requirement Elaboration Process [116].

Figure 4.6 shows what may be described as a requirement elaboration process; it begins with documenting the requirements. This entails documenting and collating information gathered from the system's stakeholders, analysing existing documents, and finally, writing a new draft requirement document. A draft requirement document usually contains information pertaining to the domain of the system to be developed, system constraints, environment information and related documentation.

The discussing requirements stage has three steps. Initially, the collected requirements are presented to the stakeholders. Then, feedback is collected in the form of opinions, suggestions and answers to questions, until a general agreement is reached. In the discussing requirements stage, the output of the requirements must be documented for future reference as well as for further refinement, which is the third stage.

The evolving requirements stage is the last one in the requirement elaboration process. In this stage, the input is requirement document derived from the discussing requirement stage, and this input is taken as a key factor in deciding to freeze a requirement, to change it or to add more information to it.

In 1990, Holbrook proposed Scenario Based Requirements Elicitation (SBRE) [65], which involves creating scenarios in a recurring process. Feedback is taken from the stakeholders to achieve refinement. The stakeholders then provide feedback, and finally the functions expected from the system are reviewed. The importance of this scenario review is because hitherto unstated requirements may emerge and this may improve the system requirements as a whole.

In general, scenarios reveal information about functions, operations, features of the system under investigation and the system's surrounding environment. A scenario may describe organisational settings, manual system actions within the organisation, actors and their roles.

4.5.1.2 Scenarios

The scenario-based method consists of three steps, which are:

1. Scenario Description
2. Scenario Analysis
3. Scenario Modelling

Scenario Description:

Many methods have been used for scenario description; they are used as tools to capture scenarios in different formats, such as text narratives, sketches, screen shots and informal media. The result of a scenario can be represented as a formal presentation, an event, or in visual form (as in use cases).

Several techniques and tools have been introduced for capturing and modelling scenarios in the field of RE. For example, ScenIC proposes schema of goals, objectives, tasks, obstacles and actors; actors are usually people but they can be machines

[137]. ScenarioPlus is another tool for scenario-based analysis [69]; it verifies, animates and plays back scenarios. These tools allow stakeholders to express their requirements to software developers. ScenarioPlus contains a set of add-on tools, such as a use case toolkit and a diagrams toolkit, to be used in scenario-based requirement elicitation and analysis.

An example of a plain-text scenario is given Figure 4.7 below through an automated teller machine (ATM) scenario. Consider that an ATM system has been developed to perform basic financial transactions. A simple scenario may then be:

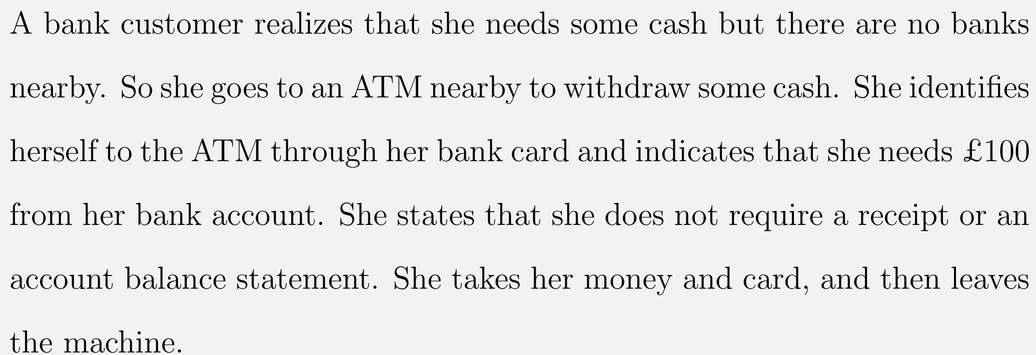

Figure 4.7: ATM Scenario

Scenario Analysis:

According to Carroll a scenario contains the following four characteristic elements [126]:

1. Actors

2. Background information about the actors and their environment

3. Goals

4. Sequence of actions or events

For every scenario, there must be at least one actor who completes a sequence of tasks in order to achieve a goal in the circumstances of the given context or environment. Events are actions done by external systems.
Based on the example above, the scenario characteristic elements are five in number; they are:

- Actor: the person who uses the ATM (bank customer).
- Environment: the location of the ATM.
- Action: the user requests money from the ATM.
- Events: the system retrieves the bank customer's profile and account information.
- Goal: performing a bank transaction.

Scenario Feature Modelling:

Feature modelling helps software engineers to elicit similarities and differences among the features covered by a system. At this level of abstraction, a system feature is defined as stakeholder's view of the system's characteristics and of product functionality. A system feature is extracted based on the business objectives or goals. The business goals and the system features are related though certain scenario behaviours [81]. Figure 4.8 shows the relationships between scenarios, goals and features.

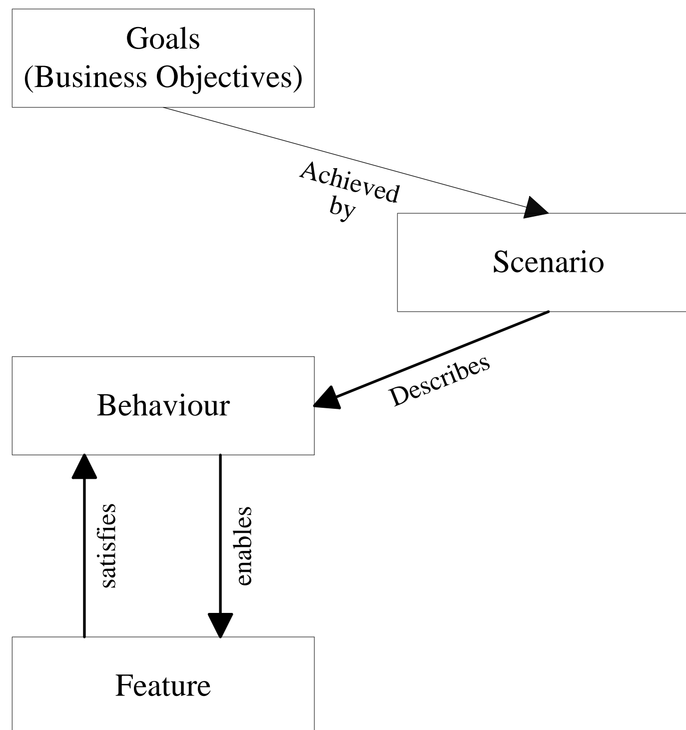

Figure 4.8: Goals, Scenario and Features [81]

4.5.2 Story Cards

Story cards are generally identified as hand-written paper notes designed for requirement elicitation [129]. Besides providing a visible expression of a user story, story cards "represent customer requirements rather than document them" [43]. Cards not only contain the text of a story, they also contain details derived from a recording of the ensuing conversation [113].

For example, consider that a local bank needs an ATM system for basic banking transactions. Analysts need to elicit the requirements of the required system in order to develop that system. To achieve this, the stakeholders state their requirements in story card format. In this scenario, the bank customer is welcomed into the system, logs in securely and accesses the main menu. These can be represented as

in Tables 4.1, 4.2 and 4.3.

Story Card No. 1	SC1
Description	The system should a present Welcome screen to the ATM user.
Note	Provide a Welcome message to the bank customer showing where to log in.

Table 4.1: Story Card for Bank Customer - Welcome Screen

Story Card No. 2	SC2
Description	The bank customers must log in before performing any transaction.
Note	Provide authentication procedure, such as a message for the user to enter an account number and password.

Table 4.2: Story Card for Login

Story Card No. 3	SC3
Description	The bank customer chooses a type of transaction to perform.
Note	Provide a menu wherein the ATM customer is able to select a banking transaction.

Table 4.3: Story Card for Main Menu

4.5.3 Analysis to Derive Features

The system requirement analyst takes the information from the story cards, such as user ID, username, story card number, story description, date and time, and documents all this information in a database. Once the information has been collected, the analyst combines and analyses all the information. The information is then filtered and prioritized.

Thus, analysts derive system features based on information filtering and prioritizing. Table 4.4 shows an example of the Welcome screen feature, and Table 4.5 shows a user authenticated feature.

Feature No. 1	F1 Welcome Screen
Description	The bank customer sees the Welcome screen once they access the system.

Table 4.4: Feature 1 - Welcome Screen

Feature No. 2	F2 Login
Description	Bank customer must be authenticated before any bank transaction is performed.

Table 4.5: Feature 2 - User Authentication

4.6 UML-based Feature Modelling

Each feature represented in the feature cards in the above examples can be represented as a UML diagram. The feature cards in Tables 4.4 and 4.5 can be modelled in UML modelling diagrams. The power of the UML representation of a system's structure and behaviour is that it enables software engineers to represent these feature cards in UML modelling diagrams. This transformation process involves software engineers in modelling these feature representations (feature-oriented modelling was described in Chapter 2).

Two examples can be given here to illustrate the modelling of structural and behavioural feature representations. The feature presented in Table 4.4 can be modelled in a UML structure diagram. For example, the class diagram ***Message***, which has the attribute *MSG : String = Welcome* and the operation *PrintMessage*,

is shown in Figure 4.9.

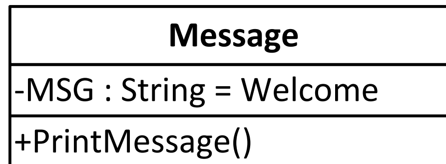

Figure 4.9: Class Diagram - Display a Welcome Message.

View Account Balance is Another example. This feature can be represented as UML class diagram as well as shown in Figure 4.10

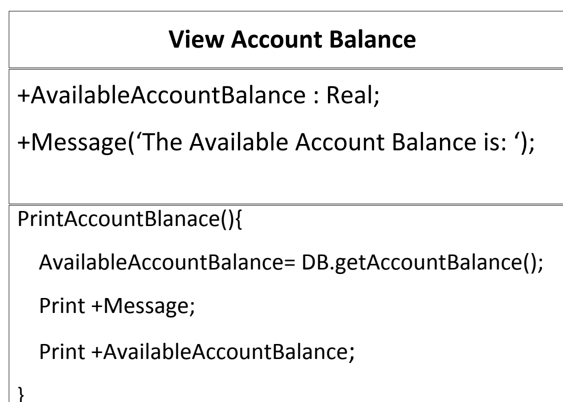

Figure 4.10: Class Diagram - View Account Balance.

Another example of presenting a feature was described in Table 4.5, and this can be modelled by using a UML behaviour diagram.

The feature *Login Required* reflects the fact that an ATM user must be authenticated before he or she is allowed to perform a banking transaction. This feature can be presented as a UML activity diagram, as shown in Figure 4.11. However, this feature can be decomposed into a set of two features. One feature displays a message to the user in order to enter information, and the other feature authenticates these data.

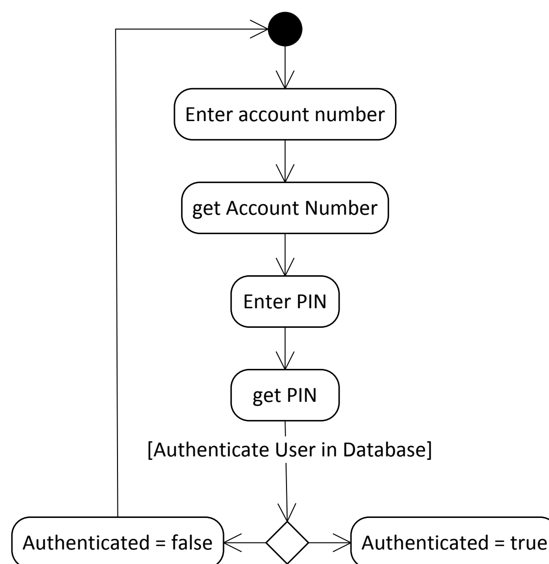

Figure 4.11: ATM User Authentication Activity Diagram

Feature modelling can be represented at any level of software abstraction representation. In this feature modelling, the feature is represented at the design level, since the matching point of this research is chosen to be at this level.

4.7 Conclusion

This chapter began with an overview (and highlighted the importance of) the business feature elicitation stage. In the feature IT business framework, this stage has two main objectives; the first is business analysis and the second is software requirement engineering. The software requirement engineering stage is the main focus of this research. The business analyst elicits the business change and business needs, and seeks to provide solutions, while the software requirement engineering process transforms these solutions into a formal specification report.

System analysis and requirement engineering are important aspects of the problem space. The output of the requirement engineering process is eventually transferred into the solution space.

Chapter 5

IT Feature Extraction

Objectives:

- To discuss the role of the IT Feature Extraction phase within the proposed framework
- To define the relevant terms used in this phase
- To highlight the role of program slicing
- To illustrate a feature extraction rule

5.1 Importance of the Phase

The feature extraction phase is a reverse engineering process. Software reverse engineering starts with understanding the system that is to be reverse engineered. Software understanding is defined as any activity that involves extracting knowledge from a program in order to better understand the software [147]. Figure 5.1 shows level of abstraction of entity:

1. Data level: the data and program instructions.
2. Information level: the system presentation and design.
3. Knowledge level : the system architecture.

The higher the level abstraction, the less detail presented. The lower the level, the more detail presented.

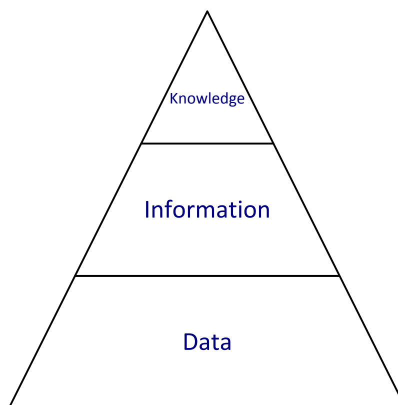

Figure 5.1: Level of Abstraction

Software that has not been updated (or inaccurate software documentation) is a common problem in software development, mainly created by a program or software package being allowed to lag behind its optimal evolution [114]. When software documentation is no longer reliable, software developers return to the software source code, as this becomes the only (or the most reliable) form of documentation.

Studying software code in order to extract knowledge is a complex and difficult task, particularly when the system has thousands of lines of code. Therefore, theories, techniques and tools have been developed to support software understanding. Some of these techniques are lexical analysis, syntactic analysis, control flow analysis and data flow analysis, and some of the tools automate the process of program understanding; however, developer support is always required.

It is important to transform source code into an upper level of abstraction so that software developers are more able to undertake the program understanding process. Program visualisation (PV) is the process of transforming program source code or program algorithms into graphical form in order to better illustrate the software program in question [14]. PV tools are very useful in supporting developers in system analysis, modelling, testing and maintenance. However, there is a need to visually represent the whole software system in a blueprint manner.

Having improved the level of software understanding, the feature extraction phase aims to target the software system features embedded in the source code and to represents these features in a level of higher abstraction. The first step is to slice the software system static source code into a set of related statements, called slices. The second step is to present them in Program Dependency Graph (PDG), highlighting the slices. Finally, the PDGs are used as a source to extract software

features, which are then represented as UML diagrams. Figure 5.2 shows the steps of this IT feature extraction phase.

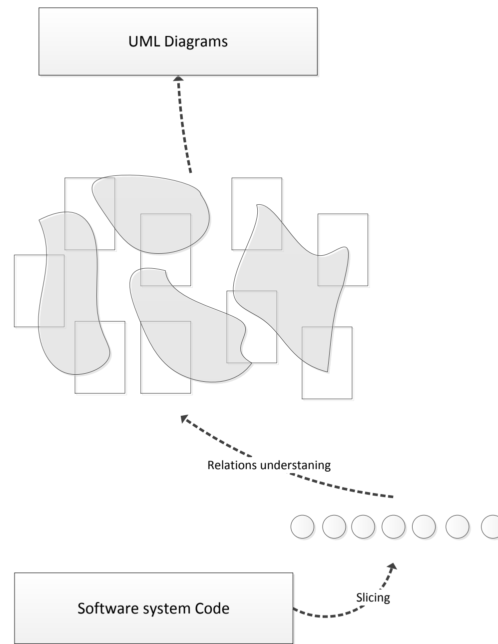

Figure 5.2: IT Feature Extraction Process

5.2 Program Understanding

The main goal of this phase is building a knowledge base the program undertaken. Program understanding aims to extract knowledge of a computer program. The extracted knowledge, about computer program, enable software engineer to better understand and model computer program in higher abstract level. Moreover, it is important for activities such as bug correction, enhancement, reuse, and documentation.

Software understanding process works in a reverse engineering process; it can

starts at any software development level. However, reverse engineering practise usually starts at the system source code level. It transforms software system code into higher level of abstraction of a software life cycle level of abstraction shown in Figure 5.1. Reverse engineering techniques analyse and identify the system's components and the relationships among them. From these relationships, the reverse engineering process creates higher representations of the system based on these relationships.

5.3 Program Slicing Step

Program slicing is a reverse engineering technique, first introduced in Mark Weiser's PhD thesis in 1981 [149]. It is a technique used to decompose a given program into independent slices, based on given criteria [148]. Weiser defined program slicing as a transformation technique used to extract an executable set of related statements from a program; he called them slices [149]. With respect to the slicing criterion, the resultant slices must together have the same behaviour as the original program [31]. A 'slice program' is therefore a subset program of the original program but the slice program itself is an executable program whose behaviour must be identical to some part of the original program's behaviour [131].

Figure 5.3 shows static program slicing applied to a simple sequential program. This example shows that program slicing is the process of removing those parts of the program that are either not related to or have no affect on a certain statement or variable.

(1) int x;	(1) int J;
(2) int total = 0;	(2) int total = 0;
(3) int gama = 1;	
(4) for(J = 0; J < N; J++) {	(4) for(J = 0; J < N; J++) {
(5) total = total + J;	(5) total = total + J;
(6) gama = gama *J;	
}	}
(7) write(total);	(7) write(total);
(8) write(gama);	
The new subprogram was produced with respect to the criterion (write(total),{total}):	

Figure 5.3: Backward Slicing Example

In this step, program slicing is used as a program understanding technique. It has been studied and used for the purposes of program analysis and program transformation as well as a reverse engineering technique. Also, program slicing has been used to identify the relationships between individual parts of program statements, based on a given initial program point, which is called a slice criterion.

A slice criterion is usually a single statement within a program or program variable. For example, given a slice criterion C, all program statements that have a direct or indirect influence on C are added to the slice S. In this case, S is called a backward slice, based on the slice criterion point C. On the other hand, a forward slice S contains all the program statements that are directly or indirectly influenced by any change in the slice criterion point C. Thus, at the end of the slicing process, a set of related statements is generated, based on the given criterion C.

Figure 5.4 shows part of a slice based on forward slicing technique and on the statement criterion *'userAuthenticated = false'* . The figure also shows all the statements that have been excluded from (as well as included in) the slice.

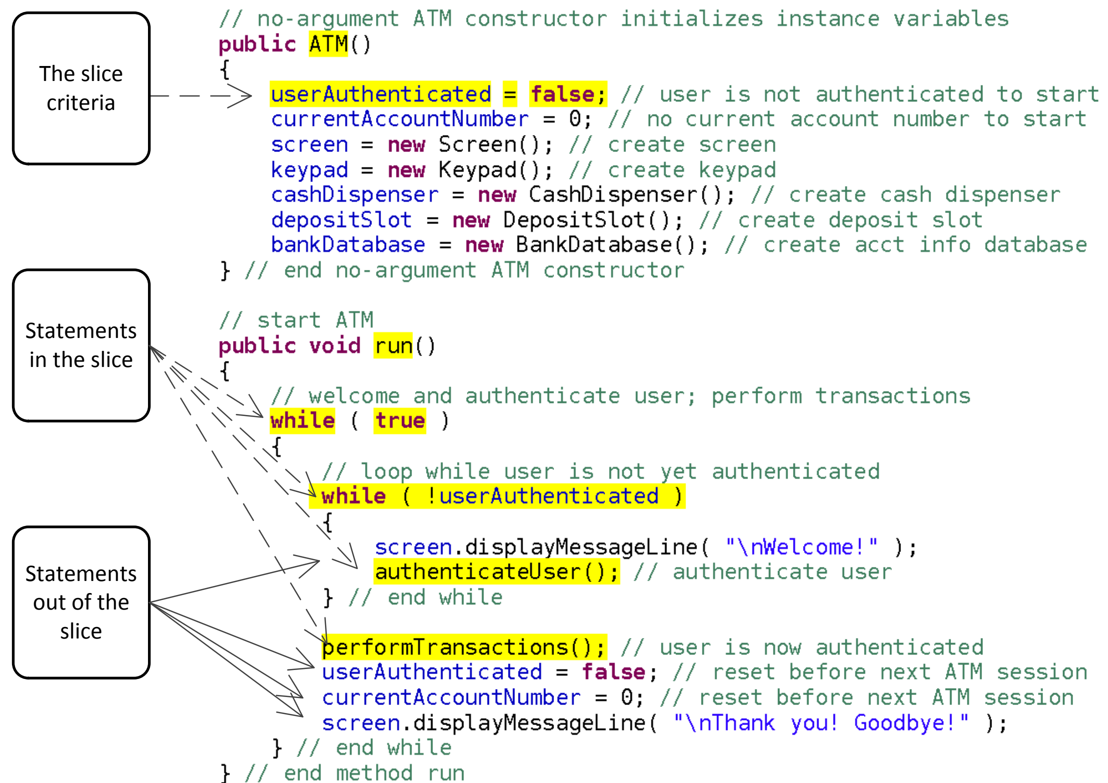

Figure 5.4: Forward Slicing Example

Usually, Program Slicing is viewed as a program transformation technique. It is viewed as a technique, which delete part of none related statements to yield a slice based on a certain statement (the slice criteria). It's also used as a technique for program restructuring, program differencing, program testing, program reuse and program security.

In this phase, the program static slicing is used as a program analysis technique. On other words, it is used to extract related statements which are usually related to a certain feature. However, these statements which represent a certain

feature can belong to a different classes in a software system as Figure 5.2 illustrates.

5.4 Slices-to-UML Step

The second step of the IT feature extraction phase is building a representation of the software system following the program slicing process. Accordingly, a program is presented as a Program Dependence Graph (PDG). A PDG is a directed graph that contains a set of nodes and edges. A node represents a program statement and an edge represents dependency direction. Each node in a PDG must be in between two other nodes, and the PDG must start with a '*start node*' and end with an '*end node*' or a '*stop node*'. Figure 5.5 shows an example of a PDG.

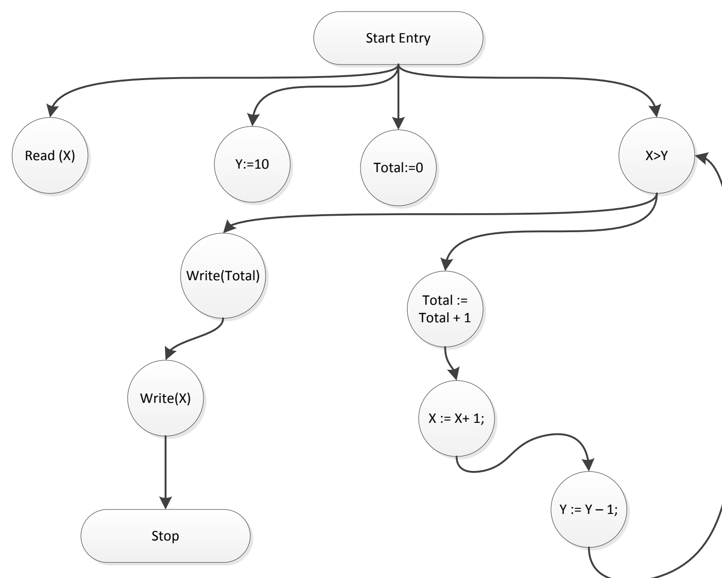

Figure 5.5: Program Dependence Graph

Besides the PDG, the software representation step includes a program Control Flow Graph (CFG). A CFG presents the program flow from one statement to another. In a CFG, a node represents a statement in a program, and an edge represents a possible flow from one node to another. Figure 5.6 shows an example of a CFG, taken from the program presented in Figure 5.4.

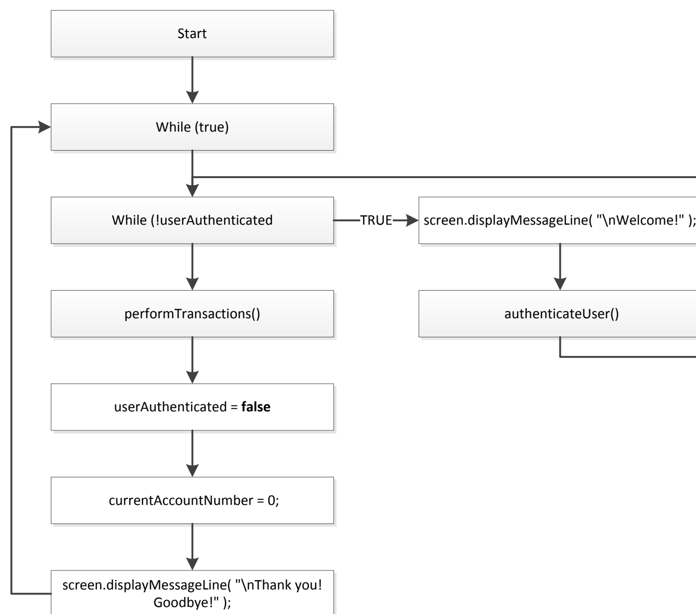

Figure 5.6: Control Flow Graph

In this step, the slices taken from the program slicing stage are presented alongside the statements outside the slice, as a program flow graph; this is shown in Figure 5.7, where the grey circles represent statements outside the slice, and the white ones within the slice. This process can be performed using software tools; however, the participation of software developers is needed in this part of this process, as discussed in the literature review chapter.

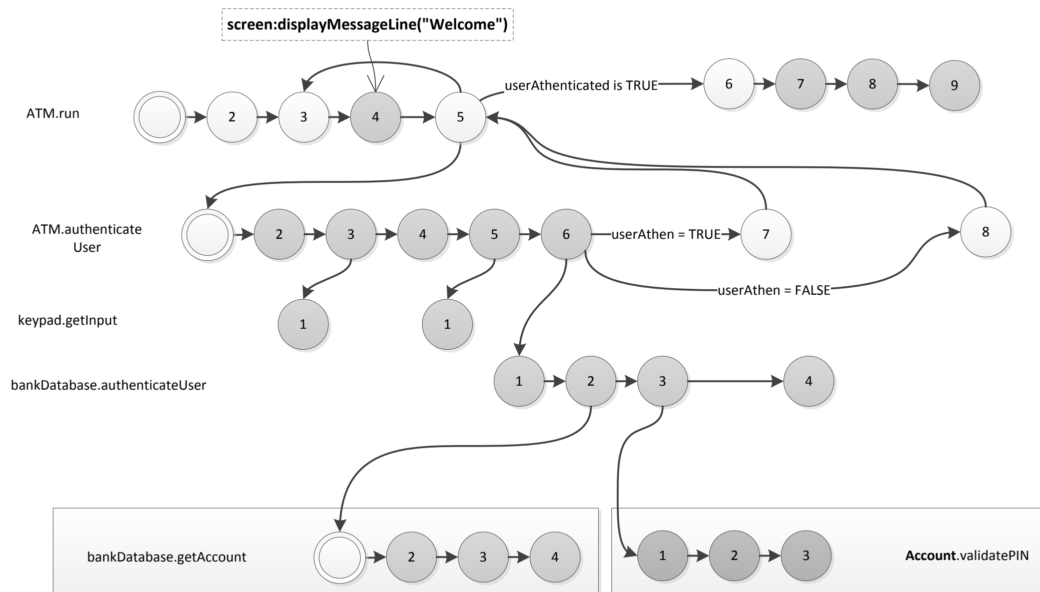

Figure 5.7: CFG and Program Slicing

Because a system achieves its functionality by passing messages among classes. Figure 5.7 shows a slice contain statements belongs to different classes. The statement *"screen:displayMessageLine('Welcome')"* is part of the control flow graph (CFG); however, it's not part of the path of the Program Dependence Graph (PDG) of the two statements before it and after it.

Figure 5.7 Shows that PDG is connected to class methods. Examples of these methods are:

1. *ATM.authenticateUser* method is divided into sub PDGs to achieve its functionality. These divisions contains accepting data to be validate, accessing database, validate data, and finely showing result.

2. *BankDatabase.authenticateUser* method takes user login and password then authenticate the user by access the database. Even it is part of the CFG; it has its own path in the PDG.

3. *BankDatabase.getAccount* method forks from BankDatabase.authenticateUser PDG. It access the database to retrieve bank account number.

4. *Account.validatePIN* method forks from the BankDatabase.authenticateUser PDG. It access database to retrieve bank customer Personal Identification Number (PIN).

5.4.1 IT Feature Representation

The Unified Modelling Language (UML) is modelling language that has been widely used to represent system behaviours and structures. It consists of six diagrams for modelling the software system structure and seven models for modelling the software system behaviour. The structure diagrams that sometimes called static diagrams are used to model the system static structure. These diagrams are:

1. Class diagram

2. Composite diagram

3. Component diagram

4. Deployment diagram

5. Object diagram

6. Package diagram

The other seven UML diagrams are behaviour or dynamic diagrams. They are used to model system behaviour. These diagrams can be divided into two group. The first group model the system behaviour. These diagrams are:

1. State diagram

2. Use Case diagram

3. Activity diagram

The second group are these diagrams that model how the system interaction to achieve its goals. These interaction diagrams are:

1. Communication diagram
2. Interaction diagram
3. sequence diagram
4. Timing diagram

As 'software system feature' is defined as "an increment of program functionality", these functionalities can be presented as a structure or behaviour model in UML.

Based on Figure 5.7, a sliced based feature extraction method can be introduced for extracting a software system feature:

Sliced-based feature extraction method states that: a program statement (or set of statements) can represent a software system feature if: (1) it is not part of a program slice, (2) it is located in between two other statements, which are both parts of a program slice.

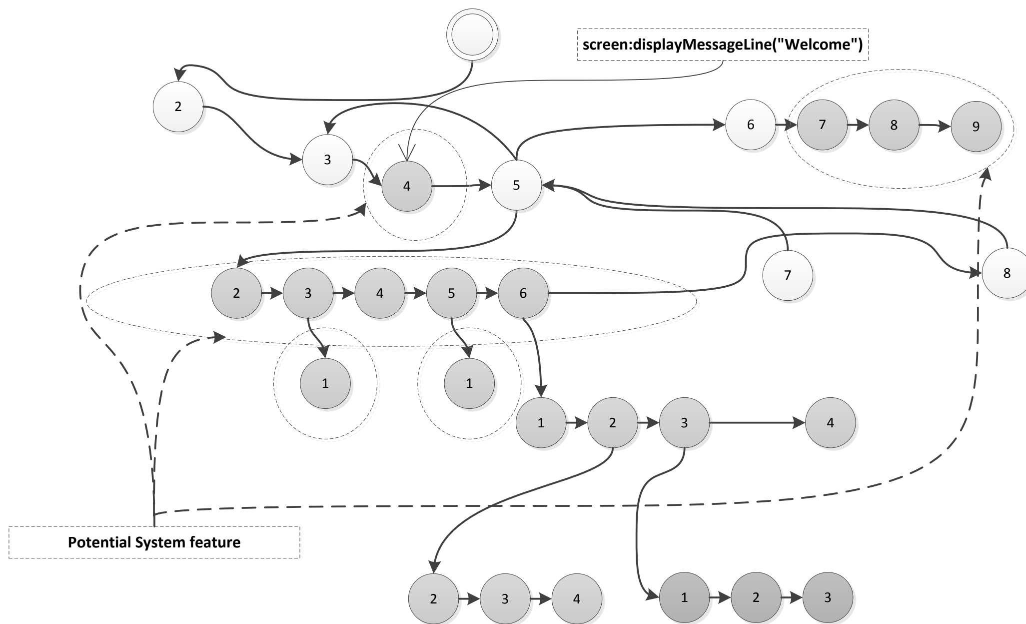

Figure 5.8: Potential System Features in a CFG

In Figure 5.8, the statement "*screen:displayMessageLine("Welcome")*" is located in the middle of two statements that belong to a slice. This statement can be seen as a simple system feature.

The feature *"screen:displayMessageLine("Welcome")"* can be modelled as a simple structure UML class diagram. Figure 5.9 shows a representation of this feature as a class diagram.

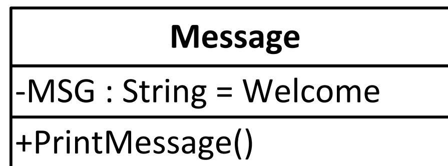

Figure 5.9: UML Class Diagram

As mentioned earlier software developers participation in this process can help. Any of the UML diagrams can be used to model system features extracted from the CFD.

In the case study in chapter 7 parts of ATM and library software systems features were presented as a class diagram and activity diagram.

5.5 Conclusion

This chapter began with an overview (and highlighted the importance of) the IT feature extraction phase. The whole process of this phase is reverse engineering; it takes the system source code from the bottom of the software development lifecycle and reverse engineers it into a modelling language. This phase comprises two main objectives; firstly, the system program source coded is sliced using a static program slicing technique, and secondly, the slicing is used to extract system features, which are then represented in UML form.

Program reverse engineering is a difficult and complicated task. However, the program source code usually represents a reliable source for understanding the system's requirements. The output of this phase is a set of features modelled in UML modelling diagrams, which can be exploited in the next phase within the proposed framework.

Chapter 6

Feature Model Matching

Objectives:

- To give an overview of model comparison
- To discuss algorithms for diagram comparison
- To discuss model comparison approaches
- To introduce the class model matching algorithm

6.1 Introduction

In the feature-based IT business framework described in Chapter 3, feature model matching is the final phase. As mentioned in the proposed approach chapter, the proposed framework builds a linkage between the business needs and the supporting IT infrastructure. The feature model matching phase links the outputs of the other two (as illustrated in the previous two chapters, they are the top-to-middle phase, which was covered in the business feature elicitation chapter, and the bottom-to-middle phase, which was covered in the IT feature extraction chapter). Figure 6.1 illustrate the goals of the previous two phases and the position of this current one.

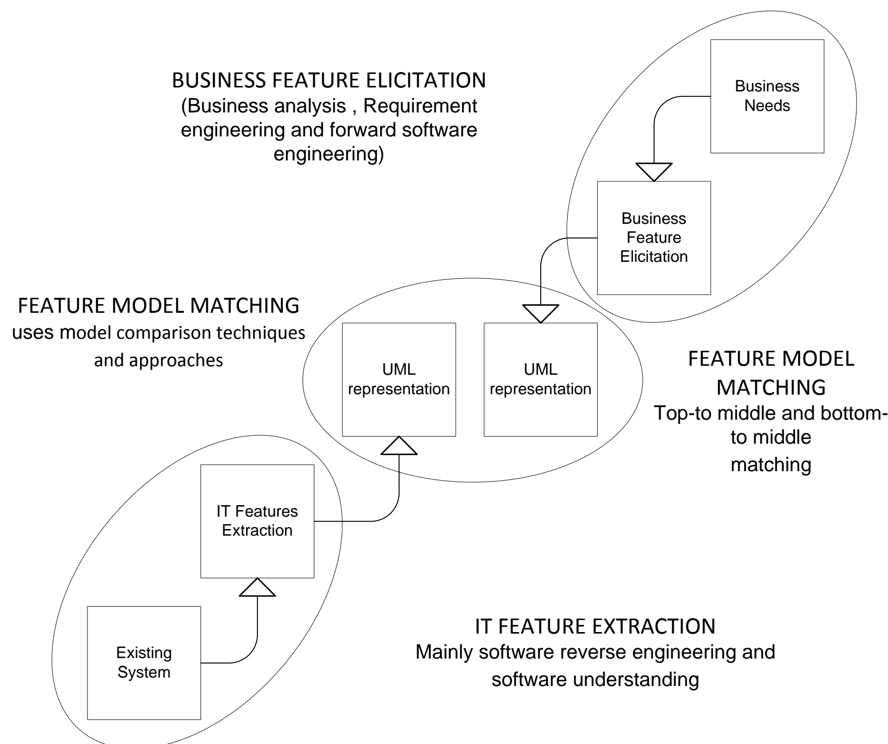

Figure 6.1: IT-Business Framework and Matching Stage

Thus, the aim of this phase is to match the features (represented as UML diagrams) generated in the previous two phases (business feature elicitation and IT feature extraction). The matching phase reveals implemented features or missing features from the both sides.

6.2 The Matching Problem

Matching problem is an important and a well-known area of research in computer science. It has been researched in many computer science subjects such as semantic web, ontology integration, database, data warehouse, model driven engineering and others domains.

The term semantic matching is used in computer science to refer to elements semantically related [50].

An example of a semantic matching is the S-Match algorithm. Two graphic-like structure are matched to identify elements those semantically similar to one another. The similarity can be for example torch and flashlight or associate professor in the US academic system corresponds to senior lecturer in the UK academic system as shown in Figure 6.2.

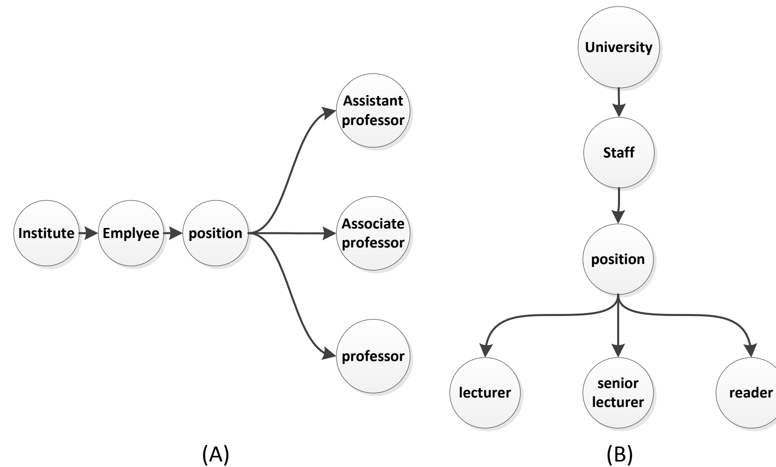

Figure 6.2: A Scholarly Teacher in USA and UK Academic System

In general, matching takes schemas or ontologies as an input and generate an output as relationship. An input for matching is two sets of entities such as tables or UML diagrams. The output is relationship between these two entities (equivalence, not equivalence).

6.3 Model Comparison and Approaches

As illustrated in Section 2.7 of the chapter Feature-based Approach: State of the Art, model comparison is the practice of identifying similarities and differences between two model's elements. It is an MDE practice which has been used in model versioning and model clone detection, beside many other MDE areas such as model composition, model transformation testing, and others [136]. Model comparison methods are categorized based on the models they designed to work on. A number of comparison methods are here reviewed in order to investigate the need for their use in this phase.

1. Multiple model methods:

   These methods are able to compare two different types of models: structural and behavioural.

2. Behaviour and data-flow model methods:

   These methods are designed to deal with behaviour and data-flow models; comparison approaches use concepts derived from graph theory as well as graph theory applications [46].

3. Structural model methods:

   These methods are for comparing the models of two system structures. In this respect some work has been done in UML class diagram differences [60][121].

4. Product line architecture methods:

   These methods are applied to models that are taken to be different versions of the same artefact [33].

5. Process model methods:

   These methods are for comparing models that represent software development processes. Process models are compared based on node similarity, structural similarity and behavioural similarity [49].

Delta algorithms (or difference algorithms) are applied to find any differences between two different versions of the same artefact. Delta algorithms are extensively used in text-based artefacts to reduce storage space through text merging [101]. However, these algorithms can be used to calculate and represent differences between two models [5]. There are two delta algorithms: symmetric delta and directed delta.

Symmetric delta can be defined as:

$\Delta(v_1, v_2) = (v_1/v_2) \cup (v_2/v_1);$

where v_1 and v_2 represent two versions of an artefact.

An example of symmetric delta consider that:

$$v_1 = \{1,2,3\}$$
$$v_2 = \{3,4\}$$

By applying a Δ operation

$$\Delta(v_1, v_2) = \{1, 3, 4\}$$

Figure 6.3 shows an intersection shape that represents the shadow as $\Delta(v_1, v_2)$

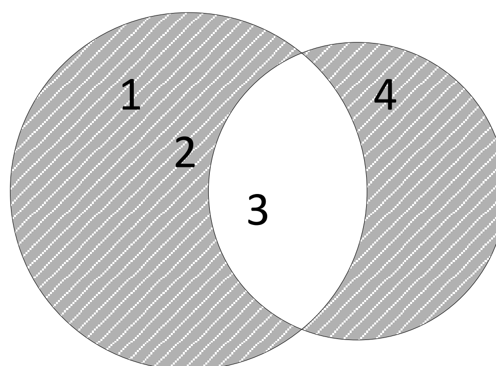

Figure 6.3: Intersection Shape for $\Delta(v_1, v_2)$

Directed delta (or sometimes called change delta) is a set of sequential operations (or changes) such that when this set of operations acts on one version's model (v_1), it transforms that model into another version (v_2). These sequential operations are adding or deleting elements from one set in order to change it to another.

Directed delta is usually connected to data compression. It takes a set of data and transforms this set into a new set. The new set is smaller than the original set and can be transform back to the old set. The directed delta achieve this by removing data redundancy.

6.4 Proposed Model Matching Approach

The word matching in computer science is defined as a process of taking two graphic-like models or structures (e.g. UML models, graphs, etc.) and producing a map between those two input based on similarities to each other.

In this research, a rule is introduced which states that:

If only an "ADD" operator or only "REMOVE" operator is applied to a set v_1 and a set v_2 can be generated; then v_1 is matched to v_2 and v_2 is matched to v_1.

For example:

$$v_1 = \{1\}$$
$$v_2 = \{1, 2, 3, 5\}$$

By adding 2,3,5 then v2 can be generated. Therefore, v_1 in matched to v_2. On the other hand, if an operator remove is applied to v_2 by removing 2,3 and 5, v_1 can be

generated.

Figure 6.4 shows a class diagram is transformed into another class diagram for matching.

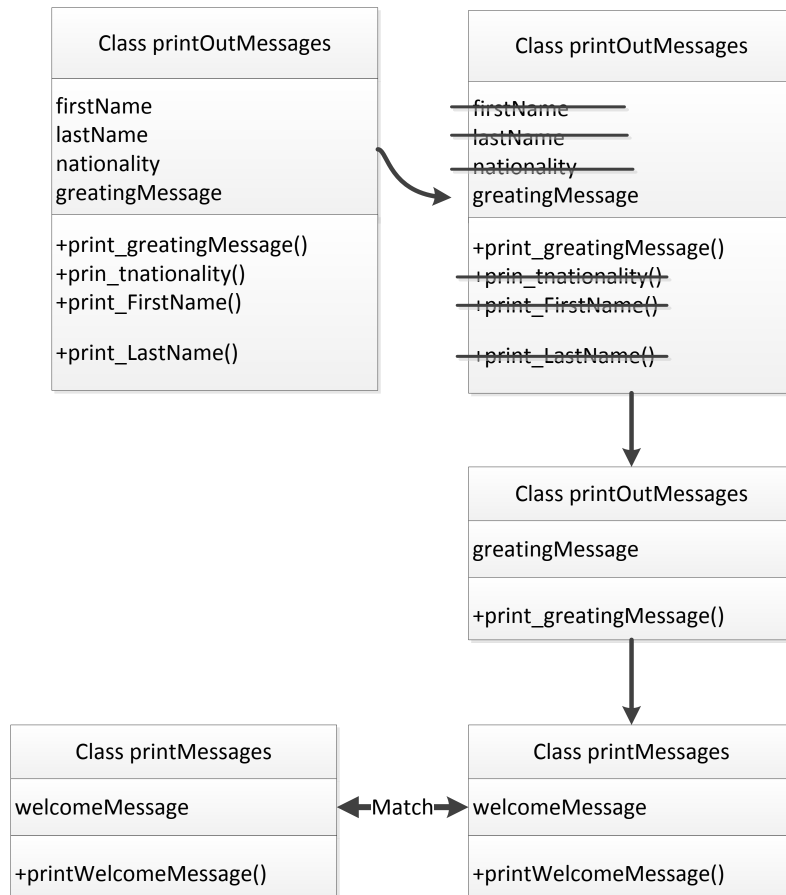

Figure 6.4: Class Converting for a Matching Algorithm

The proposed model comparison algorithm is based on a directed delta algorithm. A directed delta algorithm uses a set of operations to turn one model into another, i.e. it transforms one set (v_1) into another set (v_2) by searching for ele-

ments within set v_2 and then adding or removing elements from v_1 until sets v_1 and v_2 contain the same elements.

As some models might be subsets of other models, it is important to understand that one or more features might be represented as a single model. Therefore, one or more models, representing system features, might be matched into a single model, which represents a set of features; as shown in Figure 6.5. Model matching in this research refers to elements that represent the same idea or artefact.

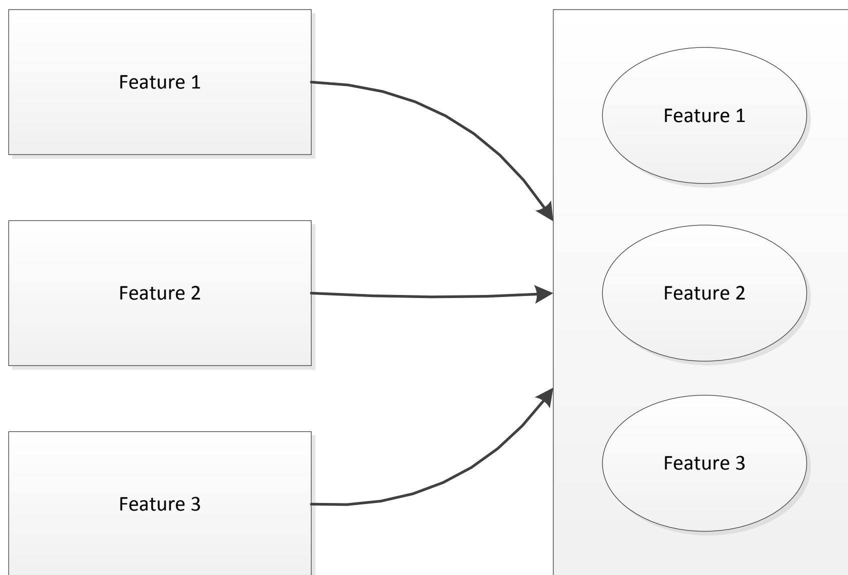

Figure 6.5: Feature Model Mapping

Based on this discussion and the structural model methods, a feature matching algorithm is proposed:

Algorithm : A feature of class-based Model v_1 is matched to a feature class-based model v_2 if all element of v_1 exist in model v_2.

The algorithm achieves its matching approach of two class models through a set of steps:

1. Given 2 class models, set all elements of class model 1 to v_1 and all elements of class model 2 to v_2.

2. Set *Flag-match* to false.

3. If all elements within v_1 exist in v_2, set Flag-match to true.

4. If all elements within v_2 exist in v_1, set Flag-match to true.

5. If Flat-match is true, the models v_1 and v_2 are a match.

In some models, such as class UML models, attributes and operations are represented in natural language (e.g. English). However, this natural language representation makes the model comparison or matching a complicated task. One approach is to use AI language-related techniques; however, this is beyond the scope of this research. Another approach is that software developers be part of the process in order to overcome the natural language and design complexity.

6.4.1 Model Matching Example

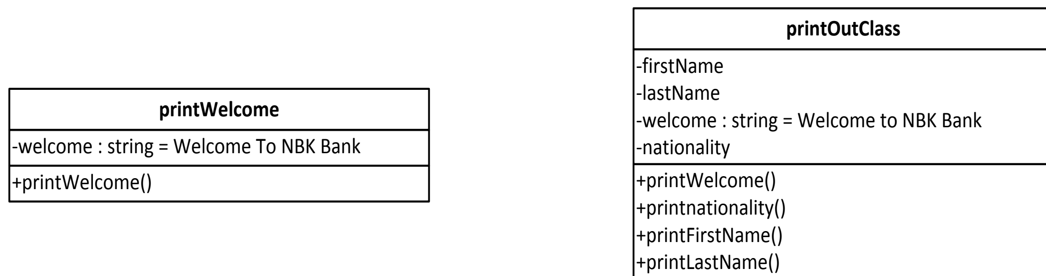

Figure 6.6: Class Model Matching Example

Figure 6.6 shows two examples of class diagrams; one has only one feature and the other has more than one feature. Each diagram consists of a set of elements, i.e. attributes and operations:

printWelcome contains two elements: an attribute, i.e. *welcome*, and an operation, i.e. *printWelcome()*.

printOutClass contains eight elements: 4 attributes and 4 operations.

Attributes: *firstName*, *lastName*, *welcome* and *nationality*.
Operations: *printWelcome()*, *printnationality()*, *printFirstName()* and *printLastName()*.

It is clear that not all elements of *printOutClass* are in the *printWelcome* class; however, all elements of *printWelcome* are in *printOutclass*. An activity diagram for the proposed algorithm and pseudo code is shown below.

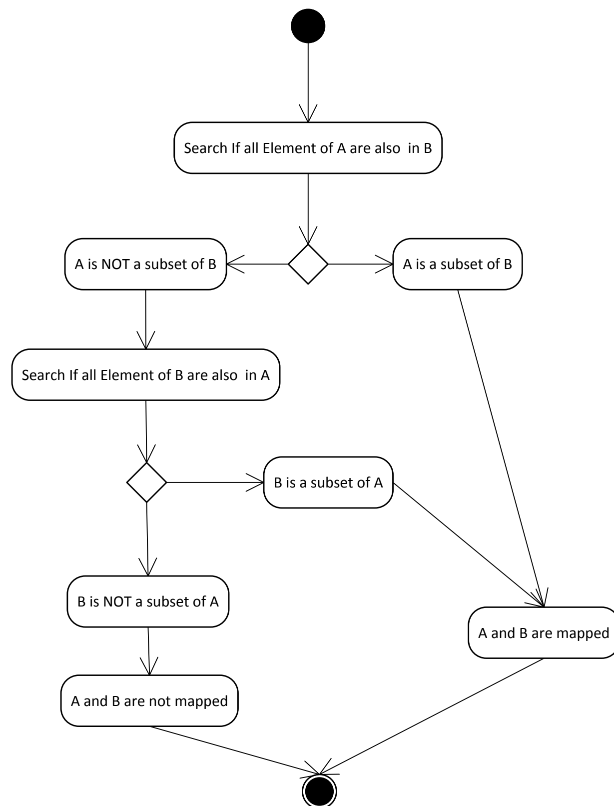

Figure 6.7: Proposed Algorithm Activity Diagram

```
// load first model elements
WHILE (more_element_in_model_A){
```

```
    ADD element to A_element_array
}

// load second model elements
WHILE (more_element_in_model_B){
    ADD element to B_element_array
}

// reset Flags
Flag-match = false;
break_flag = false;

// check if all elements in A are in B as well
FOR (String A_element : A_element_array) {
    FOR (String B_element : B_element_array) {
      IF (B_element <> A_element) THEN // Found an element in A but not
          in B
      BEGIN_IF
          break_flag = true;
         Break;
      END_IF
    } // B_element_array for loop
} // A_element_array for loop


// check if all elements in B are in A as well
IF break_flag = true THEN    //If A is not a subset of B
```

```
BEGIN_IF
    FOR (String B_element : B_element_array) {
        FOR (String A_element : A_element_array) {
        IF  (A_element <> B_element) THEN
                    // Found an element in B but not in A
        BEGIN_IF
                break_flag = true;
              Break;
        END_IF
      } // A_element_array for loop
    } // B_element_array for loop
END_IF

IF (break_flag <> true)
    BEGIN_IF
        Flag-match = TRUE
   END_IF
```

6.5 Conclusion

The feature modelling matching chapter is the final phase of the proposed feature-oriented IT business framework. It works as a linkage that maps the feature modelling from the business feature elicitation phase with that of the IT feature extraction phase.

At the beginning of this chapter, an overview was given and all the relevant terms defined. As this phase is a mapping process, it illustrated the model comparison concepts and approaches.

The main goal of this phase is to link the models of the business elicitation phase to those of the IT extraction phase, and consequently, this phase searches for similarities between two models in order to make that link. For this purpose, an algorithm (for a UML class diagram) was developed to match the features within the two given models; the algorithm was illustrated through an example. At the end of the chapter, an algorithm activity diagram and a pseudo code were provided.

Chapter 7

Case Study

Objectives:

- To review the tools used in this case study
- To show how the framework phases of the proposed approach can be used with an existing system
- To show how to use the selected tools in the software reverse engineering process

7.1 Overview

This case study chapter aims to evaluate the proposed framework presented in Chapter 3. Its objective is to assess the efficacy of the implementation of the proposed approach. All three phases of proposed framework are covered in this case study (i.e. business feature elicitation, IT feature extraction and feature model comparison). To demonstrate this, an ATM software system application is here used for the case study.

Section 7.3 provides background information on the ATM system. In Section 7.2, a brief description of the tools used is given. The first phase of the proposed framework (business feature elicitation) is demonstrated in Section 7.3.2. The second phase (IT feature extraction) is demonstrated in Section 7.3.3. Finally, the third phase (feature model comparison) is demonstrated in Section 7.3.4.

7.2 Tool Support

7.2.1 Eclipse

Eclipse was a project launched in November 2001 by IBM. It is a community for individuals and organisations who wish to cooperate in developing 'open source' software [56]. The Eclipse community works on a variety of open source projects, varying from open development platforms, to tools for building and deploying or managing software.

The Eclipse project first started as a Java Integrated Development Environments (IDE) project but has since evolved to cover many other languages. Moreover, this project has tools designed for modelling as well as for business reporting and mobile applications. One important feature of the Eclipse project is the use of plug-ins

to develop applications in others languages such as COBOL, C, C++, PHP, Perl, Python amongst others. Developers use Eclipse Java IDE as a tool for managing all their software artefacts, such as software coding and debugging. It has a built-in Java compiler and a model of the Java source file.

Eclipse is used in this project as a tool to install the Java code slicer plug-ins. Eclipse 3.4.1 is used even though it is not the latest version available; this is because the Indus Java slicer was developed to work with this application version. Also, the Indus Java slicer was chosen as it works only under Eclipse applications, and the hardware requirements to install Eclipse run acceptably well on an average desktop computer.

7.2.2 Indus Java Program Slicer

Indus is a Java program slicer designed to slice object-oriented Java applications. Besides being the first Java program slicer, Indus is the only publicly available Java slicer [119]. Alongside Indus, Kaveri is a front-end Eclipse plug-in for the Indus Java slicer; t adds highlight annotations to the sliced Java source code [74].

Figure 7.1 shows a screen snapshot of the Eclipse application. This snapshot was taken after running a forward slicing process on the ATM software using the Indus slicer. Kaveri highlights all the statements included in the slice. However, these statements are not all the statements included in the slice, as some other statements, which are part of the slice, are in other ATM application classes. The left side of the screen shows other classes, which comprise other statements included in the slice.

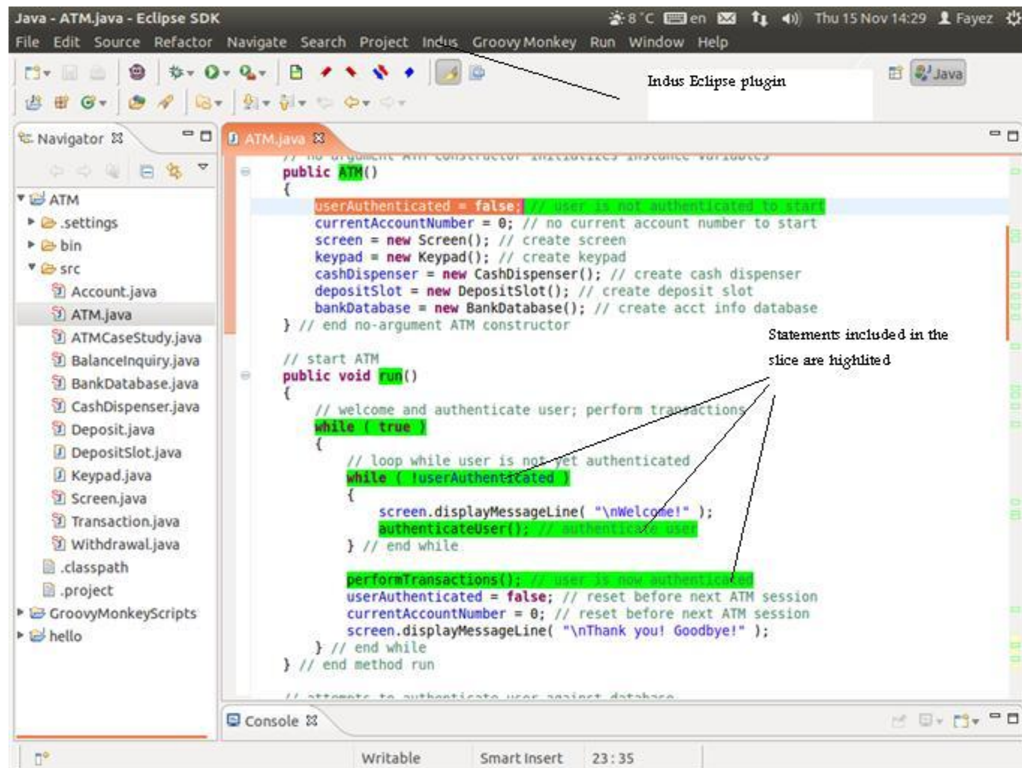

Figure 7.1: Indus Slicing Result

7.3 An ATM System

ATM stands for Automated Teller Machine or Automatic Teller Machine. An ATM is an electronic banking outlet that gives bank customers access to perform basic financial transactions without the aid of a branch teller. Nowadays, ATM systems are important tools, where customers can withdraw cash at anytime of the day.

The ATM software system used as a case study in this research is written in the Java programming language, and it handles basic ATM features [47]. Usually, the main ATM features are:

1. Check account balance

2. Withdraw cash

3. Deposit cash

4. Transfer money

The ATM software system was selected to demonstrate the proposed feature-oriented IT business mapping framework because it provides basic, easy-to-understand features; the fact that they represent banking transactions is of no particular relevance. Figure 7.2 shows a user interface for an ATM software system.

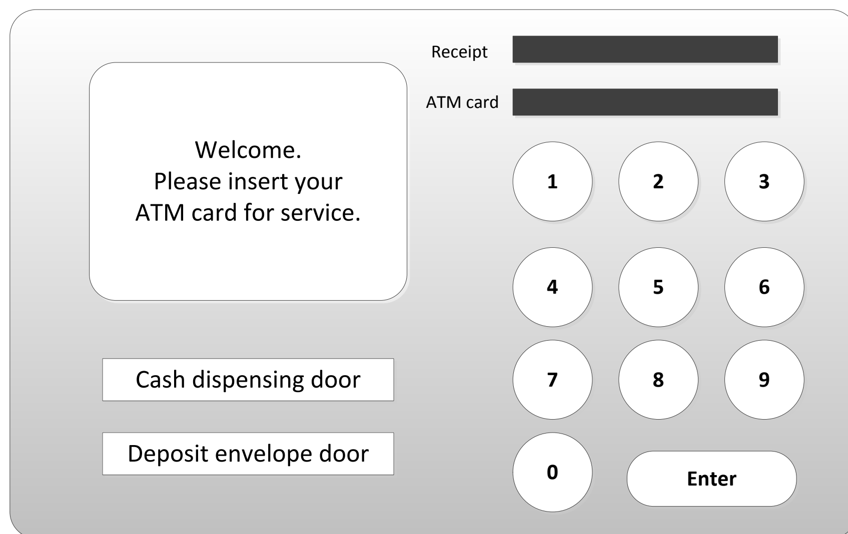

Figure 7.2: A User Interface For An ATM Software System

7.3.1 The ATM Software System Application

The ATM software system application used in this case study runs on Ubuntu, a desktop Linux operating system. The program is simple and the system has a basic

user interface; its overall simplicity is essential for fully comprehending the research approach.

Welcome screen. The program starts by displaying a welcome screen, requesting the end-user to enter his/her account number. A screen snapshot of the start screen is shown in Figure 7.3.

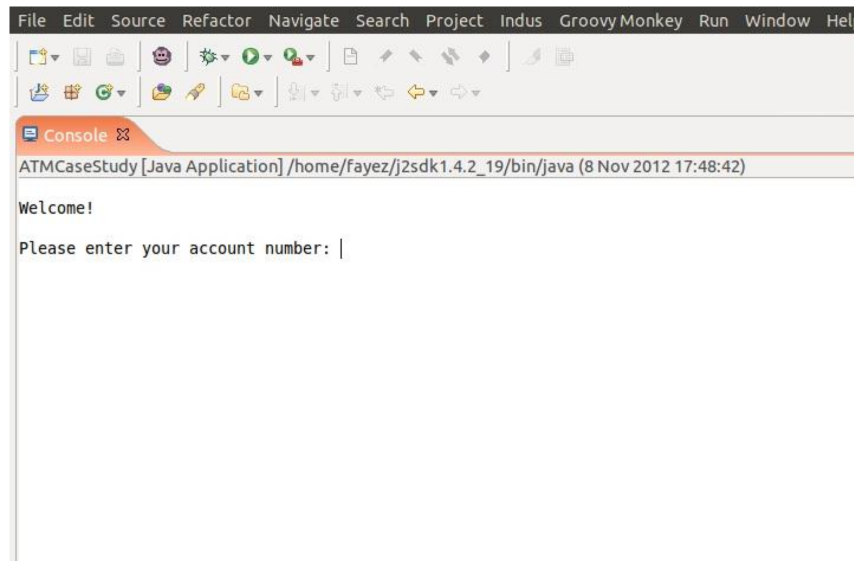

Figure 7.3: An ATM User Welcome Screen

As the above screen snapshot shows, the first functionality of the application is providing a welcome statement to the system user. A welcome statement is considered to be a software feature as the definition of feature can be applied to it:

- A distinctive user-visible aspect
- An increment of program functionality

Users Authentication. The user authentication process is applied within the initial screen of the program display. Firstly, an ATM user is asked to enter his/her

account number and to press the Enter button. Once this entry has been accepted, another line is invoked, which requests the user to enter the bank account PIN. A snapshot of the user authentication process is shown in Figure 7.4.

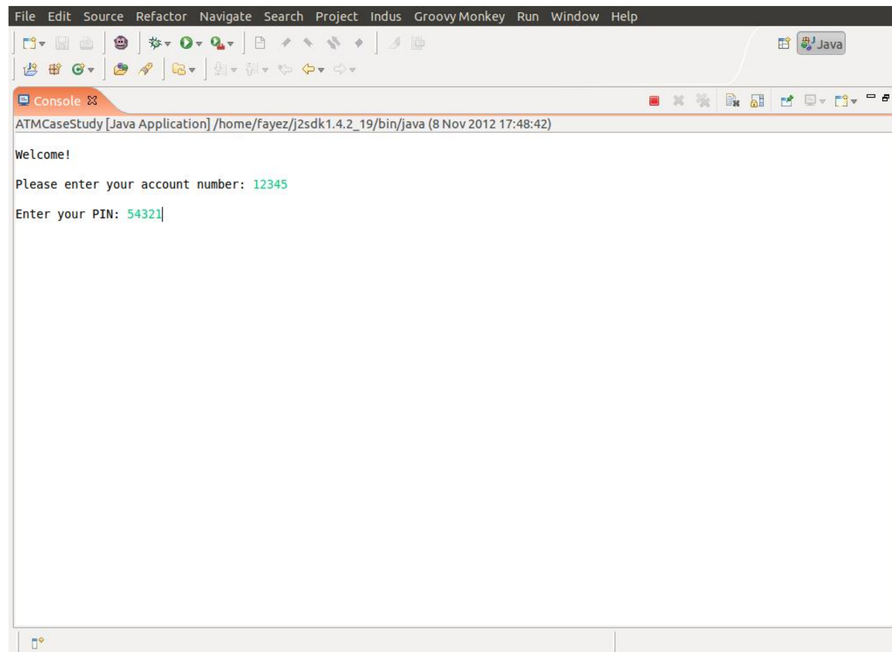

Figure 7.4: User Authentication Screen

The User Authentication process is another feature of the ATM software system. This feature can be defined as "a set of individual requirements". If a user enters invalid data, the authentication process replies with an error message, and ask the user to re-enter his/her data. Figure 7.5 shows the error screen.

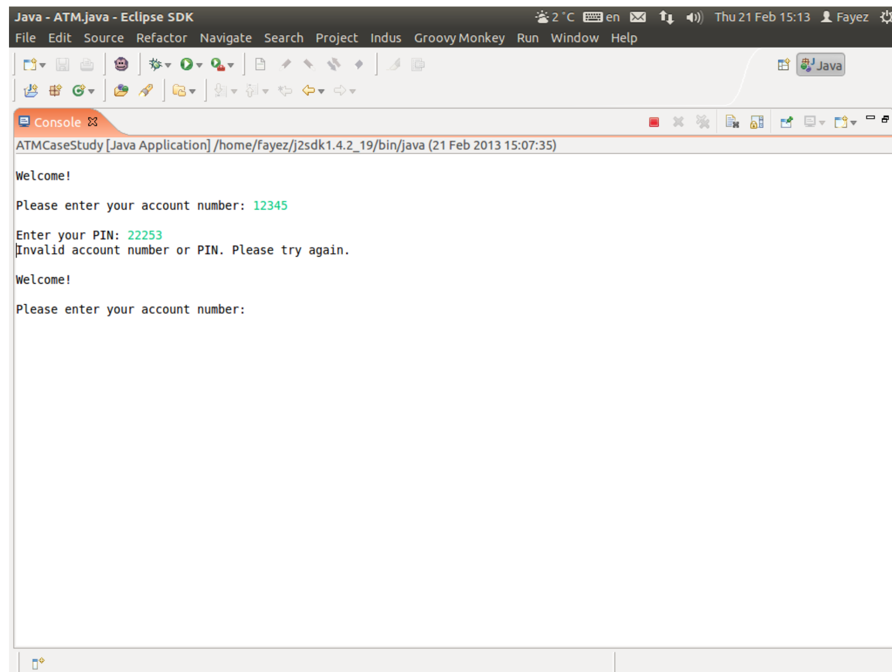

Figure 7.5: Invalid Data Entry Screen

Once the ATM user has entered a valid bank account number and PIN, the software system takes the user to the next screen, which is the main menu screen, as shown in Figure 7.6. The software's main menu screen shows the banking services available to the ATM user. As shown in Figure 7.6, these services are: view the account's balance, withdraw cash, deposit cash and an option to exit the ATM software system. A system user must enter the number corresponding to his/her choice. The Exit choice terminates the validation process as well as the session.

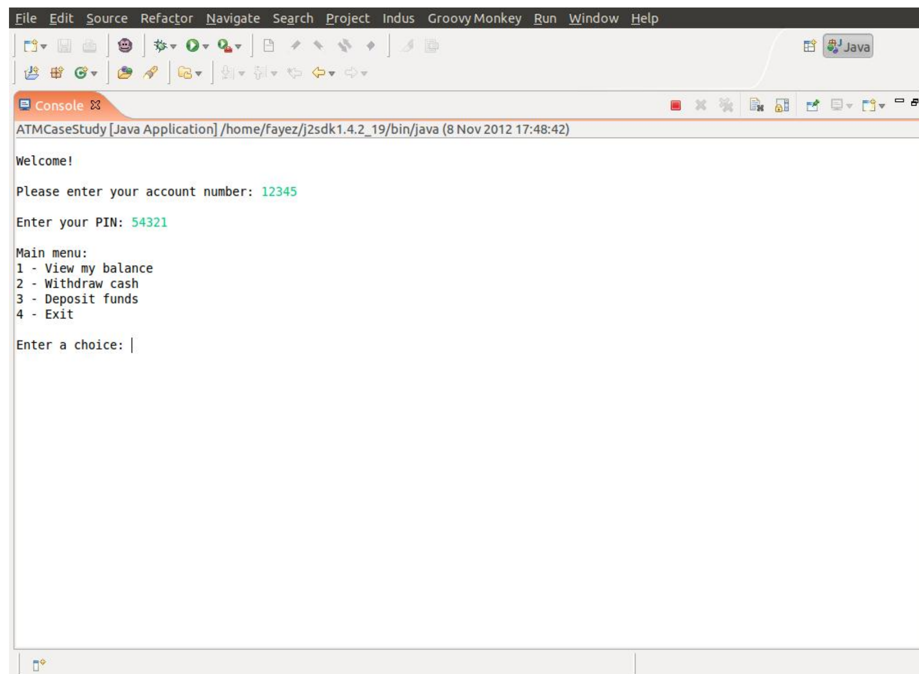

Figure 7.6: Main Menu Screen of the ATM Software System

Statistics table. Statistical information pertaining to the ATM Java source code is shown in Table 7.1. It provides the ATM system metrics, which give the number of classes, methods and total number of lines of code in the ATM software source code used for this case study.

Metric	Total
Number of Classes	12
Number of Methods	20
Lines of code	700

Table 7.1: ATM System Source Code Statistics

The ATM software system application used in this case study is not a complete one; however, it does have the basic concepts and features of an ATM software system. In this case study, the ATM software system application is used merely to

illustrate the proposed feature-based IT business framework.

7.3.2 Business Feature Elicitation Phase

As discussed in the business feature elicitation chapter, this phase covers the business requirements elicitation process. In the proposed framework, the business elicitation process begins with business analysis, which aims to identify the business needs and goals, and to clarify any business changes (usually in relation its operational environment). The business analysis processes was illustrated in Section 4.2 but owing to the limitations of this research, this process will not form part of this case study.

In this phase, the case study begins with the requirement engineering step, which describes the scenario, feature modelling and UML modelling. At the beginning of this step, a scenario is analysed using the Carroll analysis methods previously explained in Chapter 4. In the subsequent step, story cards are created and analysed. Finally, feature cards are produced.

At the end of this phase, features are represented in UML models. These models represent the features from the first phase of the proposed feature-oriented framework, which are later mapped onto the feature models derived from the second phase (the IT feature extraction phase).

7.3.2.1 Scenarios

A scenario is an example of an interactive session; it describes a sequence of actions that relate to real-life examples rather than to abstract descriptions of the functions. To illustrate the proposed approach, a usage scenario is generated. The Scenario is analysed and in this case study the results deliver a set of ATM software features. All these processes are part of the requirement engineering stage.

For the purpose of this case study and as an example, a scenario for an ATM

system is produced, which is shown in Figure 7.7:

The bank customer realizes that she needs some cash but there are no banks nearby. So, she goes to an ATM to withdraw some. She identifies herself to the ATM and indicates that she needs £100 from her bank account. She states that she does not require a receipt for the transaction or to know her account balance. She withdraws her money when it is delivered.

Figure 7.7: ATM Scenario

As illustrated in Chapter 4, a scenario is a description of a high level of abstraction. In the above scenario, the description avoids any discussion about the ATM card, the PIN or any implementation or technical solutions associated with the software system. It is important to separate the user activities from the technology; the user activities remain constant and consistent, regardless of any technological improvement.

Analysis:

For the above scenario, and based on the Carroll method described on Chapter 4, the scenario elements are:

- **Setting:** the environment or context of the user.

- **Actors:** the bank customer, i.e. the ATM user; someone is performing activities or actions.

- **Actions:** the actions the user performs to achieve his/her goal.
- **Events or decisions:** the user makes from the alternatives/options.
- **Goal:** desired outcome of the user who is using the system under consideration.

7.3.2.2 Story Cards

Story cards were illustrated in Chapter 4. As mentioned previously, a story card is a communication technique for user stories. User stories are methods through which user requirements are elicited.

To demonstrate the role of story cards in this case study, a number of story cards have been produced in order to build the ATM software system. Two story cards are shown below; Table 7.2 shows a story card for customer Welcome message, and the second story card (shown in Table 7.3) is for the login screen.

Story Card No. 1	
Description	The system must present a welcome message to bank customers.
Note	Provide a welcome message to the bank customer once an ATM user starts using the system.

Table 7.2: Story Card for Customer - Welcome

Story Card No. 2	
Description	The bank customer must login to perform any bank transactions.
Note	Provide a message for the user to enter an account number and password.

Table 7.3: Story Card for Customer - Login

Analysis:

Chapter 4 indicates that stakeholder story cards can be collected and analysed to extract system features. From the above story cards, ATM software features can be extracted. Some of the features that can be produced from these stakeholder story cards are shown below. Table 7.4 shows a 'welcome screen' feature; it shows the feature name and provides a description of this feature. Table 7.5 shows a 'user authentication' feature; it too shows the feature name and provides a description.

Feature No.	F1 Welcome Screen
Description	Bank customer sees a welcome screen having accessed the system

Table 7.4: Feature 1 - Welcome Screen

Feature No.	F2 Login
Description	Bank customer must be authenticated before any bank transaction is performed.

Table 7.5: Feature 2 - User Authentication

7.3.2.3 Feature Presentation in UML Models

As covered in Chapter 4, the software requirement process deals with managing and developing software requirements; it produces a set of diagrams, algorithms, documentation and other requirement artefacts. Each one of these requirement artefacts is designed to serve a specific goal in software understanding. In this case study, UML is used to describe and organize the software system requirements.

Figure 7.8 shows a class diagram representing the feature of the welcome screen, as described in Table 7.4.

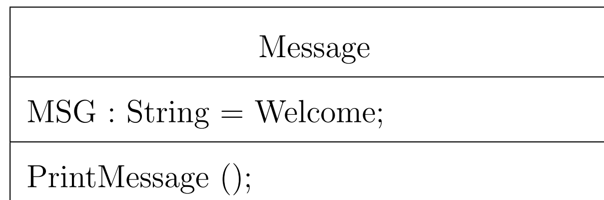

Figure 7.8: Class UML Representation of the Welcome Feature

Another representation of a software feature for this case study is the Login feature. This type of feature entails behavioural activity. It can be presented as a UML activity diagram, as shown in Figure 7.9. The activity diagram changes the state of an ATM authenticated user from the 'not authenticated' to the 'authenticated' state.

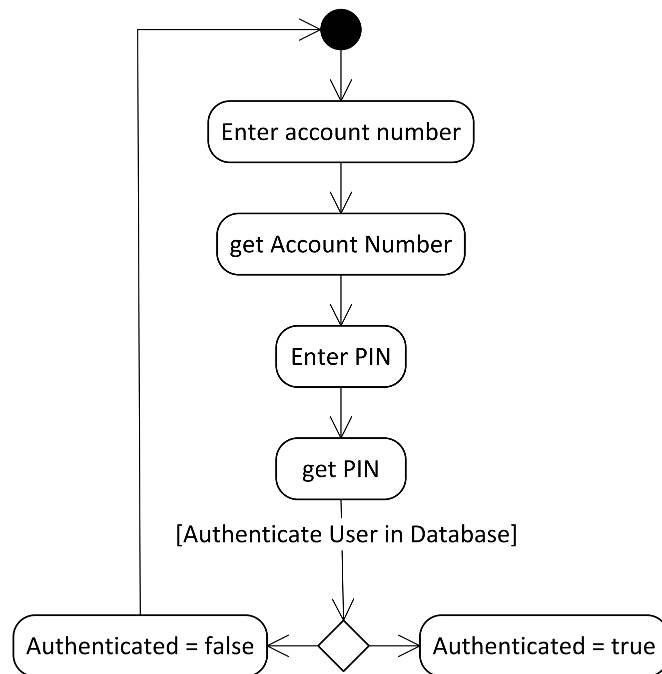

Figure 7.9: Activity UML Representation of 'authenticated user' Feature

7.3.3 IT Feature Extraction Phase

The IT feature extraction phase is mostly a program understanding process. The main objective of a program understanding process is to comprehend the functional aspects of the existing software via extracting relationships from within the source code and then representing these relationships in a diagrammatical view.

The IT feature extraction phase has two main objectives; the first is extracting knowledge from the software program source code, and the second is representing this knowledge as software system features.

The first objective is achieved in a set of steps. These steps are demonstrated in the following two subsections: program slicing, and program dependency and control

flow graphs. The second objective of this phase is to recover the feature structures of the software through feature UML diagrams, as described in Chapter 5.

7.3.3.1 Program Slicing Step

Program slicing was reviewed in the literature review chapter. Moreover, Chapter 5 illustrates the role of program slicing in the IT feature extraction phase. As mentioned previously, program slicing is a program analysis technique that uses program statement dependence information to identify a program's statement relationships, based on an initial program point called a slice criterion point.

The ATM software system used in this case study consists of 12 classes, as shown above in Table 7.1. However, one of these twelve classes is used as a simple class that allows the whole software system to start up. Therefore, it is not included in Figure 7.10, which shows the classes and methods of the ATM software system used in this case study.

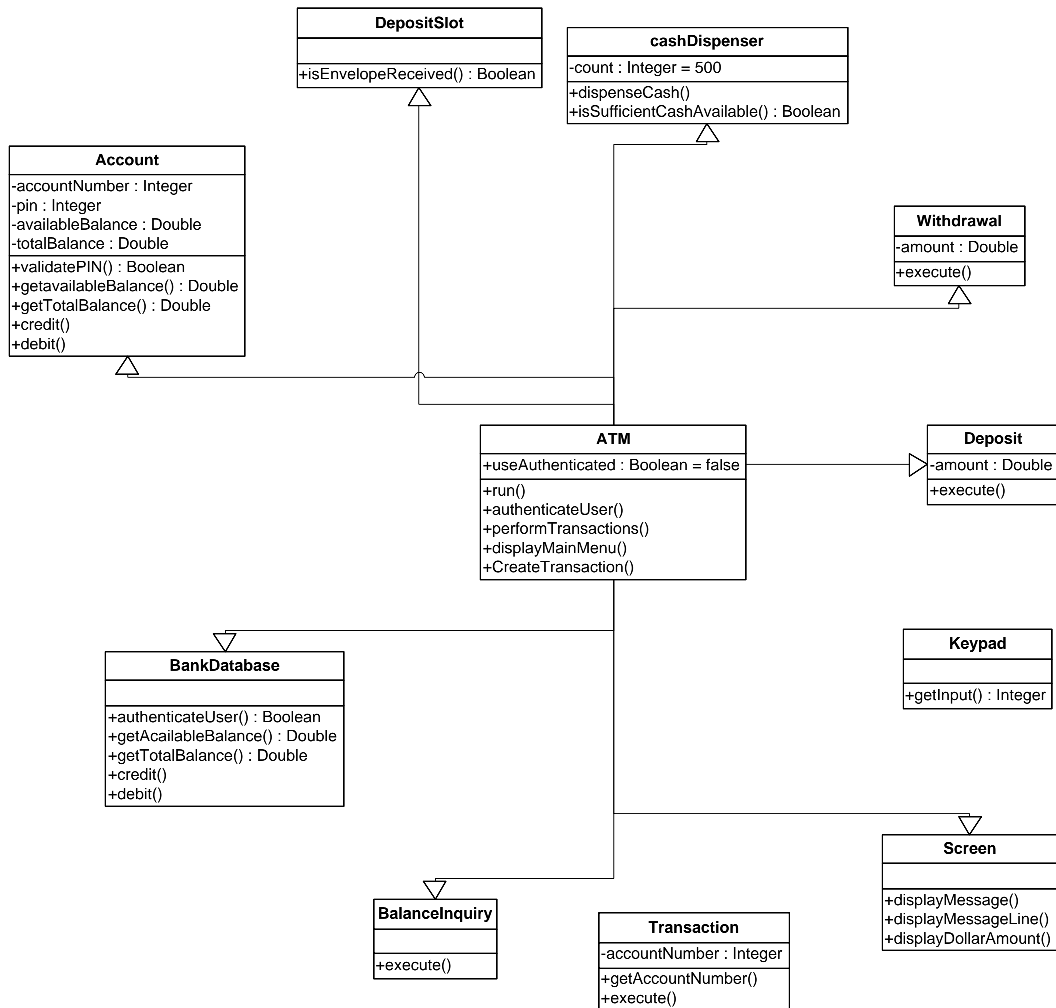

Figure 7.10: ATM Software Classes and Methods

Applying any slicing technique (forwards, backwards or full) on the ATM software system used here produces sets of statements sharing relationships. These sets of statements are called program slices. More details about program slicing were given in the literature review chapter. In this case study, a forward slicing technique and a global variable are chosen to illustrate the proposed framework. The forward

slicing was used because the slice criterion point is located at the beginning of the software program.

A system achieves its functionalities by executing a set of program statements. These statements can be viewed as sending messages between classes, as statements can belong to different classes. The relationships within the source code exist across all the system classes (in order to achieve the system functionalities), as shown in Figure 7.11.

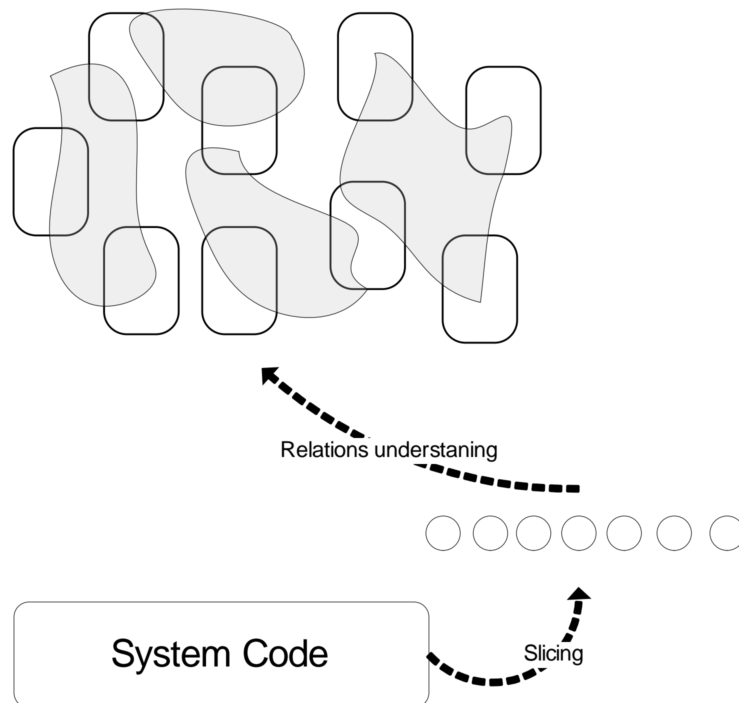

Figure 7.11: Program Slices Across the System

Indus is a static Java program slicing tool; it was discussed in the Tool Support section. Also, Kaveri is an Indus front-end and an Eclipse plug-in, which highlights all the sliced statements of a sliced application source code. By applying the Indus

slicer with the assistant of Kaveri, a set of statements is generated that together represent an ATM application slice.

Figure 7.12 shows a snapshot of the Indus slicer. Also, it shows the slicing criterion point, i.e. *"userAuthenticated = false."*

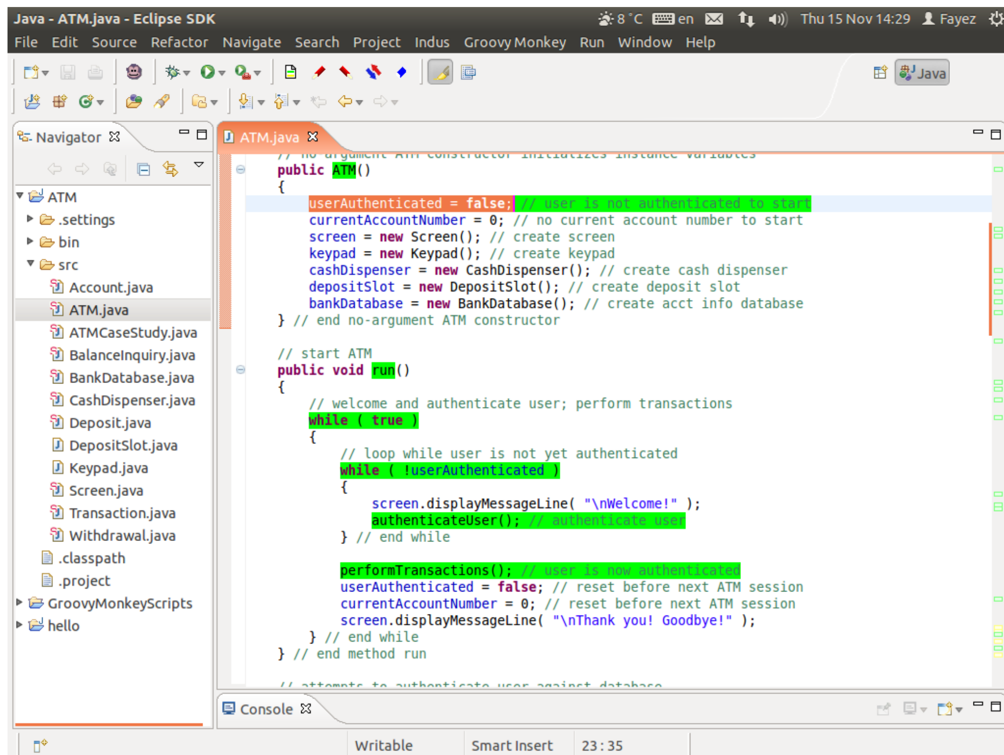

Figure 7.12: Snapshot of the Indus Slicing Tool

The output of the slicing stage produces a set of slices based on slicing criteria points. These slices can be used to extract relationships among these statements and from within a system as a whole. Some of these slices can be used to identify code that is unique to a software system feature. However, these sets of statements need to be presented in graphical form in order to better comprehend the relationships.

After using the program slicing technique, with *"userAuthenticated"* as the slic-

ing criterion point and with forward slicing as the slicing method, the output of the slicing process is shown in the Eclipse framework (Figure 7.12 above). For illustration purposes, part of the source code is shown below in Figure 7.13. The statements that were affected by the forward slicing are shown with a gray background; the others are not part of the slice. The output of all the slicing classes is included as an appendix at the end of this research. Presenting these sets of statements in a graphical form is the next step.

The first method is 'run' in the ATM Class:

```
1.public void run(){
2.    while ( true ){// loop while user is not yet authenticated
3.      while ( !userAuthenticated ){
4.          screen.displayMessageLine( "\nWelcome!" );
5.          authenticateUser(); // authenticate user
        } // end while                              \
6.    performTransactions(); // user is now authenticated
7.    userAuthenticated = false; // reset before next ATM session
8.    currentAccountNumber = 0; // reset before next ATM session
9.    screen.displayMessageLine( "\nThank you! Goodbye!" );
    } // end while
} // end method run
```

The second method is '*authenticateUser*' in the ATM Class:

```
1.    private void authenticateUser(){
2.      screen.displayMessage( "\nPlease enter your account number: " );
```

```
3.      int accountNumber = keypad.getInput(); // input account number
4.      screen.displayMessage( "\nEnter your PIN: " ); // prompt for PIN
5.      int pin = keypad.getInput(); // input PIN
6.      userAuthenticated = bankDatabase.authenticateUser( accountNumber,
    pin );
7.      if ( userAuthenticated ){
         currentAccountNumber = accountNumber; // save user's account #
         } // end if
8.    else
     screen.displayMessageLine( "Invalid account number or PIN. Please
         try again." );
} // end method authenticateUser
```

The third method is '*authenticateUser*' in the BankDatabase Class:

```
1.  public boolean authenticateUser( int userAccountNumber, int userPIN ){
2.  Account userAccount = getAccount( userAccountNumber );
3.  if ( userAccount != null ) return userAccount.validatePIN( userPIN );
4.  else  return false; // account number not found, so return false
      } // end method authenticateUser
```

The fourth method is '*getAccount*' in the BankDatabase Class:

```
private Account getAccount( int accountNumber )
{
    int i;
    for (i=0; i < 1;i++){
        if ( accounts[i].getAccountNumber() == accountNumber )
        return accounts[i];
```

```
    }
    return null;
}
```

The fifth method is '*validatePIN*' in the BankDatabase Class:

```
public boolean validatePIN( int userPIN )
{
if ( userPIN == pin )
return true;
else
return false;
} // end method validatePIN
```

Figure 7.13: Part of The ATM Source Code Shows Sliced Statements

7.3.3.2 Program Dependency and Control Flow Graphs

This step aims to take the output of the program slicing step and to transform it into a new form. These resulting sets of statements are thus presented in program dependency graphs (PDG). The sliced statements from the whole software system are spread across the software system classes, as Figure 7.11 shows.

The next step in of this phase entails representing these program statements in a graphical manner. Figure 7.14 shows part of the PDG of the sliced statements given in the previous example. However, this graph includes statements that are

not in the slice but that are part of the control flow graph (CFG) of the application program. The graph thus shows the statements that are part of the slicing process as white circles, and the ones that are gray are those statements that are not part of the slicing process.

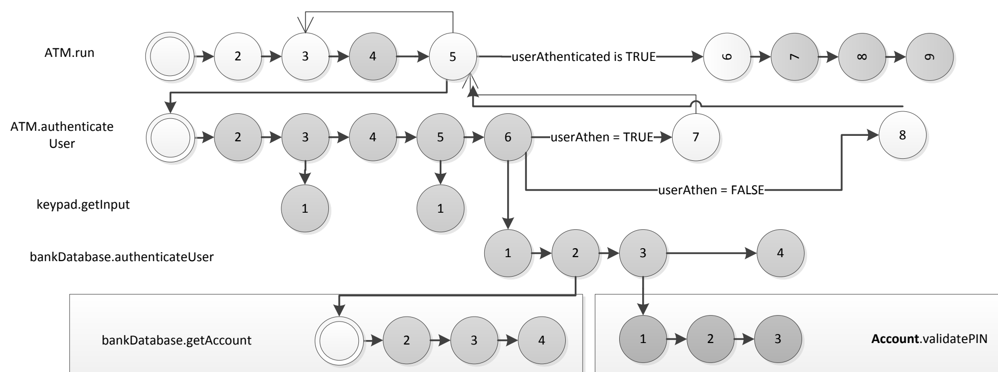

Figure 7.14: CFG and Statements Both Inside and Outside the Slice.

More details on the transformation process to a PDG were given in Chapter 5. The next step entails applying the rule explained in Chapter 5 to extract system features.

7.3.3.3 UML Model Feature Representation

This step takes the graphical representation of the sliced software system source code (and other statements that are part of the CFG) and extracts potential software system features. In Figure 7.14 above, there are two sets of statements that are potential software features. As the proposed rule in Chapter 5 states, feature statements are located in between two other sliced statements.

- The first set of statements that could be a software feature is a simple set,

i.e. consisting of one statement. Statement No. 4 in the ATM.run method, i.e. "*screen:displayMessageLine("Welcome!");*"

• The second feature is a set of statements which is called from the statements number 5 in Figure 7.14 in the ATM.run method. Several other statements could be considered as potential system features in this case, such as statements numbers 7, 8 and 9 in the ATM.run.

The simple statement "screen.displayMessageLine("*nnWelcome!*")" can be represented as a UML class diagram. As discussed in Chapter 5, this process requires the involvement of software developers. Figure 7.15 shows a UML class diagram representation of this single statement, which it represents as a system feature.

On the other hand, the second set of statements from the graph can be represented as a UML activity diagram. Figure 7.16 shows a UML activity diagram representing the user authentication feature.

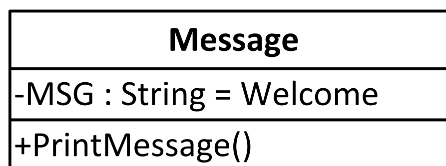

Figure 7.15: Welcome UML Representation

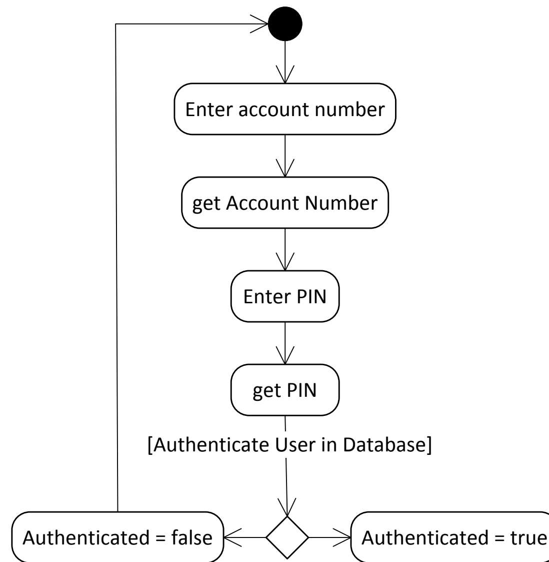

Figure 7.16: UML Activity Diagram of 'authenticate user' Representation

7.3.4 Feature Model Matching Phase

The feature model matching phase is the final one in the proposed feature-oriented framework. It takes two inputs (from the business feature elicitation and the IT feature extraction phases) and matches them.

Based on the model matching algorithm proposed in Chapter 6, the two class models generated from the business elicitation and the IT extraction phases are tested for matching. For the simplicity and to make the illustration clear, a basic and simple example is used.

From the business feature elicitation stage, a message class was generated (shown in Figure 7.8). Another class was generated from the IT feature extraction phase (shown in Figure 7.15). To match these two classes, the matching algorithm proposed in the feature model matching chapter is applied.

7.3.4.1 Matching Algorithm

Based on the algorithm to match two classes of UML models (introduced in Chapter 6), the following steps are undertaken:

• First: the elements of each model are assigned to a set, which in this case are either v_1 or v_2.

v_1 is the class from the business feature elicitation phase; it has two elements: one attribute and one operation.

$$V_1 = \{MSG : String = Welcome, PrintMessage()\}$$

v_2 is the class from the IT feature extraction phase; it also has two elements: one attribute and one operation.

$$V_2 = \{MSG : String = Welcome, PrintMessage()\}$$

• Second: set *Flag-match* to false.

• Third: check if all elements of v_1 exist in v_2; if so, set the *Flag-match* to true. In this case, the *Flag-match* is set to true, as all elements of v_1 exist in v_2.

• Fourth: check if all the elements of v_2 exist in v_1; if so, set the *Flag-match* to true. In this case, the *Flag-match* is set to true, as all elements of v_2 exist in v_1.

• Fifth: check if the *Flag-match* is true; if so, the models x and y are a match. If the *Flag-match* is true in Step 3, then model x is a subset of model y. If the *Flag-match* true in Step 4, then y is subset of x.

For the second example, the activity diagrams in Figures7.9 and 7.16, they can be tested for matching by using other matching techniques. Deissenboecket

al. proposed a CloneDetective approach that uses ideas from graph theory that is applicable to any model represented as a data-flow graph[46]. The cloneDetectibe approach shows that both activity diagrams of Figures7.9 and 7.16 are matched.

7.3.5 Matching Results

ATM software system features such as Login screen, Welcome Message, Transfer, Withdraw and deposit can be matched to business goals. There are three possibilities in business IT features matching results:

- A required business feature exists in software system.
- A required business feature does not exist in software system.
- A required business feature cannot be proven to be exist in software system.

A feature such as transfer money was removed from the ATM system. The bottom-to middle approach failed to extract this feature. However, this feature is part of the business requirement goals. Therefore, such a feature cannot be matched. One of the benefits the business-IT feature-based framework able to provide is the efficiency feature allocation. Such a benefit is very important during software evolution and software testing.

7.4 A Library Management System

A library management system is the second case study used to demonstrate the proposed approach of this research. The library management system is free to use for academic purposes and available to download from Planet-Source-Code.com. It was written in Java and it supports all the main functions needed in a library.

The main purpose of a regular library management system is to manage the status of books, where each one has a unique identification code. Besides books, a

library system deals with library members who are able to borrow books based on a set of rules. Books and members can be added to, removed from or modified in the system by library staff. Moreover, librarians can search for books and library members. Books can be searched for by using the book title, author, ISBN or other criteria. In addition, members can be searched for by name, address or other criteria.

7.4.1 The Library Management Software System Application

Statistics table. Statistical information pertaining to the library management system's Java source code is shown in Table 7.6. It provides the library management system metrics, which give the number of classes, methods and total number of lines of code in the library management software source code used for this case study.

Metric	Total
Number of Classes	29
Number of Methods	88
Lines of code	4500

Table 7.6: Library Management System Source Code Statistics

The library management system's application software used in this case study is not complete; however, it does have the basic concepts and features of a library management software system. In this case study, the library system's application software is used merely to illustrate the benefits of the proposed feature-based IT business framework.

7.4.2 Business Feature Elicitation Phase

This phase works as a process to elicit the business feature requirements. More details were given and discussed in the business features chapter and in the previous case study, the ATM.

As discussed in the previous case study regarding the limitations of this research, this case study begins with the requirements engineering step. During this step, scenario, feature modelling and UML modelling will be studied and analysed.

7.4.2.1 Scenario

A scenario is narrative or a written story that explains how a user or users interact with the system; it shows how a person uses a product or service in real-life examples rather than through abstract descriptions of the functions. To describe the proposed approach, a scenario is thus produced. Library software system features are elicited through analysing the scenario. A library book-borrowing scenario is shown in Figure 7.17

> A library member identifies himself to the librarian and presents one or more books. The librarian conducts a search for each book to make sure it can be loaned. If the search for a book fails, i.e. the book is not in the system, he adds the book's details into the system. Moreover, the librarian checks for membership validity. If acceptable, the librarian stamps each book with a return date.

Figure 7.17: : Library Management System Scenario

Figure 7.17 shows a book-borrowing scenario. As chapter 4 illustrated, a scenario shows a description of a high level of abstraction; thus, for example, details about the type of a member's identification is not discussed. As mentioned previously, in a scenario, it is important to separate activity from technology.

Analysis: For the book-borrowing scenario, and based on the Carroll method described on Chapter 4, the scenario elements are:

- **Setting:** the environment or context of a library.
- **Actors:** there are at least two actors in this scenario: library member and librarian.
- **Actions:** the actions a user performs to achieve his or her goals.
- **Events or decisions:** those that the system user makes from the alternatives or options.
- **Goal:** desired outcome of a user who is using the system under consideration.

7.4.2.2 Story Cards

In chapter 4, a story card is described as a communication technique for a user story. It is a method by which users' requirements are elicited. For a library management system, a story card (add a book) is generated as shown in Figure 7.7.

Analysis:

Story cards provide a path for extracting system features. However, these story cards must be analysed before extracting any system features; Chapter 4 provides

Story Card No. 1	
Description	The system must be able to store books details and confirm that book has been added
Note	Provide a function to enable books to be added to system and show a confirmation message

Table 7.7: Story Card for Library Management System - Add Book And Confirm

more details about story card analysis. From the story card 'add book', there are two features: first, an 'add book' feature and second, a 'confirm book added' feature. A system feature confirming addition can be provided, as shown in Table 7.8.

Feature No.	F1 Confirm Add book
Description	Librarian is able to see a confirmation message that a book has been added.

Table 7.8: Feature 1 - Confirm Add Book

7.4.2.3 Feature Presentation in UML Models

The final step of this phase is to represent the system features as a UML representation. Besides features representation, the software requirements process produces a set of artefacts pertaining to the system under consideration. These artefacts can be a diagrams, algorithms and system documentation. Each product of the requirements engineering process serves a specific purpose. In the library management system case study presented here, UML representation describes system

requirements.

Figure 7.18 shows a class diagram of the Confirm Add Book feature which was described in Table 7.8.

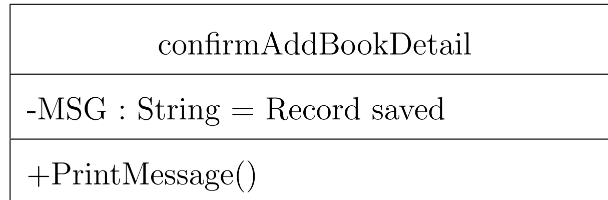

Figure 7.18: : UML Class for Confirm Add Book Confirmation Message

7.4.3 IT Feature Extraction Phase

The second phase of the proposed framework is a reverse engineering process; it is a bottom-to-middle process. A program understanding technique is used in this phase. Software source code is sliced in order to extract code relationships. These relationships are transformed into diagrammatical views. On these diagrammatical views, the proposed algorithm is applied to extract system features. Chapter 5 illustrates more of this phase of the proposed framework. In general, there are two main objectives to this phase. The first is extracting knowledge from the software source code by presenting the relationships among the source code statements. The second is presenting this knowledge as software features.

7.4.3.1 Program Slicing Step

Program slicing was described in the literature review chapter (Chapter 2), as a program understanding technique for extracting software code relationships.

The library management system used in this case study consists of 29 classes, 88 methods and about 4,500 lines of code. The classes are shown in Table 7.6.

A slice is a set of program code statements sharing relationships. A slice can be produced by applying a program-slicing technique (forwards, backwards or full).

Program-slicing was also covered in the literature review chapter. The library management system, like all other systems, achieves its functions by executing a set of statements. Therefore, program-slicing is used to extract these program code statements relating to the system features. Figure 7.19 shows part of the Addbooks class of the library management system.

AddBooks Class:

```
public class AddBooks extends JFrame implements ActionListener
{
    TextField book_id,book_title,author,year,available,total,category;

public void actionPerformed(ActionEvent ae)
{
    if(ae.getSource()==b)
    {
       try
    {
      String a=book_id.getText();
       String b=book_title.getText();
       String c=author.getText();
       String d=year.getText();
       String e=available.getText();
       String f=total.getText();
       String g=category.getText();
       int di=Integer.parseInt(d);
       int ei=Integer.parseInt(e);
```

```
       int fi=Integer.parseInt(f);
       Class.forName("com.mysql.jdbc.Driver");
       Connection con=DriverManager.getConnection
           ("jdbc:mysql:///Library_Database","root","");
       Statement st=con.createStatement();
       String query="insert into Books
           values('"+a+"','"+b+"','"+c+"',"+di+","+ei+","+fi+",'"+g+"');";
      System.out.println(query);
       int x=st.executeUpdate(query);
         if(x>0)
       {
          //System.out.println("record saved");
           JOptionPane.showMessageDialog(AddBooks.this,"Record saved");
       }
   } catch(Exception ex)
       {
           System.out.println(""+ex.getMessage());
       }
  }
  if(ae.getSource()==b2)
    {
        f.setVisible(false);
    }
 }

 public static void main(String args[])
 {
        AddBooks add=new AddBooks();
```

```
  }
}
```

Figure 7.19: Part of the Library Management System Source Code

7.4.3.2 Program Dependency and Control Flow Graphs

The purpose of the IT feature extraction step is to take the results of the previous step (program-slicing), and to transform them into a visual representation form. In this case study, the visual representation is PDG form. Figure 7.20 shows the Control Flow Graphs (CFG) of the results of the previous step (program-slicing). The whole CFG is represented; however, the circles are coloured depending on which statements are part of the slice and which are not. The white ones are those statements that are part of the CFG and part of the slice as well, and the grey ones are just part of the slice. More details of the transformation process were illustrated in the IT feature extraction chapter (Chapter 5).

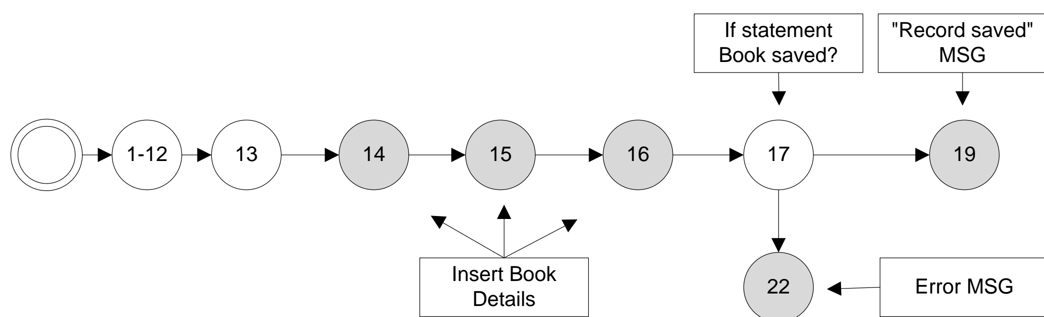

Figure 7.20: CFG Includes Statements Both Inside and Outside the Slice.

7.4.3.3 UML Model Feature Representation

The purpose of the UML feature representation step is to extract system features from the CFG presented in the previous step. This step represents these features as UML diagrams, which are an input for the next step.

In the Figure 7.20 shows potential software features. As proposed in Chapter 5, potential features are located in between two other sliced statements. Three potential features can be considered:

1. The first feature to be considered is Add Book, which can be seen in statements 14, 15 and 16. These three statements insert book details into the library database.

2. The second feature to be considered is a message statement indicating that book details have been saved.

3. The third feature is similar to the second one. However, it shows that an error has occurred and that a book is not been added to the library system.

From the three potential features, the second one is considered "(AddBooks.this,"Record saved");". This feature can be represented as a UML class diagram. As discussed in the IT feature extraction chapter (Chapter 5), this process requires a software engineer's involvement. Figure 7.21 is a representation of the above feature as a UML class diagram.

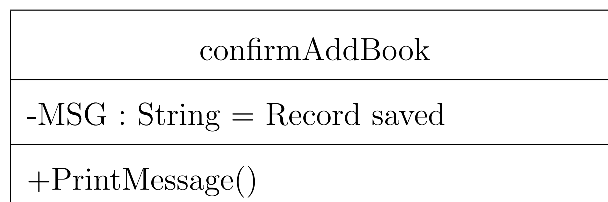

Figure 7.21: UML Class for add Book Confirm Message

7.4.4 Feature Model Matching Phase

The outputs of the previous two phases, the business feature elicitation and the IT feature extraction phase, are the input for this final phase. The UML diagrams are matched in this phase to show matching, existing or missing features. The model-matching algorithm proposed in Chapter 6 is applied in this phase. The two UML models produced from the previous two phases are used as an example to apply and test the matching algorithm. These two UML models are shown in Figure 7.18 and Figure 7.21.

7.4.4.1 Matching Algorithm

To test the algorithm proposed in the feature model matching chapter, certain algorithm steps are undertaken.

1. Elements of each UML model in Figur 7.18 (x) and Figure 7.21 (y) is assigned to a set $v1$ and $v2$ respectively.

   $v1$ is a set from the class was generated from the business feature elicitation phase; it has two elements: one attribute and one operation.

   $$v1 = \{\text{MSG : String = Record saved; PrintMessage()}\}$$

   $v2$ is a set from the class was generated from the IT feature extraction phase; it also has two elements: one attribute and one operation.

   $$v2 = \{\text{MSG : String = Record saved; PrintMessage()}\}$$

2. Set Flag-match to false.

3. Check if all elements of $v1$ exist in $v2$; if so, set the Flag-match to true. In this case, the Flag-match is set to true, as all elements of v1 exist in $v2$.

4. Check if all the elements of $v2$ exist in $v1$; if so, set the Flag-match to true. In this case, the Flag-match is set to true, as all elements of $v2$ exist in $v1$.

5. Check if the Flag-match is true; if so, the models x and y are a match. If the Flag-match is true in Step 3, then model x is a subset of model y. If the Flag-match true in Step 4, then y is subset of x.

The model match algorithm proposed shows that both models from both proposed framework are matched. This match prove that feature 'confirm Add book' is implemented in the library management software source code.

7.4.5 Matching Results

Library Management System usually comes with several features to achieve its goals. Features such as add book, delete book, add member and loan a book can be matched to the system's goals.

Adding or removing features from the software system was detected in the proposed framework. Features can be removed from both sides of the framework (business goals or IT side). Such ability of detection provide a powerful approach especially in software evolution.

7.5 Evaluation and Discussion

In these two case studies, the proposed Business-IT feature-based framework was tested to produce a match between features from the business side and features from the IT side.

The framework begins with software a requirement engineering (RE) process; this represents the first part of the first phase. The first phase is designed to elicit

software features from the upper level of the proposed framework (the business feature elicitation phase). This phase is called an upper-to-middle process as it follows the software lifecycle process from the upper to the middle part of the cycle.

The second phase (the IT feature extraction) is designed to extract software features from the bottom level of the framework. In the middle of the framework is the feature model-matching phase. Features from the first and second phases were subsequently tested for matching.

An ATM (Automated Teller Machine) was the first case study presented in this work; it was introduced in Section 7.3. The first phase of the proposed framework (Business Feature Elicitation phase) was applied to produce two ATM features, i.e. the 'welcome message' and 'login' features. These two features were transformed into UML models representing software system features, as shown in Figures 7.4 and 7.5. This phase is an upper-to-middle process, as shown in the proposed framework in Chapter 3.

IT Feature Extraction is the second phase of the framework; it was applied on the ATM software source code. The ATM software system is a program of 700 lines of code, as shown in Table 7.1. A static slicing tool was used to generate each program slice; a slice is a set of statements sharing a relationship. This set of statements was transformed into CFG and PDG representations. The methods proposed in Chapter 5 were used to generate system features from the CFG and PDG representations. Two features were generated and modelled: as a class UML diagram and an activity diagram. These two UML models are a 'welcome message' class diagram and a 'login' activity diagram.

The final phase of the framework is feature model matching. It takes models from both previous phases and applies the algorithm proposed in Chapter 6 to generate a matching result.

The second case study was a library management system, presented in Section 7.4. It is a software system with 4,500 lines of code, as shown in Table 7.6. The Business-IT feature-based framework was tested on the library management system. Features were generated from both the upper-to-middle and the bottom-to-middle phases of the framework, and were matched in the third (middle) phase.

Scalability is defined in terms of the aspect of the product that needs to be developed. It describes the product's ability to handle the complexity of the problem addressed. On the other hand, software evaluation scalability describes the effectiveness of the approaches or methodologies for handling increasing amounts of work [70].

Two case studies were presented in the case study chapter; the ATM and the Library Management system. The ATM software contains about 700 lines of code whereas the Library Management system contains more than 4,500 lines of code. The proposed approach (Business-IT feature-based framework) was applied on both software systems and yet it was able to produce similar results. Thus, the size of the software system has no noticeable effect of the proposed framework.

7.6 Conclusion

The proposed model-matching algorithm shows that both models from the proposed framework are matched. This match proves that the feature Confirm Add Book is implemented in the library management software's source code.

At the beginning of this case study chapter, an overview of an ATM system, the system's functions, source code and metrics were given. Following the overview of the ATM system, a short description of the tools used in this case study was provided; these include Eclipse, and the Indus slicer and Kaveri as Eclipse plug-ins.

Following this overview, the first phase of the proposed framework was demonstrated. Initially, scenarios and story cards were presented and then analysed in order to elicit system features. Following this, feature cards were produced and then transformed into UML models. Two feature-based UML models were produced: the Welcome screen and the Login screen.

In the second phase (IT feature extraction), the ATM Java application source code was presented. Indus and Kaveri were used to slice the software system source code. This process generated a slice that is a set of related statements. From these statements, a program dependency graph was built, which included a control flow diagram. The graph generated presents all the statements, whether or not they were in the slicing set.

The rule proposed in Chapter 5 (on feature extraction) was applied on the generated graph, and this rule delivered some software system features. Two of these features were considered: the Welcome message and the user Login features. At the

end of this stage, the features were modelled in UML; the first as a class diagram and the second as an activity diagram.

The third phase was the feature model matching. It took the two models of the previous phases and sought to make a match. As described in Chapter 6, the proposed algorithm for matching these two models was applied. The proposed algorithm matched the two UML class diagrams, i.e. those generated from the first and second phases. For the second example, The CloneDetective approach proposed by Deissenboecket shows that the UML activity diagrams matched as well.

Chapter 8

Conclusion And Future Work

Objectives:

- To give a summary for the Thesis
- To provide a statement of evaluation
- To provide research limitations and future works

8.1 Summary of the Thesis

This chapter provides a summary of the work conducted in this thesis. Chapter 1 gave an introduction to the research; it described the problem to be addressed and listed the research questions. Also, it provided the thesis structure. Chapter 2 was a background study on the feature-oriented domain, describing the problem and solution domains as well as other related subjects. It reviewed many related works, basic concepts and definitions considered fundamental to this research, such as the concept of features, feature-oriented software development, feature modelling, software evolution, UML, program slicing, requirement engineering and model comparison.

The third chapter reviewed work related to the proposed feature-oriented business IT framework. It illustrated three main concepts: forward engineering, reverse engineering and model comparison. This chapter proposed a novel conceptual feature-based business IT framework to map the problem domain onto the solution domain.

In Chapter 4, the first phase of the proposed framework was covered. It is a top-to-middle process, and it begins with business analysis, passes through software requirement engineering and ends with software system modelling.

The second phase of the proposed feature-based framework was covered in Chapter 5. This is a bottom-to-middle process, and it mainly involves reverse engineering. It starts with the source code of the software system, which is then reverse engineered and modelled in UML diagrams.

Chapter 6 deals with the final phase of the proposed feature-oriented frame- work, wherein models from the first and second phases are mapped. Model-matching is thus conducted between the two models in order to identify links between the outputs from the first and second phases (business feature elicitation and IT feature extraction).

Before the final chapter of this thesis, a case study was presented in Chapter 7 to demonstrate that the proposed approach is indeed a useful construct. Besides the case study, this chapter presented and discussed the tools that can be used to support the implementation of the proposed approach.

8.2 Research Questions Revisited

The research questions are here revisited in order to evaluate the significance of this work and of the contributions to the field that were claimed in the first chapter. The main research question presented in Chapter 1 was:

- Is it possible to trace business goals to the software features at the source code level?

The main research question was generally answered through proposing the feature-oriented business IT framework. The novelty of the proposed framework emanates from the three characteristics of the business-IT gap: the problem domain, the solution domain and the matching process.
A set of sub-questions were then introduced as a result of posing the main research question:

- Does the software satisfy the stakeholders' requirements?
- Can the efficiency of the software system be improved during the software evolution process?

These questions were answered in Chapter 6, wherein model representations from the top-to-middle process are mapped to models from the bottom-to-middle process,

i.e. identifying and linking the business and the IT domains.

8.3 Future Work

The IT-business gap is challenging; it is a dynamic gap and it is unlikely to be fully addressed for many reasons. Each reason represents an area for possible research; for example:

- Business and IT professionals are often suspicious of each other and lack the ability to work together in open teamwork.
- Business and IT professionals do not communicate in the same language, e.g. IT professionals tend to use technical terms that cannot be understood by business professionals, and verse versa.
- Business and IT professionals do not share the same goal; business managers often set requirements for the IT team that are impossible to achieve.

The first phase of the research approach entails taking the business domain closer to the IT domain by more fully comprehending the business needs; this is achieved through the business analysis processes but various other approaches could be utilized in this, e.g. identifying and then sharing a common language.

The IT feature extraction phase is a reverse engineering process, which is a very challenging area in software engineering. It is the process of extracting knowledge from software source code. This research used static program slicing as a technique for the purpose of program understating but other program slicing techniques for this purpose are available, such as dynamic slicing.

The third phase of this research is feature model matching. Models of UML class diagrams are used to demonstrate the mapping technique; however, other models could be used to match the features of the business and IT domains. UML activity

diagram is one such example; for UML activity diagrams, the area of graph theory is promising in attempting to link these types of models. Finally, to gain a better understanding of the proposed approach, real systems in the industrial or commercial fields could be used as additional case studies.

Bibliography

[1] G. D. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. In *SIGSOFT FSE*, pages 9–20, 1993.

[2] V. K. Ahmed Saleem Abbas, W. Jeberson. The need of re-engineering in software engineering. *International Journal of Engineering and Technology Volume 2 No. 2, February, 2012*, 2, 2012.

[3] A. Akundi, F. Zapata, and E. Smith. UML profile and extensions for complex approval systems with complementary levels of abstraction. *Procedia Computer Science*, 12(0):75 – 80, 2012. Complex Adaptive Systems 2012.

[4] M. Alanen and I. Porres. Difference and union of models. In P. Stevens, J. Whittle, and G. Booch, editors, *UML*, volume 2863 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2003.

[5] M. Alanen and I. Porres. Version control of software models. *Advances in UML and XML-Based Software Evolution*, pages 47–70, 2005.

[6] K. Altmanninger, G. Kappel, A. Kusel, W. Retschitzegger, M. Seidl, W. Schwinger, and M. Wimmer. AMOR - Towards Adaptable Model Versioning. In *1st International Workshop on Model Co-Evolution and Consistency Management (MCCM'08), Workshop at MODELS'08*, Toulouse, France, 2008.

[7] S. Apel and C. Kästner. An overview of feature-oriented software development, 2009.

[8] S. Apel, C. Lengauer, B. Moller, and C. Kästner. An algebra for features and feature composition. In J. Meseguer and G. Rosu, editors, *AMAST*, volume 5140 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2008.

[9] A. Arsanjani. *Grammar-Oriented Object Design: Towards Dynamically Reconfigurable Business and Software Architecture For On-demand Computing*. PhD thesis, De Montfort University, 2003.

[10] T. Ball and S. G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, Apr. 1996.

[11] E. J. Barry, C. F. Kemerer, and S. Slaughter. How software process automation affects software evolution: a longitudinal empirical analysis. *Journal of Software Maintenance*, 19(1):1–31, 2007.

[12] D. S. Batory. Feature models, grammars, and propositional formulas. In J. H. Obbink and K. Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[13] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 3:225–252, 1976.

[14] S. Bentrad and D. Meslati. Visual Programming and Program Visualization Towards an Ideal Visual Software Engineering System. *International Journal on Information Technology*, 1(3):7, December 2011.

[15] J.-F. Bergeretti and B. A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, Jan. 1985.

[16] B. S. Blanchard. *System Engineering Management*. John Wiley & Sons, 2012.

[17] G. Booch, J. Rumbaugh, and I. Jacobson. *The unified modeling language user guide*. Addison-Wesley, Upper Saddle River, NJ, 2005.

[18] T. Bowen, F. Dworack, C. Chow, N. Griffeth, G. Herman, and Y.-J. Lin. The feature interaction problem in telecommunications systems. In *Software Engineering for Telecommunication Switching Systems, 1989. SETSS 89., Seventh International Conference on*, pages 59 –62, jul 1989.

[19] K. Brennan. *A Guide to the Business Analysis Body of Knowledge (Babok Guide)*. International Institute of Business Analysis, 2009.

[20] J. K. A. R. Brian Berenbach, Daniel Paulish. *Software & Systems Requirements Engineering: In Practice*. 2009.

[21] M. L. Brodie and M. Stonebraker. *Migrating legacy systems: Gateways, interfaces & the incremental approach*. The Morgan Kaufmann series in data management systems. Kaufmann Publ., San Francisco, Calif, 1995.

[22] F. Brooks. *The mythical man-month*. Addison-Wesley, 1995.

[23] J. Brooks, F.P. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10 –19, april 1987.

[24] C. Brun and A. Pierantonio. Model differences in the eclipse modelling framework. In *The European Journal for the Informatics Professional*, 2008.

[25] A. Bryant. It's engineering jim ... but not as we know it: software engineering - solution to the software crisis, or part of the problem? In C. Ghezzi, M. Jazayeri, and A. L. Wolf, editors, *ICSE*, pages 78–87. ACM, 2000.

[26] S. Budhkar and D. A. Gopal. Article: Reverse engineering java code to class diagram: An experience report. *International Journal of Computer Applica-*

tions, 29(6):36–43, September 2011. Published by Foundation of Computer Science, New York, USA.

[27] E. Buss and J. Henshaw. A software reverse engineering experience. In *CASCON First Decade High Impact Papers*, CASCON '10, pages 42–60, Riverton, NJ, USA, 2010. IBM Corp.

[28] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1), 2003.

[29] A. Caplinskas and O. Vasilecas. Information systems research methodologies and models. In *Proceedings of the 5th international conference on Computer systems and technologies*, CompSysTech '04, pages 1–6, New York, NY, USA, 2004. ACM.

[30] A. Charfi, C. Mraidha, S. Gérard, F. Terrier, and P. Boulet. Toward optimized code generation through model-based optimization. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1313–1316, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[31] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In S. Ossowski and P. Lecca, editors, *SAC*, pages 1284–1291. ACM, 2012.

[32] K. Chen, W. Zhang, H. Zhao, and H. Mei. An approach to constructing feature models based on requirements clustering. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, RE '05, pages 31–40, Washington, DC, USA, 2005. IEEE Computer Society.

[33] P. Chen, M. Critchlow, A. Garg, C. van der Westhuizen, and A. van der Hoek. Differencing and merging within an evolving product line architecture. In F. van der Linden, editor, *PFE*, volume 3014 of *Lecture Notes in Computer Science*, pages 269–281. Springer, 2003.

[34] Z. Chentouf, S. Cherkaoui, and A. Khoumsi. Experimenting with feature interaction management in sip environment. *Telecommunication Systems*, 24(2-4):251–274, 2003.

[35] E. J. Chikofsky and J. H. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[36] L. Chung and J. Prado Leite. On non-functional requirements in software engineering. In A. Borgida, V. Chaudhri, P. Giorgini, and E. Yu, editors, *Conceptual Modeling: Foundations and Applications*, volume 5600 of *Lecture Notes in Computer Science*, pages 363–379. Springer Berlin Heidelberg, 2009.

[37] A. Classen, P. Heymans, and P.-Y. Schobbens. What's in a feature: a requirements engineering perspective. In *Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering*, FASE'08/ETAPS'08, pages 16–30, Berlin, Heidelberg, 2008. Springer-Verlag.

[38] R. Clayton, S. Rugaber, and L. M. Wills. On the knowledge required to understand a program. In *WCRE*, pages 69–78, 1998.

[39] S. Cook. Looking back at uml. *Software & Systems Modeling*, 11:471–480, 2012.

[40] G. Crnkovic. Constructive research and info-computational knowledge generation. In L. Magnani, W. Carnielli, and C. Pizzi, editors, *Model-Based Rea-*

soning in Science and Technology, volume 314 of *Studies in Computational Intelligence*, pages 359–380. Springer Berlin Heidelberg, 2010.

[41] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[42] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. W käsowski. Cool features and tough decisions: a comparison of variability modeling approaches. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 173–182, New York, NY, USA, 2012. ACM.

[43] R. Davies. The power of stories. *XP*, 2001.

[44] A. M. Davis. *Software requirements - objects, functions, and states.* Prentice Hall international editions. Prentice Hall, 1993.

[45] F. Deissenboeck, B. Hummel, E. Jürgens, M. Pfaehler, and B. Schätz. Model clone detection in practice. In K. Inoue, S. Jarzabek, R. Koschke, and J. R. Cordy, editors, *IWSC*, pages 57–64. ACM, 2010.

[46] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In H. Giese, M. Huhn, U. N. 0002, and B. Schätz, editors, *MBEES*, volume 2008-2 of *Informatik-Bericht*, pages 57–67. TU Braunschweig, Institut für Software Systems Engineering, 2008.

[47] P. Deitel and H. Deitel. *Java: How to Program.* How to Program. Prentice Hall, 2011.

[48] Dictionary.com. http://dictionary.reference.com.

[49] R. Dijkman, M. Dumas, B. Dongen, R. Käärik, and J. Mendling. Similarity of Business Process Models: Metrics and Evaluation. *Information Systems*, 36(2):498–516, 2011.

[50] A. Doan, J. Madhavan, P. Domingos, and A. Halevy. Ontology matching: A machine learning approach. In *Handbook on Ontologies in Information Systems*, pages 397–416. Springer, 2003.

[51] L. Du and P. Cai. A survey on applications of program slicing. In J. Luo, editor, *Soft Computing in Information Communication Technology*, volume 158 of *Advances in Intelligent and Soft Computing*, pages 215–220. Springer Berlin Heidelberg, 2012.

[52] R. Dupuis, P. Bourque, A. Abran, J. W. Moore, and L. L. Tripp. The SWEBOK Project: Guide to the software engineering body of knowledge, May 2001. Stone Man Trial Version 1.00, http://www.swebok.org/ [01/12/2003].

[53] H. Eichelberger and K. Schmid. Guidelines on the aesthetic quality of uml class diagrams. *Information & Software Technology*, 51(12):1686–1698, 2009.

[54] S. Faily and I. Fléchais. A meta-model for usable secure requirements engineering. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, SESS '10, pages 29–35, New York, NY, USA, 2010. ACM.

[55] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[56] E. Foundation. About the eclipse foundation.

[57] M. Fowler and K. Scott. *UML distilled (2nd ed.): a brief guide to the standard object modeling language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[58] G. Gabrysiak, H. Giese, A. Seibel, and S. Neumann. Teaching requirements engineering with virtual stakeholders without software engineering knowledge. In *Requirements Engineering Education and Training (REET), 2010 5th International Workshop on*, pages 36 –45, sept. 2010.

[59] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume I. River Edge, NJ: World Scientific Publishing Company, 1993.

[60] M. Girschick and T. Darmstadt. Difference detection and visualization in UML class diagrams, 2006.

[61] O. C. Z. Gotel and C. W. Finkelstein. An analysis of the requirements traceability problem. In *International Conference on Requirements Engineering*, pages 94–101, 1994.

[62] D. A. Gretar Tryggvason. *Shaping Our World: Engineering Education for the 21st Century*. ohn Wiley & Sons, 2011.

[63] Ä. Hajnal and I. Forgäcs. A demand-driven approach to slicing legacy cobol systems. *Journal of Software: Evolution and Process*, 24(1):67–82, 2012.

[64] M. Harman. Why source code analysis and manipulation will always be important. In *SCAM*, pages 7–19. IEEE Computer Society, 2010.

[65] H. Holbrook, III. A scenario-based methodology for conducting requirements elicitation. *SIGSOFT Softw. Eng. Notes*, 15(1):95–104, Jan. 1990.

[66] M. E. C. Hull, K. Jackson, and J. Dick. *Requirements Engineering, Second Edition.* Springer, 2005.

[67] M. E. C. Hull, K. Jackson, and J. Dick, editors. *Requirements Engineering, Third Edition.* Springer, 2011.

[68] J. Hutchinson, M. Rouncefield, and J. Whittle. Model-driven engineering practices in industry. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 633 –642, may 2011.

[69] L. B.-D. Ian Alexander. *Discovering Requirements: How to Specify Products and Services*. John Wiley & Sons, 2009.

[70] H. Ibrahim, B. Far, and A. Eberlein. Scalability improvement in software evaluation methodologies. In *Information Reuse Integration, 2009. IRI '09. IEEE International Conference on*, pages 236–241, Aug 2009.

[71] IEEE. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, Dec. 1990.

[72] IIBA. *A guide to the Business Analysis Body of Knowledge (BABOK guide), version 2.0.* 2009.

[73] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-oriented software engineering - a use case driven approach.* Addison-Wesley, 1992.

[74] G. Jayaraman, V. P. Ranganath, and J. Hatcliff. Kaveri: delivering the indus java program slicer to eclipse. In *Proceedings of the 8th international conference, held as part of the joint European Conference on Theory and Practice of Software conference on Fundamental Approaches to Software Engineering*, FASE'05, pages 269–272, Berlin, Heidelberg, 2005. Springer-Verlag.

[75] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[76] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. S. Batory, and G. Saake. On the impact of the optional feature problem: analysis and case studies. In D. Muthig and J. D. McGregor, editors, *SPLC*, volume 446 of *ACM International Conference Proceeding Series*, pages 181–190. ACM, 2009.

[77] S. Kausar, S. Tariq, S. Riaz, and A. Khanum. Guidelines for the selection of elicitation techniques. In *Emerging Technologies (ICET), 2010 6th International Conference on*, pages 265 –269, oct. 2010.

[78] D. O. Keck and P. J. Kühn. The feature and service interaction problem in telecommunications systems. a survey. *IEEE Trans. Software Eng.*, 24(10):779–796, 1998.

[79] C. H. P. Kim, C. Kästner, and D. S. Batory. On the modularity of feature interactions. In Y. Smaragdakis and J. G. Siek, editors, *GPCE*, pages 23–34. ACM, 2008.

[80] I.-W. Kim and K.-H. Lee. A model-driven approach for describing semantic web services: From uml to owl-s. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 39(6):637 –646, nov. 2009.

[81] M. Kim, H. Yang, and S. Park. A domain analysis method for software product lines based on scenarios, goals and features. In *Software Engineering Conference, 2003. Tenth Asia-Pacific*, pages 126 – 135, dec. 2003.

[82] K. Kimbler, C. Capellmann, and H. Velthuijsen. Comprehensive approach to service interaction handling. *Comput. Netw. ISDN Syst.*, 30(15):1363–1387, Sept. 1998.

[83] B. Kitchenham, S. L. Pfleeger, L. Pickard, P. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Software Eng.*, 28(8):721–734, 2002.

[84] D. S. Kolovos, R. F. Paige, and F. A. Polack. Model comparison: a foundation for model composition and model transformation testing. In *Proceedings of the 2006 international workshop on Global integrated model management*, GaMMa '06, pages 13–20, New York, NY, USA, 2006. ACM.

[85] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, Oct. 1988.

[86] B. Korel and J. Rilling. Dynamic program slicing in understanding of program execution. In *Program Comprehension, 1997. IWPC '97. Proceedings., Fifth Iternational Workshop on*, pages 80 –89, mar 1997.

[87] P. Kruchten. Architectural blueprints – the "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50, November 1995.

[88] P. A. Laplante. *What Every Engineer Should Know about Software Engineering*. 2007.

[89] D. Leffingwell. Features, Use Cases, Requirements, Oh My!

[90] M. Lehman, J. Ramil, P. Wernick, D. Perry, and W. Turski. Metrics and laws of software evolution-the nineties view. In *International Software Metrics Symposium*, pages 20–32, Albuquerque, NM, Nov. 1997.

[91] M. M. Lehman. On understanding laws, evolution and conservation in the large program life cycle. *Journal of Systems and Software*, 1(3):213–221, 1980.

[92] M. M. Lehman and L. A. Belady. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[93] M. M. Lehman and J. F. Ramil. Software evolution: background, theory, practice. *Information Processing Letters*, 88(1-2):33–44, 2003.

[94] W.-H. F. Leung. Writing reusable feature programs with the feature language extensions. In S. Reiff-Marganiec and M. Ryan, editors, *FIW*, pages 163–177. IOS Press, 2005.

[95] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 112–121, New York, NY, USA, 2006. ACM Press.

[96] N. H. Madhavji, J. Fernandez-Ramil, and D. E. Perry. *Software Evolution and Feedback: Theory and Practice*. Wiley, 2006.

[97] A. Malony, D. Hammerslag, and D. Jablonowski. Traceview: a trace visualization tool. *Software, IEEE*, 8(5):19 –28, sept. 1991.

[98] D. Mancl, S. D. Fraser, and W. F. Opdyke. No silver bullet: a retrospective on the essence and accidents of software engineering. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 758–759, New York, NY, USA, 2007. ACM.

[99] D. McDavid. Systems engineering for the living enterprise. In *Systems Engineering, 2005. ICSEng 2005. 18th International Conference on*, pages 244 – 249, aug. 2005.

[100] S. Mellor and M. Balcer. *Executable UML: A foundation for model-driven architectures*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[101] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Software Eng.*, 28(5):449–462, 2002.

[102] T. Mens and S. Demeyer. *Software Evolution.* Springer, 2008. ISBN 978-3-540-76439-7.

[103] D. Miller. *Business Focused It and Service Excellence.* British Comp Society Series. BCS, 2008.

[104] D. P. Mohapatra, R. Mall, and R. Kumar. An overview of slicing techniques for object-oriented programs. *Informatica (Slovenia)*, 30(2):253–277, 2006.

[105] T. Moreira, M. Wehrmeister, C. Pereira, J.-F. Pétin, and E. Levrat. Generating vhdl source code from uml models of embedded systems. In M. Hinchey, B. Kleinjohann, L. Kleinjohann, P. Lindsay, F. Rammig, J. Timmis, and M. Wolf, editors, *Distributed, Parallel and Biologically Inspired Systems*, volume 329 of *IFIP Advances in Information and Communication Technology*, pages 125–136. Springer Berlin Heidelberg, 2010.

[106] J. Morrison and J. F. George. Exploring the software engineering component in MIS research. *Commun. ACM*, 38(7):80–91, 1995.

[107] M. M. Morshed, M. A. Rahman, and S. U. Ahmed. A literature review of code clone analysis to improve software maintenance process. *CoRR*, abs/1205.5615, 2012.

[108] P. V. Nguyen. The study and approach of software re-engineering. *CoRR*, abs/1112.4016, 2011.

[109] D. Ohst, M. Welle, and U. Kelter. Differences between versions of uml diagrams. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 227–236, New York, NY, USA, 2003. ACM.

[110] C. Pacheco and I. Garcia. A systematic literature review of stakeholder identification methods in requirements elicitation. *Journal of Systems and Software*, 85(9):2171 – 2181, 2012.

[111] F. Paetsch, A. Eberlein, and F. Maurer. Requirements engineering and agile software development. In *IEEE International Workshops on Enabling Technologies: Infrastructure*, pages 308–313, Linz, Austria, 2003.

[112] M. Page-Jones. *Fundamentals of object-oriented design in UML.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[113] M. Palamalai, R. Ahmad, and M. Nizam. Story based mobile application for requirements engineering process. In *Advanced Computer Theory and Engineering, 2008. ICACTE '08. International Conference on*, pages 303 –307, dec. 2008.

[114] D. Parnas. Precise documentation: The key to better software. In S. Nanz, editor, *The Future of Software Engineering*, pages 125–148. Springer Berlin Heidelberg, 2011.

[115] D. Pilone and N. Pitman. *UML 2.0 - in a nutshell: a desktop quick reference.* O'Reilly, 2005.

[116] C. Potts, K. Takahashi, and A. Anton. Inquiry-based requirements analysis. *Software, IEEE*, 11(2):21 –32, march 1994.

[117] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP*, pages 419–443, 1997.

[118] V. Ranganath and J. Hatcliff. An overview of the indus framework for analysis and slicing of concurrent java software (keynote talk - extended abstract).

In *Source Code Analysis and Manipulation, 2006. SCAM '06. Sixth IEEE International Workshop on*, pages 3 –7, sept. 2006.

[119] V. P. Ranganath and J. Hatcliff. Slicing concurrent java programs using indus and kaveri. *STTT*, 9(5-6):489–504, 2007.

[120] J. Reagan. Virtual prototyping: Bridging the Business/IT gap, Second 2008. Copyright - Copyright Data Warehousing Institute Second Quarter 2008; Document feature - Tables; Diagrams; ; Last updated - 2013-08-08; ; Duggan, Jim, Matthew Hotle, Matt Light, and Robert Francis Solon Jr. 2005] . "Will I Save Money Moving From a Waterfall Method to an Iterative One?" Gartner, November 7; Havenstein, Heather 2007]. "Survey: Time, Cost Woes Mar Data Warehousing Projects," Computerworld, March 21; Royce, Winston 1970] . "Managing the Development of Large Software Systems," Proceedings of IEEE WESCON 26: 1-9, August, http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf.

[121] R. Reddy and R. France. Model composition - a signature-based approach. In *in "Aspect Oriented Modeling (AOM) Workshop, Montego*, 2005.

[122] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, 2009.

[123] N. Rozanski and E. Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, 2011.

[124] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14(2):131–164, Apr. 2009.

[125] J. Sadiq and T. Waheed. Reverse engineering amp; design recovery: An evaluation of design recovery techniques. In *Computer Networks and Information Technology (ICCNIT), 2011 International Conference on*, pages 325 –332, july 2011.

[126] G. Salvendy. *Handbook of Human Factors and Ergonomics, 4th Edition*. John Wiley & Sons, 2012.

[127] N. M. Sampat. *Stakeholder Negotiations in Component Based Development*. PhD thesis, De Montfort University, 2004.

[128] A. Shaikh, U. K. Wiil, and N. Memon. Evaluation of tools and slicing techniques for efficient verification of uml/ocl class diagrams. *Adv. Soft. Eng.*, 2011:5:1–5:18, Jan. 2011.

[129] H. Sharp, H. Robinson, and M. Petre. The role of physical artefacts in agile software development: Two complementary perspectives. *Interacting with Computers*, 21(1-2):108–116, 2009.

[130] C. E. Silva. Reverse engineering of gwt applications. In S. D. J. Barbosa, J. C. Campos, R. Kazman, P. A. Palanque, M. D. Harrison, and S. Reeves, editors, *EICS*, pages 325–328. ACM, 2012.

[131] J. Silva. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 44(3):12, 2012.

[132] H. Singh. Software reengineering: New approach to software development. *INTERNATIONAL JOURNAL OF RESEARCH IN EDUCATION METHODOLOGY*, 1(3), 2012.

[133] I. Sommerville. *Software Engineering*. Addison-Wesley, Harlow, England, 9. edition, 2010.

[134] M. Stephan and Cordy. Application of model comparison techniques to model transformation testing. In *MODELSWARD*, 2013.

[135] M. Stephan and J. Cordy. A survey of model comparison approaches and applications. In *Modelsward 2013*, page 10 pp . (to appear), Barcelona, Spain, 2013. 1st International Conference on Model-Driven Engineering and Software Development.

[136] M. Stephan and J. R. Cordy. Application of model comparison techniques to model transformation testing. *MODELSWARD*, 2013.

[137] A. Sutcliffe. Scenario-based requirements engineering. In *Requirements Engineering Conference, 2003. Proceedings. 11th IEEE International*, pages 320 – 329, sept. 2003.

[138] A. G. Sutcliffe and A. Gregoriades. Automating scenario analysis of human and system reliability. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 37(2):249 –261, march 2007.

[139] TheFreeDictionary.com. http://www.thefreedictionary.com.

[140] M. Tian. Scenario-driven requirements engineering : method and tool. 2003.

[141] D. Tucker and D. Simmonds. A case study in software reengineering. In *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pages 1107 –1112, april 2010.

[142] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf. A conceptual basis for feature engineering. *J. Syst. Softw.*, 49(1):3–15, Dec. 1999.

[143] E. Uzuncaova, S. Khurshid, and D. Batory. Incremental test generation for software product lines. *Software Engineering, IEEE Transactions on*, 36(3):309 –322, may-june 2010.

[144] M. van den Brand, Z. Protić, and T. Verhoeff. Generic tool for visualization of model differences. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, IWMCP '10, pages 66–75, New York, NY, USA, 2010. ACM.

[145] K. P. Vangalur S. Alagar. *Specification of Software Systems.* Springer, 2011.

[146] S. Vasilakos, G. Iacobellis, C. Stylios, and M. Fanti. Decision support systems based on a uml description approach. In *Intelligent Systems (IS), 2012 6th IEEE International Conference*, pages 041 –046, sept. 2012.

[147] B. L. Vinz and L. H. Etzkorn. Improving program comprehension by combining code understanding with comment understanding. *Know.-Based Syst.*, 21(8):813–825, Dec. 2008.

[148] M. P. Ward and H. Zedan. Slicing as a program transformation. *ACM Trans. Program. Lang. Syst.*, 29(2), 2007.

[149] M. Weiser. Program slicing. In S. Jeffrey and L. G. Stucki, editors, *ICSE*, pages 439–449. IEEE Computer Society, 1981.

[150] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, July 1982.

[151] M. Weiser. Program slicing. *Software Engineering, IEEE Transactions on*, SE-10(4):352 –357, july 1984.

[152] S. Wenzel, J. Koch, U. Kelter, and A. Kolb. Evolution analysis with animated and 3d-visualizations. In *ICSM*, pages 475–478. IEEE, 2009.

[153] K. E. Wiegers. *Software Requirements.* MICROSOFT PRESS, November 2009.

[154] J. Wu, C. Spitzer, A. Hassan, and R. Holt. Evolution spectrographs: visualizing punctuated change in software evolution. In *Proceedings. 7th International Workshop on Principles of Software Evolution*, 2004.

[155] R. K. Wysocki. *The Business Analyst/Project Manager*. John Wiley & Sons, 2010.

[156] Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 54–65, New York, NY, USA, 2005. ACM.

[157] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, Mar. 2005.

[158] H. Yang, X. Liu, and H. Zedan. Abstraction: a key notion for reverse engineering in a system reengineering approach. *Journal of Software Maintenance*, 12(4):197–228, 2000.

[159] H. Yang and M. Ward. *Successful Evolution Of Software Systems*. Artech House, 2003.

[160] P. Zave. Classification of research efforts in requirements engineering. *ACM Comput. Surv.*, 29(4):315–321, 1997.

[161] Y. Zhang, W. Fu, and H. Leung. Web service publishing and composition based on monadic methods and program slicing. *Knowledge-Based Systems*, 37(0):296 – 304, 2013.

Appendix A

An ATM Case Study Source Code

In this appendix, a JAVA source code of the ATM software system adopted from Deitel with some changes.

This appendix shows the source code of the program after running the INDUS slicer. A forward static slicing technique is used and "*userAuthenticated = false;*" as a slice criteria point. This slice criteria point is in class ATM, method ATM().

```java
// ATM.java
// Represents an automated teller machine
// forward slicing  userAuthenticated

public class ATM
{
    private boolean userAuthenticated; // whether user is authenticated
    private int currentAccountNumber; // current user's account number
    private Screen screen; // ATM's screen
    private Keypad keypad; // ATM's keypad
    private CashDispenser cashDispenser; // ATM's cash dispenser
    private DepositSlot depositSlot; // ATM's deposit slot
    private BankDatabase bankDatabase; // account information database

    // constants corresponding to main menu options
    private static final int BALANCE_INQUIRY = 1;
    private static final int WITHDRAWAL = 2;
    private static final int DEPOSIT = 3;
    private static final int EXIT = 4;

    // no-argument ATM constructor initializes instance variables
    public ATM()
    {
        userAuthenticated = false; // user is not authenticated to start
        currentAccountNumber = 0; // no current account number to start
        screen = new Screen(); // create screen
        keypad = new Keypad(); // create keypad
        cashDispenser = new CashDispenser(); // create cash dispenser
        depositSlot = new DepositSlot(); // create deposit slot
        bankDatabase = new BankDatabase(); // create acct info database
    } // end no-argument ATM constructor

    // start ATM
    public void run()
    {
        // welcome and authenticate user; perform transactions
        while ( true )
        {
            // loop while user is not yet authenticated
            while ( !userAuthenticated )
            {
                screen.displayMessageLine( "\nWelcome!" );
                authenticateUser(); // authenticate user
            } // end while

            performTransactions(); // user is now authenticated
            userAuthenticated = false; // reset before next ATM session
```

```
            currentAccountNumber = 0; // reset before next ATM session
            screen.displayMessageLine( "\nThank you! Goodbye!" );
        } // end while
    } // end method run

    // attempts to authenticate user against database
    private void authenticateUser()
    {
        screen.displayMessage( "\nPlease enter your account number: " );
        int accountNumber = keypad.getInput(); // input account number
        screen.displayMessage( "\nEnter your PIN: " ); // prompt for PIN
        int pin = keypad.getInput(); // input PIN

        // set userAuthenticated to boolean value returned by database
        userAuthenticated =
        bankDatabase.authenticateUser( accountNumber, pin );

        // check whether authentication succeeded
        if ( userAuthenticated )
        {
            currentAccountNumber = accountNumber; // save user's account #
        } // end if
        else
            screen.displayMessageLine(
                        "Invalid account number or PIN. Please try again." );
    } // end method authenticateUser

    // display the main menu and perform transactions
    private void performTransactions()
    {
        // local variable to store transaction currently being processed
        Transaction currentTransaction = null;
        boolean userExited = false; // user has not chosen to exit

        // loop while user has not chosen option to exit system
        while ( !userExited )
        {

            // show main menu and get user selection
            int mainMenuSelection = displayMainMenu();

            // decide how to proceed based on user's menu selection
            switch ( mainMenuSelection )
            {
                // user chose to perform one of three transaction types
                case BALANCE_INQUIRY:
                case WITHDRAWAL:
```

```java
                case DEPOSIT:

                  // initialize as new object of chosen type
                    currentTransaction =
                    createTransaction( mainMenuSelection );

                    currentTransaction.execute(); // execute transaction
                    break;
                case EXIT: // user chose to terminate session
                    screen.displayMessageLine( "\nExiting the system..." );
                    userExited = true; // this ATM session should end
                    break;
                default: // user did not enter an integer from 1-4
                    screen.displayMessageLine(
                        "\nYou did not enter a valid selection. Try again." );
                    break;
            } // end switch
        } // end while
    } // end method performTransactions

    // display the main menu and return an input selection
    private int displayMainMenu()
    {
        screen.displayMessageLine( "\nMain menu:" );
        screen.displayMessageLine( "1 - View my balance" );
        screen.displayMessageLine( "2 - Withdraw cash" );
        screen.displayMessageLine( "3 - Deposit funds" );
        screen.displayMessageLine( "4 - Exit\n" );
        screen.displayMessage( "Enter a choice: " );
        return keypad.getInput(); // return user's selection
    } // end method displayMainMenu

    // return object of specified Transaction subclass
    private Transaction createTransaction( int type )
    {
        Transaction temp = null; // temporary Transaction variable

        // determine which type of Transaction to create
        switch ( type )
        {
            case BALANCE_INQUIRY: // create new BalanceInquiry transaction
                temp = new BalanceInquiry(
                    currentAccountNumber, screen, bankDatabase );
                break;
            case WITHDRAWAL: // create new Withdrawal transaction
                temp = new Withdrawal( currentAccountNumber, screen,
                bankDatabase, keypad, cashDispenser );
```

```
                break;
            case DEPOSIT: // create new Deposit transaction
                temp = new Deposit( currentAccountNumber, screen,
                bankDatabase, keypad, depositSlot );
                break;
        } // end switch

        return temp; // return the newly created object
    } // end method createTransaction
} // end class ATM
```

```java
// Account.java
// Represents a bank account
// forward slicing  userAuthenticated

public class Account
{
    private int accountNumber; // account number
    private int pin; // PIN for authentication
    private double availableBalance; // funds available for withdrawal
    private double totalBalance; // funds available + pending deposits

    // Account constructor initializes attributes
    public Account( int theAccountNumber, int thePIN,
        double theAvailableBalance, double theTotalBalance )
    {
        accountNumber = theAccountNumber;
        pin = thePIN;
        availableBalance = theAvailableBalance;
        totalBalance = theTotalBalance;
    } // end Account constructor

    // determines whether a user-specified PIN matches PIN in Account
    public boolean validatePIN( int userPIN )
    {
        if ( userPIN == pin )
            return true;
        else
            return false;
    } // end method validatePIN

    // returns available balance
    public double getAvailableBalance()
    {
        return availableBalance;
    } // end getAvailableBalance

    // returns the total balance
    public double getTotalBalance()
    {
        return totalBalance;
    } // end method getTotalBalance

    // credits an amount to the account
    public void credit( double amount )
    {
        totalBalance += amount; // add to total balance
    } // end method credit
```

```java
    // debits an amount from the account
    public void debit( double amount )
    {
        availableBalance -= amount; // subtract from available balance
        totalBalance -= amount; // subtract from total balance
    } // end method debit

    // returns account number
    public int getAccountNumber()
    {
        return accountNumber;
    } // end method getAccountNumber
} // end class Account
```

```java
// ATMCaseStudy.java
// Driver program for the ATM case study
// forward slicing  userAuthenticated

public class ATMCaseStudy
{
      // main method creates and runs the ATM
      public static void main( String[] args )
      {
            ATM theATM = new ATM();
            theATM.run();
      } // end main
} // end class ATMCaseStudy
```

```java
// BalanceInquiry.java
// Represents a balance inquiry ATM transaction
// forward slicing  userAuthenticated

public class BalanceInquiry extends Transaction
{
    // BalanceInquiry constructor
    public BalanceInquiry( int userAccountNumber, Screen atmScreen,
    BankDatabase atmBankDatabase )
    {
        super( userAccountNumber, atmScreen, atmBankDatabase );
    } // end BalanceInquiry constructor

 // performs the transaction
    public void execute()
    {
        // get references to bank database and screen
        BankDatabase bankDatabase = getBankDatabase();
        Screen screen = getScreen();

        // get the available balance for the account involved
        double availableBalance =
        bankDatabase.getAvailableBalance( getAccountNumber() );

        // get the total balance for the account involved
        double totalBalance =
        bankDatabase.getTotalBalance( getAccountNumber() );

        // display the balance information on the screen
        screen.displayMessageLine( "\nBalance Information:" );
        screen.displayMessage( " - Available balance: " );
        screen.displayDollarAmount( availableBalance );
        screen.displayMessage( "\n - Total balance: " );
        screen.displayDollarAmount( totalBalance );
        screen.displayMessageLine( "" );
    } // end method execute
} // end class BalanceInquiry
```

```java
// BankDatabase.java
// Represents the bank account information database
// forward slicing  userAuthenticated

public class BankDatabase
{
    private Account accounts[]; // array of Accounts

    // no-argument BankDatabase constructor initializes accounts
    public BankDatabase()
    {
        accounts = new Account[ 2 ]; // just 2 accounts for testing
        accounts[ 0 ] = new Account( 12345, 54321, 1000.0, 1200.0 );
        accounts[ 1 ] = new Account( 98765, 56789, 200.0, 200.0 );

    } // end no-argument BankDatabase constructor

    // retrieve Account object containing specified account number
    private Account getAccount( int accountNumber )
    {
      int i;
      // loop through accounts searching for matching account number
      for (i=0; i < 1;i++)
        {
            // return current account if match found
            if ( accounts[i].getAccountNumber() == accountNumber )
                  return accounts[i];
      } // end fo
      //for ( Account currentAccount : accounts ) do
        /*
         * {
            // return current account if match found
            if ( currentAccount.getAccountNumber() == accountNumber )
                  return currentAccount;
      } // end for
      */

        return null; // if no matching account was found, return null
    } // end method getAccount

    // determine whether user-specified account number and PIN match
    // those of an account in the database
    public boolean authenticateUser( int userAccountNumber, int userPIN )
    {
        // attempt to retrieve the account with the account number
        Account userAccount = getAccount( userAccountNumber );
```

```
         // if account exists, return result of Account method validatePIN
        if ( userAccount != null )
            return userAccount.validatePIN( userPIN );
        else
            return false; // account number not found, so return false
    } // end method authenticateUser

    // return available balance of Account with specified account number
    public double getAvailableBalance( int userAccountNumber )
    {
        return getAccount( userAccountNumber ).getAvailableBalance();
    } // end method getAvailableBalance

    // return total balance of Account with specified account number
    public double getTotalBalance( int userAccountNumber )
    {
        return getAccount( userAccountNumber ).getTotalBalance();
    } // end method getTotalBalance

    // credit an amount to Account with specified account number
    public void credit( int userAccountNumber, double amount )
    {
        getAccount( userAccountNumber ).credit( amount );
    } // end method credit

    // debit an amount from of Account with specified account number
    public void debit( int userAccountNumber, double amount )
    {
        getAccount( userAccountNumber ).debit( amount );
    } // end method debit
} // end class BankDatabase
```

```java
// CashDispenser.java
// Represents the cash dispenser of the ATM
// forward slicing  userAuthenticated

public class CashDispenser
{
    // the default initial number of bills in the cash dispenser
    private final static int INITIAL_COUNT = 500;
    private int count; // number of $20 bills remaining

    // no-argument CashDispenser constructor initializes count to default
    public CashDispenser()
    {
        count = INITIAL_COUNT; // set count attribute to default
    } // end CashDispenser constructor

    // simulates dispensing of specified amount of cash
    public void dispenseCash( int amount )
    {
        int billsRequired = amount / 20; // number of $20 bills required
        count -= billsRequired; // update the count of bills
    } // end method dispenseCash

    // indicates whether cash dispenser can dispense desired amount
    public boolean isSufficientCashAvailable( int amount )
    {
        int billsRequired = amount / 20; // number of $20 bills required

        if ( count >= billsRequired )
            return true; // enough bills available
        else
            return false; // not enough bills available
    } // end method isSufficientCashAvailable
 } // end class CashDispenser
```

```java
// Deposit.java
// Represents a deposit ATM transaction
// forward slicing  userAuthenticated

public class Deposit extends Transaction
{
      private double amount; // amount to deposit
      private Keypad keypad; // reference to keypad
      private DepositSlot depositSlot; // reference to deposit slot
      private final static int CANCELED = 0; // constant for cancel option

      // Deposit constructor
      public Deposit( int userAccountNumber, Screen atmScreen,
                  BankDatabase atmBankDatabase, Keypad atmKeypad,
                  DepositSlot atmDepositSlot )
      {
            // initialize superclass variables
            super( userAccountNumber, atmScreen, atmBankDatabase );

            // initialize references to keypad and deposit slot
            keypad = atmKeypad;
            depositSlot = atmDepositSlot;
      } // end Deposit constructor

      // perform transaction
      public void execute()
      {
            BankDatabase bankDatabase = getBankDatabase(); // get reference
            Screen screen = getScreen(); // get reference

            amount = promptForDepositAmount(); // get deposit amount from user

            // check whether user entered a deposit amount or canceled
            if ( amount != CANCELED )
            {
                  // request deposit envelope containing specified amount
                  screen.displayMessage(
                        "\nPlease insert a deposit envelope containing " );
                  screen.displayDollarAmount( amount );
                  screen.displayMessageLine( "." );

                  // receive deposit envelope
                  boolean envelopeReceived = depositSlot.isEnvelopeReceived();

                  // check whether deposit envelope was received
                  if ( envelopeReceived )
                  {
```

```
                        screen.displayMessageLine( "\nYour envelope has been "
+
                                    "received.\nNOTE: The money just deposited
will not " +
                                    "be available until we verify the amount
of any " +
                              "enclosed cash and your checks clear." );

                        // credit account to reflect the deposit
                        bankDatabase.credit( getAccountNumber(), amount );
                  } // end if
                  else // deposit envelope not received
                  {
                        screen.displayMessageLine( "\nYou did not insert an "
+
                              "envelope, so the ATM has canceled your
transaction." );
                  } // end else
            } // end if
            else // user canceled instead of entering amount
            {
                  screen.displayMessageLine( "\nCanceling transaction..." );
            } // end else
      } // end method execute

      // prompt user to enter a deposit amount in cents
      private double promptForDepositAmount()
      {
            Screen screen = getScreen(); // get reference to screen

            // display the prompt
            screen.displayMessage( "\nPlease enter a deposit amount in " +
                  "CENTS (or 0 to cancel): " );
            int input = keypad.getInput(); // receive input of deposit amount

            // check whether the user canceled or entered a valid amount
            if ( input == CANCELED )
                  return CANCELED;
            else
            {
                  return ( double ) input / 100; // return dollar amount
            } // end else
      } // end method promptForDepositAmount
 } // end class Deposit
```

```java
// Screen.java
// Represents the screen of the ATM
// forward slicing  userAuthenticated

public class Screen
{
    // display a message without a carriage return
    public void displayMessage( String message )
    {
        System.out.print( message );
    } // end method displayMessage

    // display a message with a carriage return
    public void displayMessageLine( String message )
    {
        System.out.println( message );
    } // end method displayMessageLine

    // displays a dollar amount
    public void displayDollarAmount( double amount )
    {
        System.out.print( amount );
    } // end method displayDollarAmount
} // end class Screen
```

```java
// Transaction.java
// Abstract superclass Transaction represents an ATM transaction
// forward slicing  userAuthenticated

public abstract class Transaction
{
    private int accountNumber; // indicates account involved
    private Screen screen; // ATM's screen
    private BankDatabase bankDatabase; // account info database

    // Transaction constructor invoked by subclasses using super()
    public Transaction( int userAccountNumber, Screen atmScreen,
        BankDatabase atmBankDatabase )
    {
        accountNumber = userAccountNumber;
        screen = atmScreen;
        bankDatabase = atmBankDatabase;
    } // end Transaction constructor

        // return account number
    public int getAccountNumber()
    {
        return accountNumber;
    } // end method getAccountNumber

    // return reference to screen
    public Screen getScreen()
    {
        return screen;
    } // end method getScreen

    // return reference to bank database
    public BankDatabase getBankDatabase()
    {
        return bankDatabase;
    } // end method getBankDatabase

    // perform the transaction (overridden by each subclass)
    abstract public void execute();
} // end class Transaction
```

```
// Withdrawal.java
// Represents a withdrawal ATM transaction
// forward slicing  userAuthenticated

public class Withdrawal extends Transaction
{
    private int amount; // amount to withdraw
    private Keypad keypad; // reference to keypad
    private CashDispenser cashDispenser; // reference to cash dispenser

    // constant corresponding to menu option to cancel
    private final static int CANCELED = 6;

    // Withdrawal constructor
    public Withdrawal( int userAccountNumber, Screen atmScreen,
         BankDatabase atmBankDatabase, Keypad atmKeypad,
         CashDispenser atmCashDispenser )
    {
        // initialize superclass variables
        super( userAccountNumber, atmScreen, atmBankDatabase );

        // initialize references to keypad and cash dispenser
        keypad = atmKeypad;
        cashDispenser = atmCashDispenser;
     } // end Withdrawal constructor

// perform transaction
public void execute()
{
    boolean cashDispensed = false; // cash was not dispensed yet
    double availableBalance; // amount available for withdrawal

    // get references to bank database and screen
    BankDatabase bankDatabase = getBankDatabase();
    Screen screen = getScreen();

    // loop until cash is dispensed or the user cancels
    do
    {
        // obtain a chosen withdrawal amount from the user
        amount = displayMenuOfAmounts();

        // check whether user chose a withdrawal amount or canceled
        if ( amount != CANCELED )
        {
            // get available balance of account involved
            availableBalance =
```

```
                bankDatabase.getAvailableBalance( getAccountNumber() );

            // check whether the user has enough money in the account
            if ( amount <= availableBalance )
            {
                // check whether the cash dispenser has enough money
                if ( cashDispenser.isSufficientCashAvailable( amount ) )
                {
                    // update the account involved to reflect the withdrawal
                    bankDatabase.debit( getAccountNumber(), amount );

                    cashDispenser.dispenseCash( amount ); // dispense cash
                    cashDispensed = true; // cash was dispensed

                    // instruct user to take cash
                    screen.displayMessageLine( "\nYour cash has been" +
                        " dispensed. Please take your cash now." );
                } // end if
                else // cash dispenser does not have enough cash
                    screen.displayMessageLine(
                    "\nInsufficient cash available in the ATM." +
                    "\n\nPlease choose a smaller amount." );
             } // end if
             else // not enough money available in user's account
             {
                    screen.displayMessageLine(
                        "\nInsufficient funds in your account." +
                        "\n\nPlease choose a smaller amount." );
             } // end else
         } // end if
        else // user chose cancel menu option
        {
            screen.displayMessageLine( "\nCanceling transaction..." );
            return; // return to main menu because user canceled
        } // end else
    } while ( !cashDispensed );

} // end method execute

    // display a menu of withdrawal amounts and the option to cancel;
    // return the chosen amount or 0 if the user chooses to cancel
    private int displayMenuOfAmounts()
    {
        int userChoice = 0; // local variable to store return value

        Screen screen = getScreen(); // get screen reference
```

```
      // array of amounts to correspond to menu numbers
      int amounts[] = { 0, 20, 40, 60, 100, 200 };

      // loop while no valid choice has been made
      while ( userChoice == 0 )
      {
         // display the menu
         screen.displayMessageLine( "\nWithdrawal Menu:" );
         screen.displayMessageLine( "1 - $20" );
         screen.displayMessageLine( "2 - $40" );
         screen.displayMessageLine( "3 - $60" );
         screen.displayMessageLine( "4 - $100" );
         screen.displayMessageLine( "5 - $200" );
         screen.displayMessageLine( "6 - Cancel transaction" );
         screen.displayMessage( "\nChoose a withdrawal amount: " );

         int input = keypad.getInput(); // get user input through keypad

         // determine how to proceed based on the input value
         switch ( input )
         {
            case 1: // if the user chose a withdrawal amount
            case 2: // (i.e., chose option 1, 2, 3, 4 or 5), return the
            case 3: // corresponding amount from amounts array
            case 4:
            case 5:
               userChoice = amounts[ input ]; // save user's choice
               break;
            case CANCELED: // the user chose to cancel
               userChoice = CANCELED; // save user's choice
               break;
            default: // the user did not enter a value from 1-6
               screen.displayMessageLine(
               "\nInvalid selection. Try again." );
         } // end switch
        } // end while

      return userChoice; // return withdrawal amount or CANCELED
  } // end method displayMenuOfAmounts
} // end class Withdrawal
```