TOWARDS AN INTEGRATED METAMODEL BASED APPROACH TO SOFTWARE REFACTORING

BY
MOHAMMED MISBHAUDDIN

A Dissertation Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

In

COMPUTER SCIENCE AND ENGINEERING

MAY 2012

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This dissertation, written by MOHAMMED MISBHAUDDIN under the direction of his dissertation advisor and approved by his dissertation committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE AND ENGINEERING.

Dissertation Committee

Dissertation Advisor
Dr. Mohammad Alshayeb

Co-Advisor / Member
Dr. Radwan Abdel-Aal

Member
Dr. Moataz Ahmed

Member
Dr. Mahmoud Elish

Member
Dr. Aiman El-Maleh

CCSE College Dean
Dr. Umar Al-Turki

Dean of Graduate Studies
Dr. Salam A. Zummo

2|10|12
Date

عمادة الدراسات العليا
DEANSHIP OF GRADUATE STUDIES

© Mohammed Misbhauddin

2012

DEDICATION

To Mom and Dad,

for their love and sacrifice

ACKNOWLEDGMENTS

Alhamdulillah, a truly long journey of completing a PhD has materialized with me writing this acknowledgement to all who aided and supported me through this ordeal. First of all, I would like to thank my PhD supervisor Dr. Mohammad Alshayeb. I am indebted to him for his patience with me, his judicious guidance and specially his confidence and the freedom he granted me throughout the work. Apart from guidance in life at university and research, Dr. Alshayeb provided a strong sense of encouragement, inspiration and was always ready to lend an ear when I faced difficulties in my personal life. Steve Jobs' aptly quoted "*You cannot connect the dots looking forward; you can only connect them looking backward.*" With the completion of my dissertation, I can't fail to notice the support of Dr. Alshayeb in connecting my past dots.

I would like to thank Prof. Radwan Abdel-Aal, who kindly agreed to co-supervise my dissertation and generously review my work. I would also like to acknowledge the support of my committee member Dr. Moataz for his critiques and discussions on the topics of UML models. I am also grateful to all other members of my PhD committee, Dr. Aiman El-Maleh and Dr. Mahmoud Elish for reviewing and providing constructive feedback on my work.

I gratefully acknowledge the support from the Information and Computer Science Dept., and the King Abdul Aziz City of Science and Technology Graduate Student Grant (No. 18-20 – أ ط).

The time spent in King Fahd University has been great fun due to my friendly colleagues. It has been a great pleasure sharing an office with Yousef Elarian. Although not in my research area, his friendly presence, discussions and useful insights and ideas improved my perception on my work and personal life. I also thank my friends Imran, Abdul Rahmaan, Fareed, Mumtaz, Mujahid, Abdur Rahman, Ammar, Sameh and Walid for all the unforgettable conversations. They vivified my days at KFUPM.

Of course I wish to thank my parents and siblings, specially my Mom who would have been proud of me. My special thanks goes to my beloved wife for her support and patience through difficult times and the good ones. Also to my son, Mohammed Saud, who has given me a lot of joy and his strength has motivated me to excel.

TABLE OF CONTENTS

ACKNOWLEDGMENTS V

TABLE OF CONTENTS VII

LIST OF TABLES....... XIV

LIST OF FIGURES.......XVII

LIST OF ABBREVIATIONS XX

ABSTRACT XXIII

ملخص الرسالة....... XXV

1 CHAPTER 1 INTRODUCTION 1

1.1 Problem Description 4

1.2 Motivation 6

1.3 Research Objectives 8

1.4 Research Methodology 9

1.5 Research Contributions....... 13

1.6 Outline of Dissertation....... 14

2 CHAPTER 2 BACKGROUND 17

2.1 Model Driven Software Engineering....... 17

2.2 UML: Object-Oriented Modeling Language 18

2.2.1 UML Class Diagram....... 23

2.2.2 UML Sequence Diagram 25

2.2.3 UML Use Case Diagram....... 29

2.3 OCL: Modeling Constraints....... 32

2.4 Model Transformation 35

2.4.1 Software Refactoring 39

2.5 Model Transformation Framework 40

2.5.1 Model Transformation System 42

2.5.2 Model Smells 45

2.5.3 Model Behavior 47

2.5.4 Refactoring Quality 48

2.5.5 Refactoring Tool Support 48

2.5.6 Consistency Management 49

3 CHAPTER 3 LITERATURE REVIEW 50

3.1 Code Based Refactoring 50

3.1.1 Bad Smell Identification and Refactoring Suggestion 51

3.1.2 Behavior Preservation 54

3.1.3 Refactoring Application 55

3.1.4 Refactoring Effect Evaluation 57

3.1.5 Consistency Preservation 58

3.2 Model Based Refactoring 58

3.2.1 Model Specification 60

3.2.2 Model Transformation Language 63

3.2.3 Model Smells 65

3.2.4 Model Behavior 70

3.2.5 Model Refactoring 72

3.2.6 Refactoring Quality 74

3.2.7 Tool Support 75

3.2.8 Consistency Management 80

3.3 Refactoring Consistency Management 80

3.4 Metamodel Extension 84

4 CHAPTER 4 INTEGRATED METAMODEL 91

4.1 UML Metamodel 91

4.2 UML Class Diagram 96

4.2.1 UML Class Diagram Metamodel 97

4.2.2 Class Diagram Metamodel Extension 99

4.3 UML Sequence Diagram 99

4.3.1 UML Sequence Diagram Metamodel 100

4.3.2 Sequence Diagram Metamodel Extension 102

4.4 UML Use Case Diagram 108

4.4.1 UML Use Case Diagram Metamodel 108

4.4.2 Use Case Diagram Metamodel Extension 110

4.5 Object Constraint Language (OCL) 142

4.5.1 OCL Metamodel 142

4.5.2 OCL Metamodel Extension 144

4.6 Integrated Metamodel 145

4.6.1 STEP 1: Sequence and Use Case Metamodel Composition 151

4.6.2 STEP 2: Class Metamodel Composition 156

4.6.3 STEP 3: OCL Metamodel Composition 159

5 CHAPTER 5 INTEGRATED MODEL REFACTORING 163

5.1 Model Refactoring Strategy 164

5.1.1 Model Transformation System 164

5.1.2 Model Smell Detection Strategy 166

5.1.3 Model Refactoring Application ... 166

5.1.4 Model Behavior ... 168

5.1.5 Refactoring Process ... 168

5.2 Model Refactoring Template ... 169

5.3 Running Case Study ... 171

5.4 Integrated Model Smells ... 174

5.4.1 Creeping Featurism ... 174

5.4.2 Multiple Personality ... 183

5.4.3 Excessive Alternation ... 196

5.4.4 Undue Familiarity ... 209

5.4.5 Spider's Web ... 220

5.4.6 Specters' ... 232

5.4.7 Model Duplication ... 240

5.4.8 Ripple Effect ... 253

6 CHAPTER 6 TOOL SUPPORT ... 263

6.1 UCDesc: A Use Case Description Tool ... 263

6.1.1 Analysis of Existing Use Case Modeling Tools ... 264

6.1.2 UCDesc Architecture ... 266

6.1.3 Features of UCDesc Tool ... 269

6.1.4 Current Limitations of UCDesc Tool ... 274

6.2 IntegraUML: A multi-view UML Integration and Refactoring Tool ... 275

6.2.1 IntegraUML Architecture ... 275

6.2.2 IntegraUML Input Format ... 278

6.2.3 IntegraUML Features ... 281

6.2.4 Current Limitations of IntegraUML Tool ... 283

7 CHAPTER 7 VALIDATION 285
7.1 Validation Framework 285
7.2 Case Studies 289
7.2.1 Student Projects 290
7.2.2 Published Case Studies 291
7.3 Individual Refactoring 292
7.3.1 OFD (Online Form Designer) 292
7.3.2 OG (OurGoal) 296
7.3.3 ESAP (Electronic Student Academic Portfolio) 299
7.3.4 ME (MyEvents) 303
7.3.5 FOMS (Freelancing Online Management System) 306
7.3.6 ATM (Automated Teller Machine) 309
7.3.7 SCM (Supply Chain Management) 313
7.3.8 O-Comm (OS Commerce) 316
7.3.9 ORA (On-Road Assistance) 320
7.4 Discussion 323
7.4.1 Identification of Model Smells in Use Case Diagrams 323
7.4.2 Identification of Model Smells in Sequence Diagrams 324
7.4.3 Identification of Model Smells in Class Diagrams 325
8 CHAPTER 8 ANALYSIS AND DISCUSSION 327
8.1 Integrated Refactoring 327
8.1.1 OFD (Online Form Designer) 327
8.1.2 OG (OurGoal) 332
8.1.3 ESAP (Electronic Student Academic Portfolio) 336
8.1.4 ME (MyEvents) 339

8.1.5 FOMS (Freelancing Online Management System) 344

8.1.6 ATM (Automated Teller Machine) 347

8.1.7 SCM (Supply Chain Management) 352

8.1.8 O-Comm (OS Commerce) 355

8.1.9 ORA (On-Road Assistance) 360

8.2 Analysis and Discussion 363

8.2.1 Integrated Refactoring Impact on Use Case Diagram 365

8.2.2 Integrated Refactoring Impact on Class Diagram 367

8.2.3 Integrated Refactoring Impact on Sequence Diagram 369

9 CHAPTER 9 CONCLUSION AND FUTURE WORK 371

9.1 Summary 371

9.2 Contributions 373

9.3 Threats to Validity 374

9.4 Future Works 376

APPENDIX 1: FORMAL DESCRIPTION FOR THE UML METAMODEL 379

APPENDIX 2: MODEL REFACTORING CATALOG 400

APPENDIX 3: XML & ASSOCIATED STANDARDS 445

APPENDIX 4: XQUERY FUNCTIONS FOR INTEGRATED MODEL SMELLS 457

APPENDIX 5: XMI SCHEMA FOR EXTENDED USE CASE METAMODEL 474

APPENDIX 6: UCDESC USER MANUAL 480

APPENDIX 7: XMI SCHEMA FOR INTEGRATED METAMODEL 488

APPENDIX 8: INTEGRAUML USER MANUAL 495

APPENDIX 9: UML MODEL METRICS **500**

APPENDIX 10: UML MODEL SMELLS **512**

REFERENCES **526**

VITAE **579**

LIST OF TABLES

Table 1 List of model smells detected using OO metrics 67
Table 2 Template elements from different notation proposed in the literature 112
Table 3 Summary of Alternative Scenarios 122
Table 4 Inclusion and Exclusion Meta-classes in Step 1...153
Table 5 Traceability Matrix for Use Case and Sequence Metamodel Composition 154
Table 6 Traceability Mapping between Class and UC-SD metamodel classes 156
Table 7 Mapping of Syntactic Structure of Sentences into Use Case Objects 267
Table 8 Summary of each student project case study system 291
Table 9 Summary of each published case study system...291
Table 10 Comparison of Class Diagram-level Metrics for OFD System.......................292
Table 11 Comparison of Class Element-level Metrics for OFD System 293
Table 12 Comparison of Use Case Diagram-level Metrics for OFD System.................294
Table 13 Comparison of Use Case Element-level Metrics for OFD System 294
Table 14 Comparison of Sequence Element-level Metrics for OFD System.................295
Table 15 Comparison of Class Diagram-level Metrics for OG System.........................296
Table 16 Comparison of Class Element-level Metrics for OG System.........................297
Table 17 Comparison of Use Case Diagram-level Metrics for OG System...................298
Table 18 Comparison of Use Case Element-level Metrics for OG System...................298
Table 19 Comparison of Sequence Element-level Metrics for OG System 299
Table 20 Comparison of Class Diagram-level Metrics for ESAP System 299
Table 21 Comparison of Class Element-level Metrics for ESAP System......................300
Table 22 Comparison of Use Case Diagram-level Metrics for ESAP System 301
Table 23 Comparison of Use Case Element-level Metrics for ESAP System................301
Table 24 Comparison of Sequence Element-level Metrics for ESAP System 302
Table 25 Comparison of Class Diagram-level Metrics for ME System.........................303
Table 26 Comparison of Class Element-level Metrics for ME System 304
Table 27 Comparison of Use Case Diagram-level Metrics for ME System...................305
Table 28 Comparison of Use Case Element-level Metrics for ME System 305
Table 29 Comparison of Sequence Element-level Metrics for ME System...................306
Table 30 Comparison of Class Diagram-level Metrics for FOMS System 306
Table 31 Comparison of Class Element-level Metrics for FOMS System.....................307
Table 32 Comparison of Use Case Diagram-level Metrics for FOMS System..............308
Table 33 Comparison of Use Case Element-level Metrics for FOMS System...............308
Table 34 Comparison of Sequence Element-level Metrics for FOMS System 309
Table 35 Comparison of Class Diagram-level Metrics for ATM System......................309
Table 36 Comparison of Class Element-level Metrics for ATM System.......................310
Table 37 Comparison of Use Case Diagram-level Metrics for ATM System................311
Table 38 Comparison of Use Case Element-level Metrics for ATM System.................311
Table 39 Comparison of Sequence Element-level Metrics for ATM System 312

Table 40 Comparison of Class Diagram-level Metrics for SCM System 313
Table 41 Comparison of Class Element-level Metrics for SCM System 314
Table 42 Comparison of Use Case Diagram-level Metrics for SCM System 315
Table 43 Comparison of Use Case Element-level Metrics for SCM System 315
Table 44 Comparison of Sequence Element-level Metrics for SCM System.................316
Table 45 Comparison of Class Diagram-level Metrics for O-Comm System 316
Table 46 Comparison of Class Element-level Metrics for O-Comm System.................317
Table 47 Comparison of Use Case Diagram-level Metrics for O-Comm System 318
Table 48 Comparison of Use Case Element-level Metrics for O-Comm System...........319
Table 49 Comparison of Sequence Element-level Metrics for O-Comm System 319
Table 50 Comparison of Class Diagram-level Metrics for ORA System 320
Table 51 Comparison of Class Element-level Metrics for ORA System 321
Table 52 Comparison of Use Case Diagram-level Metrics for ORA System 322
Table 53 Comparison of Use Case Element-level Metrics for ORA System 322
Table 54 Comparison of Sequence Element-level Metrics for ORA System.................323
Table 55 Comparison of Class Diagram-level Metrics for OFD System.......................328
Table 56 Comparison of Class Element-level Metrics for OFD System 329
Table 57 Comparison of Use Case Diagram-level Metrics for OFD System.................330
Table 58 Comparison of Use Case Element-level Metrics for OFD System 330
Table 59 Comparison of Sequence Element-level Metrics for OFD System 331
Table 60 Comparison of Class Diagram-level Metrics for OG System 332
Table 61 Comparison of Class Element-level Metrics for OG System..........................333
Table 62 Comparison of Use Case Diagram-level Metrics for OG System 334
Table 63 Comparison of Use Case Element-level Metrics for OG System....................334
Table 64 Comparison of Sequence Element-level Metrics for OG System 335
Table 65 Comparison of Class Diagram-level Metrics for ESAP System 336
Table 66 Comparison of Class Element-level Metrics for ESAP System......................337
Table 67 Comparison of Use Case Diagram-level Metrics for ESAP System 338
Table 68 Comparison of Use Case Element-level Metrics for ESAP System................338
Table 69 Comparison of Sequence Element-level Metrics for ESAP System 339
Table 70 Comparison of Class Diagram-level Metrics for ME System.........................340
Table 71 Comparison of Class Element-level Metrics for ME System 341
Table 72 Comparison of Use Case Diagram-level Metrics for ME System...................342
Table 73 Comparison of Use Case Element-level Metrics for ME System 342
Table 74 Comparison of Sequence Element-level Metrics for ME System 343
Table 75 Comparison of Class Diagram-level Metrics for FOMS System 344
Table 76 Comparison of Class Element-level Metrics for FOMS System.....................345
Table 77 Comparison of Use Case Diagram-level Metrics for FOMS System 346
Table 78 Comparison of Use Case Element-level Metrics for FOMS System...............346
Table 79 Comparison of Sequence Element-level Metrics for FOMS System 347

Table 80 Comparison of Class Diagram-level Metrics for ATM System......................348
Table 81 Comparison of Class Element-level Metrics for ATM System.......................349
Table 82 Comparison of Use Case Diagram-level Metrics for ATM System................350
Table 83 Comparison of Use Case Element-level Metrics for ATM System.................351
Table 84 Comparison of Sequence Element-level Metrics for ATM System 351
Table 85 Comparison of Class Diagram-level Metrics for SCM System 352
Table 86 Comparison of Class Element-level Metrics for SCM System.......................353
Table 87 Comparison of Use Case Diagram-level Metrics for SCM System 354
Table 88 Comparison of Use Case Element-level Metrics for SCM System.................354
Table 89 Comparison of Sequence Element-level Metrics for SCM System.................355
Table 90 Comparison of Class Diagram-level Metrics for O-Comm System................356
Table 91 Comparison of Class Element-level Metrics for O-Comm System.................357
Table 92 Comparison of Use Case Diagram-level Metrics for O-Comm System..........358
Table 93 Comparison of Use Case Element-level Metrics for O-Comm System...........358
Table 94 Comparison of Sequence Element-level Metrics for O-Comm System 359
Table 95 Comparison of Class Diagram-level Metrics for ORA System 360
Table 96 Comparison of Class Element-level Metrics for ORA System.......................361
Table 97 Comparison of Use Case Diagram-level Metrics for ORA System 362
Table 98 Comparison of Use Case Element-level Metrics for ORA System.................362
Table 99 Comparison of Sequence Element-level Metrics for ORA System.................363
Table 100 Refactoring impact spectrum over use case design size metrics 366
Table 101 Refactoring impact spectrum over use case complexity metrics...................367
Table 102 Refactoring impact spectrum over class metrics 368
Table 103 Refactoring impact spectrum over sequence model design size metrics 369
Table 104 Refactoring impact spectrum over sequence model message frequency 370

LIST OF FIGURES

Figure 1 Schematic Representation of the Proposed Research Approach 12
Figure 2 Hierarchical Classification of UML Diagrams 20
Figure 3 Four Layer Architecture for Metamodel Management 22
Figure 4 Graphical notations for UML Class Diagram 24
Figure 5 Graphical notations for UML Sequence Diagram 29
Figure 6 Graphical notations for UML Use case diagram 32
Figure 7 Outline of an OCL Constraint Specification 33
Figure 8 Taxonomy of Model Transformation 36
Figure 9 Model refactoring example 40
Figure 10 Relationship between models and graph representation 61
Figure 11 UML Profile Metamodel 95
Figure 12 Classification of UML Diagrams into Views 96
Figure 13 Subset of the UML Class Diagram Metamodel 98
Figure 14 Subset of the UML Sequence Diagram Metamodel 101
Figure 15 An example lightweight extension of "alt" fragment 105
Figure 16 Extended Component of the Sequence Metamodel 106
Figure 17 Extended Sequence Diagram Metamodel [368] 107
Figure 18 Subset of the UML Use case diagram metamodel 109
Figure 19 Use Case Behavior Description Approaches 110
Figure 20 Addition to the extended UML metamodel for Actor 117
Figure 21 Addition to the extended UML metamodel for Use Case 118
Figure 22 Addition to the extended UML metamodel for extend relationship 124
Figure 23 Structure of a typical text based use case description 126
Figure 24 Excerpt of the Extended Metamodel for the Use Case Flow of Events 128
Figure 25 Excerpt of the Extended Metamodel for the Use Case Flow Steps 129
Figure 26 UC Description example depicting the use of Alternative Flow 131
Figure 27 UC Description example depicting the use of Extension Points 132
Figure 28 Metamodel for the Anchor meta-class mentioned in Figure 25 132
Figure 29 Excerpt of the Extended Metamodel for UC Flow with Generalization 135
Figure 30 UC Flow Generalization example 135
Figure 31 Multiple Use Case Scenarios adapted from [1] 137
Figure 32 Excerpt of the Extended Metamodel for Constraint 138
Figure 33 The Complete Extended Use case diagram Metamodel 140
Figure 34 OCL Metamodel 143
Figure 35 Excerpt of the Extended OCL Metamodel 144
Figure 36 UML Model Integration Elements 147
Figure 37 Model Integration Framework 150
Figure 38 Abstract Relationship between Use Case and Sequence Diagram 152
Figure 39 Step 1: The UC-SD (Intermediate) Metamodel 155

Figure 40 Step 2: The View (Intermediate) Metamodel 158
Figure 41 Traceability Mapping between UC Constraint and OCL Metamodel 159
Figure 42 The Complete Integrated Metamodel 162
Figure 43 Use Case Diagram of the Running Case Study 172
Figure 44 Remove Functional Decomposition Refactoring 179
Figure 45 Excerpt of the NBS model views depicting Creeping Featurism Smell 181
Figure 46 Excerpt of the NBS model views after refactoring 182
Figure 47 Middle Man Lifeline Pattern within a Sequence Model 186
Figure 48 Decompose God Use Case Refactoring 189
Figure 49 Excerpt of the NBS model views depicting Multiple Personality Smell 191
Figure 50 Excerpt of the NBS model view depicting Multiple Personality Smell 192
Figure 51 Excerpt of the NBS model views after refactoring 193
Figure 52 Excerpt of the NBS model view after refactoring 194
Figure 53 Use Case Behavior (Sequence Model) divided into three sections 199
Figure 54 Substitute Excessive Extensions Refactoring 201
Figure 55 Excerpt of the NBS model views depicting Excessive Alternation Smell 203
Figure 56 Excerpt of the NBS model views depicting Excessive Alternation Smell 204
Figure 57 Excerpt of the NBS model views after refactoring 205
Figure 58 Excerpt of the NBS model views after refactoring 206
Figure 59 Break Intitmate Elements Refactoring 214
Figure 60 Excerpt of the NBS model views depicting Undue Familiarity Smell 216
Figure 61 Excerpt of the NBS model views after refactoring 217
Figure 62 Sample use case model depicting Spider's Web Model Smell 220
Figure 63 Redistribute Responsibility Refactoring 226
Figure 64 Excerpt of the NBS model views depicting Spider’s Web Smell 227
Figure 65 Excerpt of the NBS model views after refactoring 228
Figure 66 Remove Specters' Refactoring 236
Figure 67 Excerpt of the NBS model views depicting Specters’ Smell 237
Figure 68 Excerpt of the NBS model views after refactoring 238
Figure 69 Concepts of Paths in the detection strategy for Duplication Model Smell 242
Figure 70 Remove Duplication Refactoring 248
Figure 71 Excerpt of the NBS model views depicting Duplication Smell 250
Figure 72 Excerpt of the NBS model views after refactoring 251
Figure 73 Class Responsibility Assignment Refactoring 258
Figure 74 Excerpt of the NBS model views depicting Ripple Effect Smell 259
Figure 75 Excerpt of the NBS model views after refactoring 260
Figure 76 Sample XMI excerpt exported by CaseComplete UML Tool 265
Figure 77 High-Level Architecture of UCDesc Tool 266
Figure 78 Example yUML Link and corresponding Use Case Diagram 268
Figure 79 UCDesc Main Layout 269

Figure 80 UCDesc Use Case Description Format 271
Figure 81 UCDesc (a) Use Case Description and (b) Flow Authoring Windows 272
Figure 82 An example use case flow description and its equivalent XMI 274
Figure 83 High-Level Architecture of the IntegraUML tool 277
Figure 84 Platform-specific Architecture of IntegraUML 278
Figure 85 XML Schema Diagram of the UML Class Diagram 280
Figure 86 XML Schema Diagram of the UML Sequence Diagram 281
Figure 87 Use Case Diagram for IntegraUML 282
Figure 88 IntegraUML Main Layout 283
Figure 89 Validation Framework 287
Figure 90 Number of middle-man using design patterns used in case studies 325
Figure 91 Number of instances of model smells detected over UML Class Diagrams .. 326
Figure 92 Number of instances of Integrated Model Smells detected 364
Figure 93 Use case metrics association with model characteristics 365
Figure 94 Sequence model metrics association with model characteristics 369

LIST OF ABBREVIATIONS

AGG	:	Attributed Graph Grammar
API	:	Application Programming Interface
ATL	:	ATLAS Transformation Language
CASE	:	Computer Aided Software Engineering
CSP	:	Constraint Satisfaction Problem
DOM	:	Document Object Model
DPO	:	Double Push-Out Scheme
DTD	:	Document Type Definition
ECL	:	Embedded Constraint Language
FSL	:	Formal Specification Language
GME	:	Generic Modeling Environment
GRASP	:	General Responsibility Assignment Software Pattern
GTS	:	Graphical Transformation Language
HTML	:	Hyper Text Markup Language
IDE	:	Integrated Development Environment
MDA	:	Model Driven Architecture
MDSE	:	Model Driven Software Engineering

MET	:	Model Element Term
MOF	:	Meta-Object Facility
MTL	:	Model Transformation Language
MTS	:	Model Transformation System
NAC	:	Negative Application Condition
OCL	:	Object Constraint Language
OMG	:	Object Management Group
OO	:	Object Oriented
PIM	:	Platform Independent Model
PSM	:	Platform Specific Model
SDO	:	Simple Delegating Operation
SGML	:	Standardized Generalized Markup Language
SPO	:	Simple Push-Out Scheme
TGG	:	Triple Graph Grammar
QVT	:	Query/View/Transformation
UC	:	Use Case
UML	:	Unified Modeling Language

UMLAUT	:	UML All pUprposes Transformer
W3C	:	World Wide Web Consortium
XML	:	eXtensible Modeling Language
XMI	:	XML Metadata Interchange
XSLT	:	eXtensible Sylesheet Language Transformation

ABSTRACT

Full Name : Mohammed Misbhauddin

Thesis Title : Towards An Integrated Metamodel Based Approach to Software Refactoring

Major Field : Computer Science and Engineering

Date of Degree : May 2012

Software refactoring is the process of changing a software system in a manner that does not alter its external behavior and yet improving its internal structure. Model-Driven Architecture and the popularity of the UML have enabled the application of refactoring at model-level which earlier was applied to only software code. Refactoring at model level is more multifaceted and challenging than at source code level. Hence, research in this area is still considered to be in its infancy. The objective of this research was to develop a multi-view integrated approach to model-driven refactoring using UML models. The main motivation behind using multiple views for model refactoring was to utilize the inter-view relationships to bridge the gap between code and model refactoring. In this research, a single model from each UML view is composed at metamodel level to construct an integrated metamodel. Class diagram representing the structural view, sequence diagram representing the behavioral view and use case diagram representing the functional view were selected for integration. A total of eight integrated refactoring opportunities that can be used to improve the design models were proposed over the integrated metamodel along with a set of primitive refactorings that can be used to remove the proposed smells. A prototype tool called IntegraUML that performs model integration and refactoring was also developed to allow semi-automated identification and resolution of the model smells. Validation of the proposed approach was performed by comparing integrated refactoring approach with refactoring applied to models individually in terms of quality improvement through UML model metrics. A total of nine case studies were considered for empirical validation of the proposed approach. It is concluded that more opportunities can be detected using the integrated approach rather than the individual refactoring approach. Apart from this, there was a significant

improvement in the design size, complexity and modularity of the individual models after the application of refactoring over the integrated model as opposed to individual refactoring. Future work to this approach can investigate on using other models in the integration, application of pattern refactoring over the integrated metamodel and empirical validation over large real-world project designs.

ملخص الرسالة

الاسم الكامل : محمد مصباح الدين

عنوان الرسالة : نحو نظام تعريف نموذجي متكامل لاعادة هيكلية البرمجيات

التخصص : علوم الحاسب الآلي والهندسة

تاريخ الدرجة العلمية : مايو ٢٠١٢

إعادة هيكلية البرمجيات هي عملية تغيير نظام البرمجيات بحيث تحسن من هيكله الداخلي ولا تغير سلوكه الخارجي. مكنت الهيكلية المرتبطة بالنماذج ولغة النمذجة الموحدة (UML) تطبيق إعادة الهيكلية على مستوى النماذج والتي كانت في السابق تطبق على شيفرة البرمجيات. إعادة الهيكلية على مستوى النماذج هو متعدد الأوجه وأكثر صعوبة من على مستوى شيفرة المصدة. لهذا، لا تزال الابحاث في هذا المجال تعتبر في المراحل الأولى. الهدف من هذا البحث هو وضع نهج متكامل متعدد الاوجه لإعادة الهيكلية النماذج باستخدام لغة النمذجة الموحدة (UML). الدافع الرئيسي لاستخدام طرق المتعددة لإعادة هيكلية النماذج هو للاستفادة من العلاقات المتداخلة بهدف سد الفجوة بين شيفرة ونماذج إعادة هيكلية البرمجيات. في هذا البحث، تم استخدام نموذج واحد من أوجه لغة النمذجة الموحدة (UML) على مستوى النموذج العام لبناء نموذج عام متكامل .تم تحديد نموذج الاصناف ليمثل وجهة النظر البنيوية، نموذج مخطط التسلسل ليمثل وجهة النظر السلوكية ونموذج حالات الاستخدام ليمثل وجهة النظر الوظيفية. تم اقتراح مجموعه من ثمانية فرص إعادة الهيكلة التي يمكن استخدامها لتحسين النماذج المقترحة على النموذج العام المتكامل بالإضافة الى مجموعة من طرق اعادة الهيكلة البدائية التي يمكن استخدامها لإزالة المشاكل المقترحة. تم تصميم أداة تسمى IntegraUML والتي تنفذ تكامل النماذج وإعادة الهيكلية وكذلك تسمح بالتحديد شبه الالي وتصليح مشاكل النماذج. تم إجراء المصادقة على النهج المقترح وذلك من خلال مقارنة نهج إعادة الهيكلية المتكامل مع تطبيق إعادة الهيكلية على نماذج فردية من حيث تحسين الجودة من خلال قياس متريات لغة النمذجة الموحدة (UML). تم تطبيق تسع حالات دراسة للتحقق من صحة النهج المقترح. كانت النتيجة أنه يمكن الكشف عن فرص لإعادة الهيكلية أكثر باستخدام النهج المتكامل مقارنة بنهج اعادة الهيكلة الفردي. وفضلا عن ذلك، كان هناك تحسن كبير في حجم التصميم، والتعقيد والنمطية للنماذج الفردية بعد تطبيق إعادة الهيكلية من خلال النموذج المتكامل بدلا من إعادة هيكلية النماذج الفردية. العمل المستقبلي سيبحث في استخدام نماذج أخرى في النموذج المتكامل، تطبيق اعادة هيكلية الانماط من خلال النموذج العام المتكامل والمصادقة باستخدام تصاميم مشروع حقيقي كبير.

CHAPTER 1

INTRODUCTION

Software Maintenance has become an integral component of software development and management. The process of maintaining software requires application of certain set of activities that modify an existing software system. A number of incentives dictate the need for maintaining software, which includes factors such as failures in performance and implementation, changes in information and environment, inefficiencies in operation etc. Chapin et al. [2] categorized software maintenance based on the objective evidence of the maintainer's activities. Another term usually associated synonymously with software maintenance is *Software Evolution* [3].

Cook et al. [4] defined the term *Evolvability* as the "*capability of software products to be evolved to continue to serve its customers in a cost effective manner*". Hence, software evolution is a subset of software maintenance activities that occur when perfective (add, remove or modify functionality), corrective (remove errors) or performance (improve operation and quality) maintenance for the customer's benefit is executed. Over the years, software practitioners and managers have been struggling to get hold over the software development process in order to cope up with the rate of change and minimize its effects on delivering better software products. Hence, software maintenance and evolution not only incorporate activities after the delivery of the system but also during its development phase.

An interesting feature of evolution is its structural aspect. Portion of a system is considered weak or instable if its structure (code, design, architecture) hinders its evolution. Structural weakness, if not identified and removed or at least improved, will spread throughout the system causing more weaknesses and result in a system difficult to maintain. In order to meet this challenge, studies have identified and developed software engineering methodologies to improve the structure of software systems leading to improvement in overall software quality. Although activities that target removing or improving structural weaknesses are categorized under maintenance, Chikofsky and Cross [5] referred to them as *Restructuring*. According to Chikofsky and Cross, "*Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior*". William F. Opdyke as an outcome of his PhD dissertation [6] redefined restructuring in terms of Object Oriented Development domain as "*behavior preserving program transformations*" and termed this paradigm as *Refactoring*.

The main objective of refactoring is simplicity – keeping the system as simple as possible. Refactoring improves the internal design of the software and is considered an essential activity during software development and maintenance. It provides developers with the ability to understand the software better, to modify and maintain and as a result account for a significant portion of the development effort.

Motivated from the work of Opdyke, a large portion of the methodologies and tools related to refactoring that operate at source code level are proposed in the literature. These tools aid the developer with identification of code blocks in need of refactoring and a few provide automated refactoring support by selecting the most appropriate way to

restructure. With the growing popularity of Model-driven software engineering (MDSE), refactoring has moved in recent times from the more generic code-based refactoring to a higher level of abstraction. MDSE is a discipline that promotes the use of models at different levels of abstractions for developing, maintaining and evolving software systems [7]. Software researchers are now concentrating their efforts on refactoring software design models. Some of the motivations for moving from code to design models for refactoring are [8]:

- A model provides an abstract view of the system; hence, visualization of the structural changes required is easier.
- Problems uncovered at the design-level can be improved directly on the model.
- Exploring alternate decision paths is much cheaper at the design-level.

Although there exist numerous terms related to MDSE such as Model-Driven Development (MDD), Model Driven Software Development (MDSD) and Model-Driven Architecture (MDA), they do not imply the same methodology. MDA [9] tends to be more restrictive and focuses on UML-based modeling languages. Due to its widespread use for modeling Object Oriented Systems, UML (Unified Modeling Language) [10] models are used as suitable candidates for model-driven refactoring in recent literature.

1.1 Problem Description

Although model-driven refactoring has attained wide recognition and acceptance, several vexing problems remain. Research in this area is still under development bounded by a number of challenges and open issues. Some of the key challenges and issues, highlighted by Mens et al. [11-14] when applying refactoring to software models and based on our systematic review of the literature of the field, are summarized here:

- **Lack of model refactoring opportunities**: In order to apply refactoring to models, identification of structural weaknesses and design defects within the model (also known as Model Smells or Refactoring Opportunities) is required. There exists quite a few refactoring opportunities when it comes to code refactoring [15-17]. In contrast, only a few refactoring opportunities have been discussed in terms of model-driven refactoring. One of the main reasons behind this research-gap is because models are typically built up from different views composed of multiple diagrams as opposed to source code that conforms to a single model (based on the language used). Of all the studies that relate to model-driven refactoring in the literature, only 54% of them address the concept of model smells and their detection strategies [18]. There is a need to identify a comprehensive and commonly accepted list of model refactoring opportunities. Apart from this, there is also a need to establish relation between the refactoring opportunity (problem) and appropriate refactoring operations that can improve the model by removing the problem.
- **Lack of precise definition of proposed refactoring opportunities:** Studies that discuss the same refactoring opportunity use different identification strategies. This

inconsistency is mainly because of the multi-view nature of UML modeling. For instance, the God Class refactoring opportunity is described by both the class and sequence diagram in some studies [19, 20] while it is described by the class diagram only in others [21-25]. Apart from this, studies that use the same diagram use different threshold values to quantify the opportunity. For instance, Ghannem et al. [22] classify a class as God Class if it has more than 10 attributes and 20 methods whereas Llano and Pooley [23] classify it if a class is composed of 60 or more attributes and methods.

- **Lack of precise definition of behavior in models:** By definition, software refactoring is a contemporary software maintenance activity intended to modify the internal structure of the software without changing its observable behavior. In order to ensure this, a precise definition of behavior is required for models to achieve true model-driven refactoring. Apart from this, there is also need of a formal specification technique to state the behavioral invariants and methods to verify whether model refactoring preserves these invariants. A key research challenge is therefore the lack of precise definition of behavior and formalisms to define and verify behavior preservation for model-driven refactoring.
- **Lack of an evaluation framework:** Another important objective of refactoring is improvement in software quality because of restructuring the software model. Although an important activity, only 5.3% of the studies published on model-driven refactoring address it [18]. Lack of an evaluation approach severely affects the usability of model-driven refactoring approaches in industrial software development.

- **Inconsistency among different models:** Due to the multi-view nature of software models, the issue of consistency and synchronization is important. UML is a collection of different diagrams representing different views. Although different, most of these diagrams contain complementary information. Applying refactoring to one of these diagrams could result in inconsistencies among other dependent models. This issue is contrasting to code based refactoring which is often (but not always) expressed within a single programming language. The key issue here is to identify an approach to ensure model consistency between all dependent views.
- **Lack of automated tool support:** One of the main requirements in the area of refactoring is the availability of tool support to automatically detect and remove model defects. Tool support provided in the field of model-driven refactoring are usually classified based on the degree of automation provided. A fully automated tool provides automatic detection and correction of defects without human intervention. A semi-automated tool requires human assistance before the actual transformation. More than two-thirds of the studies that provide model-driven refactoring tools are not fully automated [18]. The two main factors affecting full automation are 1) no means of automated model defect detection and 2) tools that do provide detection do not provide a mapping between the defects and refactoring solutions.

1.2 Motivation

Refactoring at model-level is more multifaceted and challenging than at code-level due to the existence of multiple views. A typical software design is composed of diagrams from

all views, each capturing an important characteristic of the system. A view is a collection of diagrams that illustrate similar aspects of the system. With the growing popularity of MDA and UML based techniques, researchers have started exploring the use of multiple views for model analysis. Some prominent applications include Model Consistency Management [26-28], Model Evaluation [29] and Model Reuse [30]. Most research studies published on model-driven refactoring concentrate mainly on refactoring application on individual models from a view at a time. Model-driven refactoring approaches can be classified based on two criteria: the number of views considered for refactoring and the technique used [31]. The main motivations behind the use of multiple views for refactoring are:

- There exists a complementary relation among all the UML views. Refactoring a single diagram from a view at a time ignores the surplus information available from inter-view relationships. A few recent approaches have suggested the use of multiple views for model-driven refactoring [19, 20, 31, 32]. Although effective, these approaches either do not consider all available model views or incorporate views outside the scope of UML modeling notation.
- One of the motivations for model-driven refactoring is that the problems uncovered at the design-level can be improved directly on the model. However, since the set of refactoring opportunities for program refactoring are more detailed than the model based refactoring, a large number of smells escape and seep into the implemented code. Considering multiple views for refactoring opportunity detection provides a broad view of all aspects required for a complete description of the system.

- The use of multiple views allows integrating behavioral information into other static views. This integration allows refactoring operations to assess model behavior and ensure its preservation post refactoring.
- Refactoring an integrated multi-view model applies the refactoring operation to all related model views hence circumventing or considerably reducing the effort needed to ensure model consistency.

1.3 Research Objectives

Although the concept of refactoring is being researched thoroughly, its application to UML models is still faced with numerous issues and challenges. Most of these issues are due to the multi-view nature of modeling in UML. The main objective of this work is to fill the gap between the source-code and model-driven refactoring by applying refactoring to more than one view at a time. Our work addresses the following research questions:

1. What is the state-of-the-art in UML model-driven refactoring?
2. How can multiple UML views be used to identify refactoring opportunities?
3. Which refactoring opportunities can be re-used and adapted to model-driven refactoring because of the multi-view integrated model?
4. How to specify model refactoring steps over multi-view integrated metamodel and prove they preserve the observable behavior of the complete system?
5. How to automate the process of model integration and the process of applying model refactoring in the form of a tool?

Specifically, the objectives of this research are

1. To provide an integrated metamodel that combines the metamodel of class model, sequence model and use case model representing the three views of UML.
2. To identify refactoring opportunities within the software design using model information from multiple views. This information is obtained from the integrated model.
3. To provide refactoring solutions to mitigate the identified opportunities at the metamodel level. The use of metamodel for refactoring provides the user with additional information regarding the semantics in the model and the structure that the model is required to follow. This information aids in describing the refactoring steps.
4. To provide automated tool support for model integration and refactoring. Automated support ensures proper model conversion so they conform to the specified metamodel accordingly.

1.4 Research Methodology

In order to address the issues identified in Section 1.3, we propose the use of multiple views for model-driven refactoring. UML 2 defines 14 different diagrams as part of its most recent specification [10]. Since the use of all diagrams in included in the UML suite for refactoring is not feasible, we use the concept of views. Typically UML models are classified into three views: structural, behavioral and functional [33]. Each view represents an important aspect of the system and together they provide a complete description of the system. Although these views are independent from each other, there

exists a relationship (information dependency) among diagrams in these views. We need to establish convergence points where the integration of all the views is possible i.e. a way to represent in the structural view, the behavior of each element found from the behavioral view and the functionality found in the functional view. This integration of different UML diagrams can supplement additional meaning to the entire system thereby increasing the information available as a whole. For instance, adding behavior information available from behavioral view of UML such as sequence diagrams, state diagrams *etc*. to the structural view such as class diagrams.

With feasibility of the approach in mind, we selected a single diagram from each view based on its popularity and functional use. Core diagrams used in our approach include Class diagram from the structural view, Sequence diagram from the behavioral view and Use case diagram from the functional view. An outline of our research approach is depicted in Figure 1. The key components of our methodology include

1. **Metamodel Extension:** In order to ensure proper integration of metamodels, we extended the metamodels of sequence diagram and use case diagram. The class diagram metamodel was used as-is from the UML specification. The use case diagram metamodel was extended with behavior information in order to establish its relation to the sequence model. The sequence diagram metamodel was extended to handle model traceability and act as a liaison between the use case metamodel and class diagram metamodel. The main motivation behind these extensions is to ensure seamless integration of all selected metamodels to form the Integrated metamodel. Section 4.2 to 4.5 discusses our extensions to the UML metamodels in detail.

2. **Metamodel Integration:** The Integrated metamodel is composed of metamodels of the class diagram, the extended metamodel of the sequence diagram and the extended metamodel of the use case diagram. In order to ensure complete modeling of information, we also incorporated the Object Constraint Language (OCL) [34] metamodel within the Integrated metamodel so that constraints (from class diagrams), invariants and guards (from sequence diagrams) and pre and post conditions (from use case diagrams) are structurally represented. Section 4.6 discusses the composition of metamodels and the complete Integrated Metamodel.
3. **Integrated Model Refactoring:** An important aspect of model-driven refactoring is to identify refactoring opportunities and suggest refactoring operations. We identify and propose refactoring opportunities over the Integrated Metamodel and provide refactoring steps to remove these defects. We define a template in order to present our proposed refactoring opportunities and steps. The two main components of this template are the smell detection algorithm and model refactoring steps that consists of composite refactoring to remove the detected model smell. Chapter 5 discusses our proposed refactoring opportunities along with examples from a running case study.

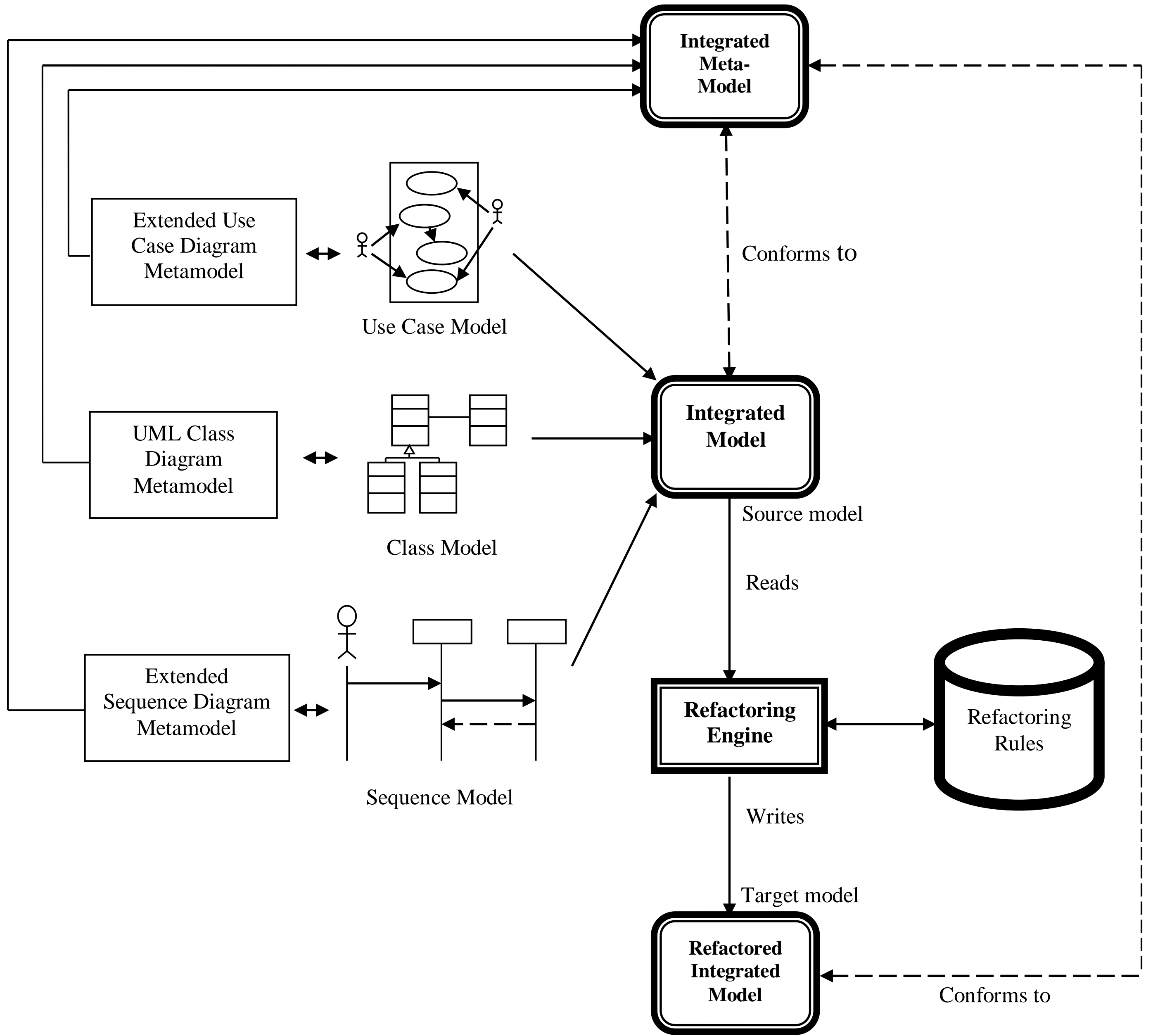


Figure 1 Schematic Representation of the Proposed Research Approach

4. **Tool Support:** An important aspect of proposing a metamodel driven refactoring approach is to provide tool support for automatic model conversion for metamodel

conformance. We implemented a tool called *IntegraUML* to aid users in converting their models to the proposed metamodel. The main objective of this tool is to support model integration and transformation on UML models serialized in the form of XMI (XML Metadata Interchange) [35]. The tool imports the XMI representations of the class diagram, sequence diagrams and use case diagrams and integrates them into an intermediate format that conforms to the proposed Integrated metamodel. The smell detection module is an XQuery (Query language for XML) based engine that imports model smell descriptions from the Refactoring Rules repository one by one and applies it over the integrated model. Each model smell within the repository is stored as an XQuery file. The refactoring module applies the appropriate refactoring to the integrated model. This process is repeated until all smells in the repository are exhausted. Chapter 6 provides a detail description on the architecture of all the prototype tools implemented as part of our work.

5. **Validation:** Due to the lack of an evaluation framework in the literature that associates external model quality to internal attributes, we used the model metrics as part of our validation framework. In order to validate our approach; we compared model metrics evaluated over refactoring individual model views versus refactoring multiple model views. Case studies used for validation and analysis and results of the validation are discussed in Chapter 7 and Chapter 8 respectively.

1.5 Research Contributions

The major contributions of the work proposed in this dissertation are as follows:

1. A state-of-the-art literature review of software refactoring and UML model-driven refactoring approaches.
2. An integrated metamodel that unifies the three different views of the UML language (Structural, Behavioral and Functional).
3. An initial catalogue of model-driven refactoring opportunities based on individual UML models and the Integrated metamodel.
4. A formal description of UML model syntax and semantics and their use in describing model constraints to ensure model behavior preservation.
5. A prototype tool that enables model integration and refactoring based on the proposed integrated metamodel. It also facilitates the refactoring process and allows verification of preconditions and automatic application of refactoring rules.
6. Providing refactorings for structural, behavioral and functional view of UML together along with an XML-based formalism to represent transformation using a new integrated metamodel.

1.6 Outline of Dissertation

The rest of this dissertation is structured as follows:

- Chapter 2 presents the background knowledge upon which the work presented in this dissertation is based upon. The chapter describes the Model-Driven Software Engineering paradigm and Object-Oriented Modeling notations such as the Unified Modeling Language (UML) and the Object Constraint Language (OCL). Next, the chapter also introduces the concept of Model Transformation and Software

Refactoring. Finally, the chapter describes model metrics, used for validating the proposed approach.

- Chapter 3 surveys state-of-the-art in the field of software refactoring. The first section reviews refactoring studies conducted at code-level also known as *Program Refactoring*. The second section reviews refactoring research conducted at model-level also known as *Model-driven Refactoring*. The third section reviews the approaches carried out in the research literature to synchronize refactorings between design artifacts and code also known as *Source-Consistent Refactoring*. Since the work presented in this literature involves proposing extensions to existing UML diagram metamodels, the fourth section briefly reviews all research efforts made in the area of metamodel extensions.
- Chapter 4 initially describes UML metamodels for the diagrams considered for the work proposed in this dissertation: Class Diagram, Sequence Diagram and Use case Diagram. The chapter then explains in detail the proposed extensions to the metamodels of these UML models. Finally, the chapter concludes with the integrated metamodel process and description.
- Chapter 5 where the main contribution of the dissertation resides, describes the detection strategies for refactoring opportunity identification and implementation of refactoring over the integrated metamodel in detail. This chapter introduces a template to describe the integrated refactorings proposed.
- Chapter 6 describes the implementation of the model integration and refactoring tool. This chapter introduces the Integration subsystem, the Refactoring subsystem and describes data structures and storage mechanisms based on the Integrated metamodel.

- Chapter 7 illustrates the methodology used for validating the integrated model refactorings. Next, the chapter describes the suite of case studies used for the validation process. The chapter also provides the data collection methodology and the information used for validating the approach. Finally, the chapter presents the results of refactoring application over individual UML models.
- Chapter 8 presents the results of the refactoring application over the integrated UML models. Finally, a thorough discussion based on the analysis of the results is included.
- Chapter 9 concludes the dissertation by answering the research questions posed. It presents the contributions, threats to validity and future work.

CHAPTER 2

BACKGROUND

This chapter provides background over some of the key concepts used in this work. These key concepts include explanations of notations and techniques used throughout the rest of this dissertation.

2.1 Model Driven Software Engineering

The use of models for software development has been around since a long time. Although used in the software development process, models were treated as informal sketches or used for "*mere*" documentation purposes [36]. Prior to the formulation of the Model Driven Software Engineering (MDSE) concept, models were considered informal drafts of the software under development. These models were discarded once the code was completed. With advent of model driven approaches, models are treated as key artifacts in all phases of the software development lifecycle.

Model-driven software engineering (MDSE) is becoming the most promising paradigm in software engineering. MDSE is a discipline that promotes the use of models at different levels of abstraction for developing, maintaining and evolving software systems [7]. It varies from the traditional software development paradigm by shifting focus to system models that capture system requirements, architecture and design decisions that fulfill them. In addition, these system models can be used to partially or fully automate code

generation in any target language. MDSE provides an environment that ensures the systematic and disciplined use of models throughout the software development process. Hence, it ensures an audit trace starting right from system requirements through the code that implements them.

Although there exist numerous terms related to MDSE such as Model-Driven Development (MDD), Model Driven Software Development (MDSD) and Model-Driven Architecture (MDA), they do not imply the same methodology. MDA [9] tends to be more restrictive and focuses on UML-based modeling languages. Out of the many approaches to MDSE, MDA adopted by the Object Management Group (OMG) has become the most favorable one. The three primary objectives of MDA are portability, interoperability and reusability.

Unified Modeling Language [10], although not originally designed for MDA, became a standard formalism for a wide range of application domains due to its wide use and popularity. UML describes various types of models in MDA. It contains diagrams and views that can represent various perspectives of a system.

2.2 UML: Object-Oriented Modeling Language

The Object Oriented paradigm has achieved immense popularity over other programming paradigms. The prime reason for this acceptance is that it gives priority to modeling concepts, which is important from the problem domain’s perspective, leaving behind programming technicalities to be filled in later. With the growing popularity of the Object

Oriented paradigm and with an intention to provide a standard for Object Oriented Analysis and Design, the Object Management Group (OMG) adopted UML as a standard language for the design and analysis of Object Oriented Programs.

UML is a graphical language that provides notations and action semantics to describe and design software systems. It was a result of amalgamation of different graphical modeling approaches by Grady Booch [37], James Rumbaugh *et al.* [38] and Ivar Jacobson [39]. Not only does UML describe a software system at different levels of abstraction, but is also used in tools for software simulation [40].

Since the adoption of UML as an open standard by OMG in 1997, it has undergone constant evolution to keep up with criticisms [41] in order to provide a more precise and expressive modeling language. The most recent specification, UML 2.4, describes 14 formal diagrams, which intend to provide different views of a system under design. A view is a collection of diagrams that illustrate similar characteristics of the system. The UML taxonomy classifies its diagrams into two views: structural and behavioral. There have been other proposals for view classifications such as the 4+1 view by Kruchten [42] and the structural, behavioral and functional view classification proposed by Iivari [33]. Since the UML taxonomy provides no categorization for representing the functional aspects of the UML modeling suite, we decided to adopt the view classification by Iivari. A typical classification of the diagrams into three different views: structural, behavioral and functional is shown in Figure 2. Use case diagrams are a means of specifying functionality according to Jacobson et al. [39]. The classification of Activity Diagram into the functional view is based on the observations by Rumbaugh et al. [38] and Shlaer

and Mellor [43, 44] who use data flow modeling concepts such as action/activity to describe the functionality of the system.

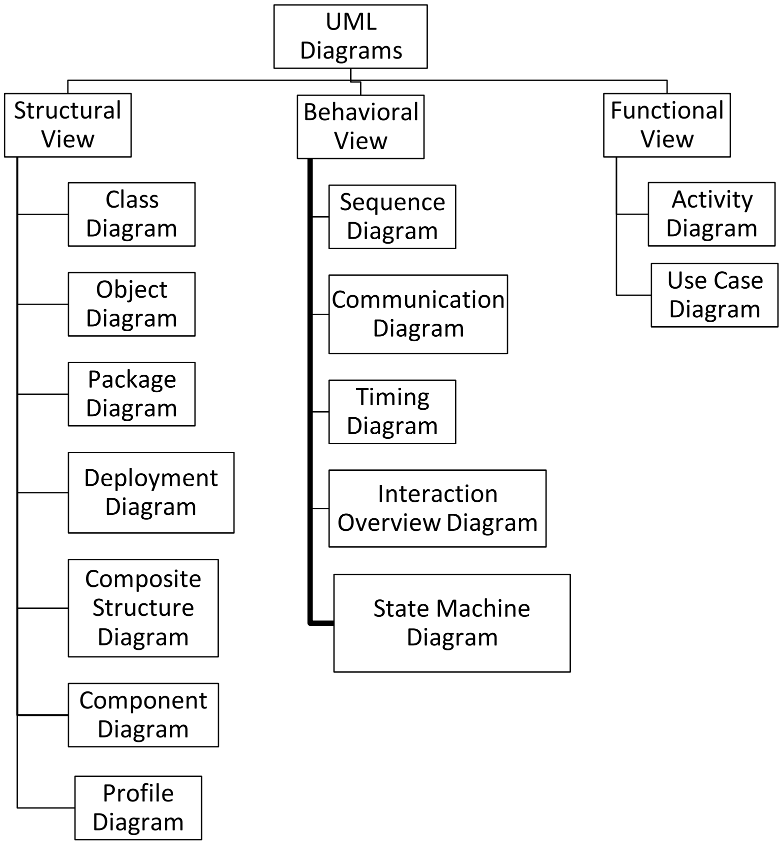


Figure 2 Hierarchical Classification of UML Diagrams

Structural View

The basic building blocks in an object-oriented design are objects and classes. The structural view provides diagrams that capture the physical organization of these blocks in the system. It describes a static structure of the system. One of the most prominent diagrams in this view is the class diagram, which is considered

part of the integrated metamodel proposed in this dissertation. Section 2.2.1 provides a detail description of the class diagram.

Behavioral View

Diagrams included in the behavioral view show the dynamic behavior of the structural objects in the system. This dynamic behavior specifies the series of changes made to the system over time. One of the most commonly used diagrams to model system behavior is the sequence diagram, which is considered part of the integrated metamodel proposed in this dissertation. Section 2.2.2 provides a detail description of the sequence diagram.

Functional View

The functional view is a collection of diagrams that depict how a system is supposed to work. It captures information about the system from the user's perspective. Because of these advantages, these diagrams are among a few which are constructed early in the development of software. One of the most vital diagrams from this view that provides modeling of system's functional requirements is the Use Case Diagram. The use case diagram is considered as part of the integrated metamodel proposed in this dissertation. Section 2.2.3 provides a detail description of the use case diagram.

All UML diagrams conform to the UML metamodel that specifies its abstract syntax, concrete syntax and semantics. A metamodel is a model of the modeling language (such as UML). A notation known as Meta-Object Facility (MOF) [45] put forward by OMG allows software engineers to build and extend UML metamodels. In order to demonstrate

the relationship between the systems under development, models and metamodels, a four-layer architecture provided by MOF is shown in Figure 3.

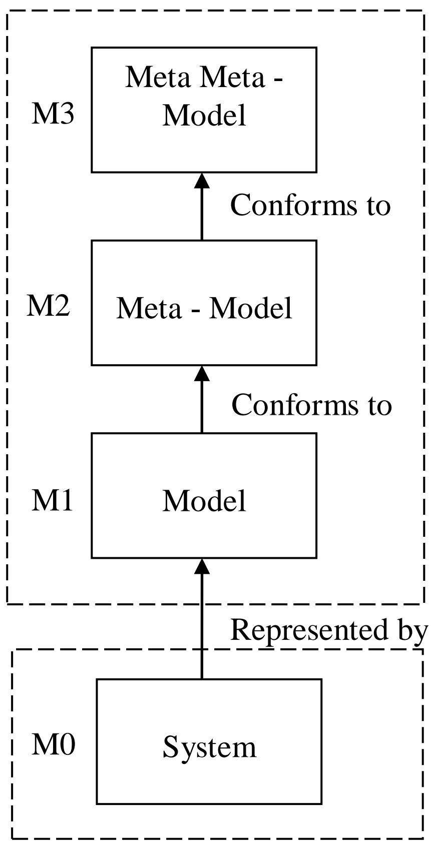


Figure 3 Four Layer Architecture for Metamodel Management

Based on this architecture, metamodels for modeling languages can be defined using MOF. These modeling languages, like UML, can then be used to describe domain specific concepts. Finally user data can be instantiated. A complete system can be developed using UML by generating a number of UML models. Models that represent the system are then used to produce a software system that conforms to the model.

2.2.1 UML Class Diagram

Class diagram represents the structural view of an object-oriented system. It consists of a set of classes designating important entities of the modeled system. Along with classes, a class diagram also consists of relationships between these classes. It is the most common diagram and considered as the backbone for modeling object-oriented systems.

Classes are defined as a set of objects sharing the same attributes and methods. Attributes are unique features of a class and methods are the means through which a class exposes its functionality to other classes. A class is typically represented in UML as a rectangle with three partitions. The top partition identifies its name, the middle partition lists all its attributes and the bottom partition lists all its methods. Associated with each attribute and method of a class is an important concept called *visibility*. Visibility specifies whether other objects are allowed to see the corresponding attribute or method of a given class. UML defines three kinds of visibility:

- Public (+) which allows access to objects of all other classes
- Private (-) which allows access to objects of the owner class only
- Protected (#) which allows access to objects of its subclasses

Classes in a class diagram are related to each other by different types of relationships. Relationships in a UML class diagram are classified into three categories: Association, Generalization and Dependency. When two or more classes are connected to each other, an association relationship exists between them. Aggregation is a type of association between classes when a class (whole) is formed by a collection of other classes (parts). Composition is a stronger form of aggregation in which the lifetime of the part classes is

dependent on the lifetime of the whole class. Generalization is a relationship between a super class and a subclass. Also termed as inheritance, the child class (subclass) inherits common functionality defined in the parent class (super class). Dependency is a directed relationship from a target class to a source class in which the target class requires the presence and information of the source class. It is not possible to give a precise semantics to the dependency relation as it is decided by the manner in which the users use it. This is why semantic description of the dependency relation will not be included. All the graphical notations available in a UML class diagram are given in Figure 4.

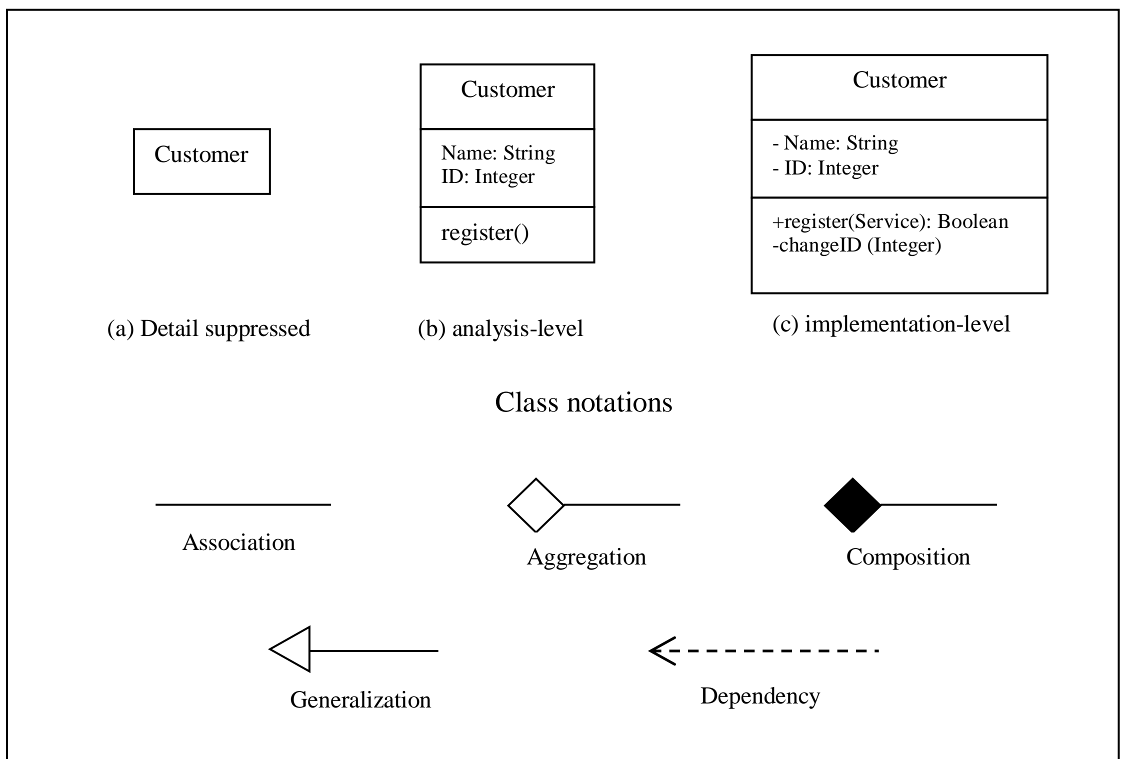


Figure 4 Graphical notations for UML Class Diagram

A class diagram can be used to provide both a conceptual design (referred to as Domain Model) as well as detailed design (referred to as Design Model) of the system under

development. It is because of this flexibility, class diagrams are primarily used to comprehend requirements and domain-level entities. A domain model mainly consists of classes and relationships between them. The classes in this model are usually detail-suppressed (Figure 4(a)) or analysis-level (Figure 4(b)) with few attributes and no methods. Association is the primary relationship used in the domain model. However, other relationships such as generalization can also be depicted. A design model is structurally similar to a domain model but more detailed. These details include visibility and type of attributes and methods, navigations and new associations discovered as part of the detail analysis.

2.2.2 UML Sequence Diagram

Sequence diagram represents the dynamic view of an object-oriented system. The main purpose of a sequence diagram is to capture dynamic behavior of a system. This is realized by modeling flow of events leading to a desired result. Vital information made available reading a sequence diagram are the messages that are sent between objects as well as the order in which they occur. This information is conveyed along the horizontal and vertical dimensions of the diagram. Moving through the vertical dimension from left to right, we can identify the objects between which the messages are exchanged and moving along the horizontal dimension from top to bottom provides the time sequence of these messages. Objects on a sequence diagram are depicted as a "*lifeline*" which includes a dotted line along the vertical axis, which extends for the period of the interaction. Messages are shown with arrows moving from the sending object to the receiving object (except for gates, which are discussed later). Different messages are depicted by different styles of arrows. Each message contains two events: a send event

occurring at the sender's end of the message and a receive event occurring at the receiver' end of the message.

In a sequence diagram, the natural order in which messages are exchanged is sequential from top to bottom. This concept of sequential ordering was broadened with the inclusion of "*Combined Fragments*". Combined Fragment is a notation element added to the UML 2 specification to allow grouping of messages together in order to depict conditional flow in a sequence diagram. Prior to this in UML 1.x, "*in-line*" guards were used which soon became incapable of handling sophisticated logic required for complex sequences.

A combined fragment is composed of two elements: an operand and a guard. An operand can be thought of as a sub-sequence diagram that constitutes the body of the combined fragment. A combined fragment can have one or more operands depending upon its type. A guard is associated with each operand, which is a Boolean condition that needs to evaluate to "true" in order to execute the sequence within the operand. The guard is positioned on the top-left corner of the operand. The UML 2 specification identified twelve kinds of combined fragments. These fragment kinds are discussed below.

- *Alt (alternatives)* is used to represent choice of behavior. It has multiple guarded operands chosen in a mutually exclusive manner based on the outcome of the guard expression. In programming terms, it realizes the "*if – else*" logic.
- *Opt (optional)* is used to represent sole choice of behavior. It has a single guarded operand executed based on the outcome of the guard expression. In programming terms, it realizes the "*if*" logic.

- *Break* is used to represent a breaking scenario. It has a single guarded operand executed instead of the remainder of the enclosing fragment or the diagram. It usually models the exception handling behavior.
- *Par (parallel)* is used to represent concurrent merge between the operands. It has multiple operands that execute in parallel without compromising the integrity of the outcome.
- *Seq (Sequencing)* is used to represent weak sequencing between the operands. It has multiple operands that enforce the execution of messages within a preceding operand before the next one starts. However, it does not impose any order within an operand on messages not sharing a lifeline.
- *Strict* is used to represent strict sequencing between operands. It is similar to *seq* with the exception that messages within an operand must follow the ordered sequence.
- *Neg (negative)* is used to represent traces designated as invalid. It has a single operand showing a sequence that should not be possible and not allowed. All other sequences are considered positive.
- *Assert* is used to represent an assertion. It designates that any sequence of messages not shown as an operand of the assertion are not valid.
- *Critical* is used to designate a sequence of messages as critical.
- *Loop* is used to represent a repeating sequence. It has a single guarded operand repeated a number of times based on the outcome of the guard. A loop guard specifies the minimum and the maximum number of iterations.

- *Consider / Ignore* is a combined single operand fragment. The "*consider*" operand identifies messages that should be considered within the combined fragment. Alternatively, the "*ignore*" operand defines the messages that should be ignored.

With the release of UML 2, the "Interaction Use" element was also introduced. Interaction Use provides the designer with the ability to merge simpler sequence diagrams to form complex sequence diagrams. In other words, it represents an abstract sequence diagram component. An interaction use element is depicted similar to a combined fragment with the keyword "ref" placed on the top-left corner. The operand of this frame contains the name of the referenced sequence diagram along with any parameters. Information is passed from and to the main sequence diagram through parameters and return values respectively. Another way of passing information from the main diagram to a referenced fragment is by using gates. As opposed to the discussion earlier that messages are depicted as arrows between lifelines, gates are messages with one end connected to a frame's edge and the other connected to a lifeline. The UML specification defines three types of gates: formal gates if it belongs to the main sequence diagram, actual gates on interaction use element and fragment gates on combined fragments.

Another significant improvement made in UML 2 was the concept of *part-decomposition*. Part-decomposition allows a lifeline in a sequence diagram to be complex element in itself. The internal interactions of this lifeline can be shown as a separate sequence diagram. Messages to or from the decomposed lifeline are treated as gates. Corresponding gates on the sequence diagram explaining the decomposition match these gates. A Sequence diagram also allows the placement of a constraint over a lifeline

known as *State Invariant*. This constraint must evaluate to true for the remainder of the trace to be valid. A state invariant is depicted on a sequence diagram by placing the constraint inside curly braces on the lifeline. All the graphical notations available in a UML sequence diagram are shown in Figure 5.

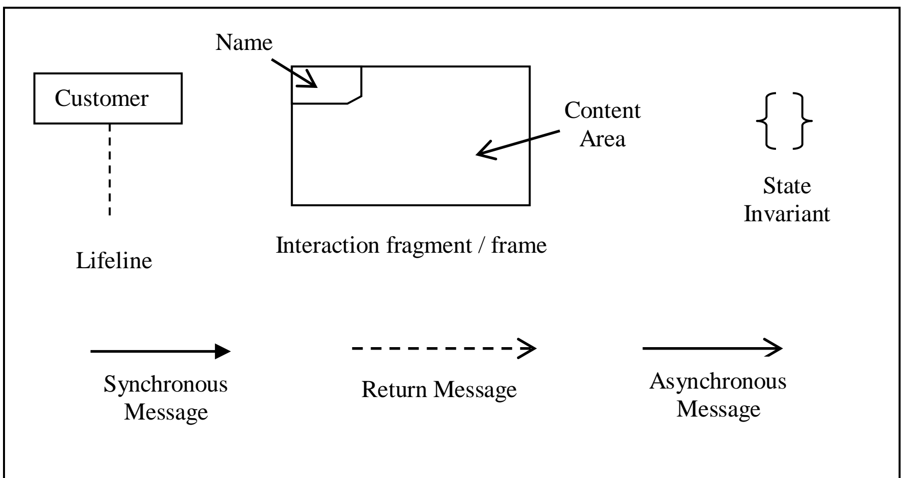


Figure 5 Graphical notations for UML Sequence Diagram

2.2.3 UML Use Case Diagram

Jacobson et al. [39] initially introduced the concept of use case diagrams that was later adopted by OMG to be part of the Unified Modeling Language. Use case diagrams represent a functional view of the object-oriented system. This diagram plays a vital role in modeling the system requirements. Requirements are represented as a set of use cases within the use case diagram. Each use case is a specification of a set of operations between the system and actors resulting in an output valuable to actors or stakeholders of the system.

A use case diagram consists of four distinct elements that depict the working of a system: The system itself, the actors that interact with the system, the services (or use cases) the

system is required to perform and the relationships between these elements. The system element sets the boundary of the system with respect to the actors who use it and the services it must provide. Actors are depicted outside the system element boundary as they are not realized by the system and services are depicted inside the system element. The notion of a system element is to establish the scope of the system.

An actor element is either a person or another system that is involved in the successful operation of the system. Relationships in a use case diagram can be classified into three broad categories:

1. Actor - Use Case Relationship
2. Actor - Actor Relationship
3. Use case - Use case Relationship

An actor in a use case diagram can be associated to one or more use cases. This relationship can specify whether the actor initiates the use case or receives results from the use case or both. An actor-use case relationship is also known as association. Although not explicitly mentioned in the UML Specification, UML provides one actor-actor relationship called generalization. Since this relationship also applies to use-cases, it will be referred here as actor generalization. Adapted from the similar concept of class diagrams, actor generalization allows different actors with common functionality to be represented by a general actor. This general actor can then be related to specialized actors that are identified by unique needs.

UML allows three different relationships between use cases: generalization, Inclusion and Extension. Use case generalization is similar in definition to actor generalization

where general functionality is separated from specific functionality in different use cases. Specific use cases inherit general functionality and add their own specific different functionality to the specification. Two use cases are related by inclusion if one use case uses the functionality offered by the other use case. The use case that includes the other use case is typically not complete on its own. This relationship induces the concept of reusability in a use case diagram. An inclusion relationship is represented by a directed arrow from the including use case to the included use case with a keyword <<*include*>> over the arrow. An extension relationship exists between two use cases when one use case wants to utilize the functionality of another use case if certain conditions are satisfied. In contrast to the inclusion relationship, a use case that extends the other use case is complete on its own. The extending use case is also known as the base use case. The base use case should have a clearly defined extension point where the extension use case can be invoked for additional functionality. An extension relationship is also represented by a directed arrow from the extension use case to the base use case with a keyword <<*extend*>> over the arrow. The base use case has a partition with the keyword <<*extension point*>> that identifies the point of an extension use-case invocation.

All the graphical notations available in a UML use case diagram are shown in Figure 6.

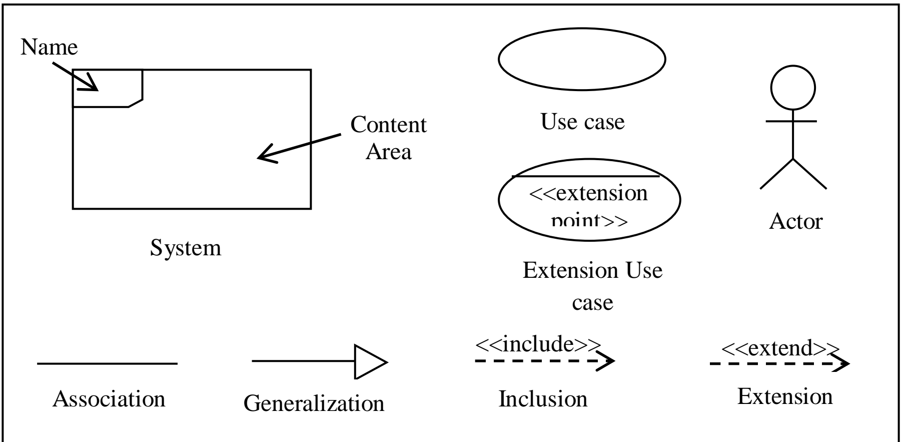


Figure 6 Graphical notations for UML Use case diagram

2.3 OCL: Modeling Constraints

The notation provided by UML can only express information that can be represented graphically. In order to express properties such as constraints, invariants etc. on UML models that cannot be represented graphically, a formal text-based declarative language is required. Object Constraint language (OCL) [34] is a declarative specification language adopted by OMG as part of the UML 2.0 specification. OCL provides the ability to access model elements and express constraints over these elements using invariants, pre-conditions and post-conditions. OCL is a declarative language that cannot change the value of a model element and hence considered side effect free. UML models annotated with OCL constraints add preciseness and well formedness to the models and assists in its verification and validation.

An OCL constraint typically consists of two parts: the context and a set of OCL expressions. As an OCL constraint highly depends upon which model element is constrained, this context of the OCL constraint specifies this information. An OCL

context can either be a classifier, attribute of a classifier or an operation. Outline of an OCL constraint is given in Figure 7 below.

```
Context <context-Name > : <model-Element>

        <expression-Type><expression-Name> :<expression-Body>

        <expression-Type><expression-Name> :<expression-Body>

        ...
```

Figure 7 Outline of an OCL Constraint Specification

The context (model-Element) of a constraint can be referenced in its body either by a keyword "*self*" or by assigning it an optional name (*context-Name*). The body of a constraint consists of a set of OCL expressions. Each expression consists of a type, name and body. Some of the mostly used expressions types include: inv, pre, post, body, init and derive.

- *inv (invariant)* is a static constraint that specifies conditions that must evaluate to true at any given moment. It is typically used when the body contains a condition that must be met by all instances of a classifier.
- *Pre (pre-condition)* specifies the conditions that must evaluate to true before execution of an operation starts.
- *Post (post-condition)* specifies the condition that must evaluate to true after the execution is completed.
- *init (initial)* specifies an initial value of an attribute.
- *derive* specifies how a value for an attribute can be obtained.

An expression can access the property or operation of the classifier in context. Since OCL is a query language, it expects a result when querying the property or operation of a context. This result can either be single-valued or multi-valued. OCL uses the "." operator when it expects a single value and uses the "->" operator when it expects a multi-valued result. Multi-valued results in OCL are known as collections and are of three different types: Sets, Bags and Sequences. A set cannot contain duplicate items, a bag can contain duplicate items and sequences are similar to bags but the elements are ordered.

Boolean operators (and, or, xor, not and implies) are used to combine multiple expressions in an expression body. A few popular expression forms that can be included in the expression body are:

- *Literal Expression* ($LiteralExp$) specifies an expression with no arguments and produces a result.
- *If Expression* ($IfExp$) specifies a Boolean condition. Based on the outcome of this condition, two other expressions specified by the "*then Expression*" and the "*else Expression*" are executed.
- *Loop Expression* ($LoopExp$) specifies a loop construct over a collection. An iterator represents each element in the collection during iterations of the loop.
- *Variable Expression* ($VarExp$) specifies reference to a variable.
- *Message Expression* ($MessageExp$) returns a collection of OCL Messages.
- *State Expression* ($StateExp$) specifies the state of a class within an expression.
- *Type Expression* ($TypeExp$) specifies an existing meta type in an expression

- *Feature Call Expression* ($FeatureCallExp$) specifies a feature defined for a classifier in the UML model like property, operation etc.

2.4 Model Transformation

One of the main reasons why models are considered as second-class development assets is that they do not raise productivity to a sufficient level. With the advent of MDSE, it ensured that models could be formally and precisely defined and hence can be used as primary artifacts in the process of software development. One of the main components that enable MDSE and accounts for the key success of model-based approaches is Model transformation. Model Transformation is considered one of the integral activities that ensure that models can be used for software evolution, refinement and realization in code. It is considered the heart and soul of model-driven architecture [46].

Model Transformation is an approach that takes as input a source model that conforms to a given source metamodel and produces another model conforming to a given target metamodel as output. A number of model transformation approaches have emerged recently in lieu of OMG's initiative for a MDSE approach. The MDSE approach provides opportunity for Platform Independent Model (PIM) to PIM and PIM to Platform Specific Model (PSM) transformations. Numerous classifications for model transformation approaches have been proposed in the literature [46-50]. An exemplary list of model transformations includes:

- Generating lower-level models from higher-level models (e.g. code from design models).
- Synchronizing models at the same level (vertical consistency) or different levels (horizontal consistency) of abstraction
- Model evolution tasks (e.g. Model refactoring)
- Reverse engineering of higher-level models from lower-level models (e.g. design models from code)

Although these classifications are thoroughly detailed, we present a simpler taxonomy of model transformation approaches to comprehend the scope of our work. This taxonomy is shown in Figure 8.

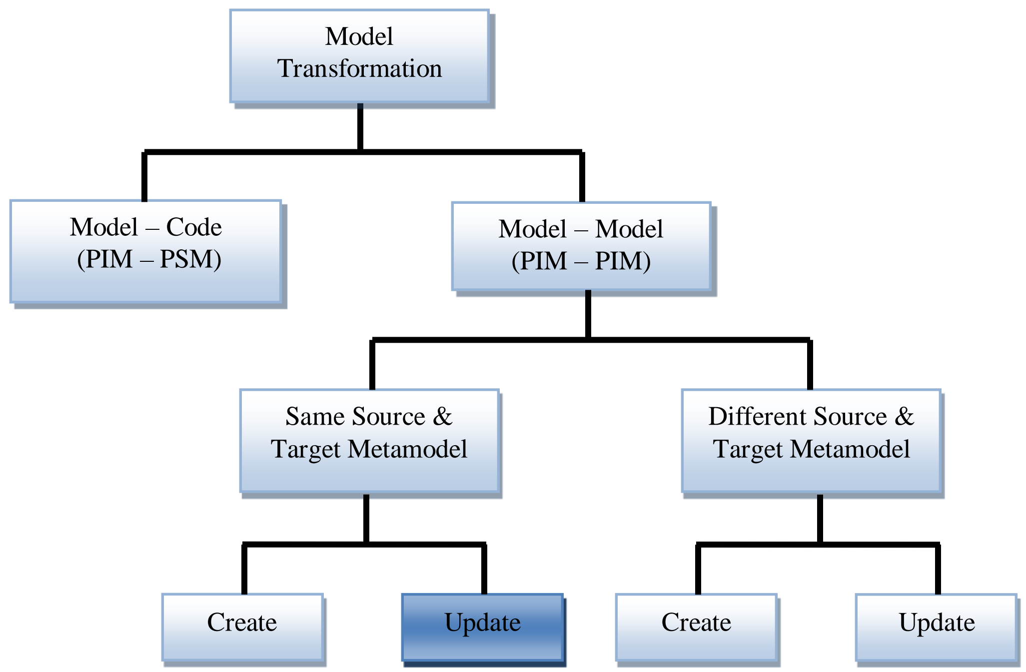


Figure 8 Taxonomy of Model Transformation

At a high level, model transformation approaches are classified into two categories: model-to-code transformation and model-to-model transformation. In model-to-code transformations, a metamodel of the target programming language is used. Discussion about this type of transformation is out of the scope of this work.

Model-to-model transformations are transformations that translate between source and target models at the same level of abstraction. The need for this kind of transformation is because

- Bridging Large Abstraction Gaps: The process of transforming PSMs to PIMs is easier when intermediate models are generated rather than a direct transformation. This makes the transformation modular and maintainable.
- Multiple Views and Synchronization: Model-to-model transformations are also useful for computing different views of a system model and synchronizing them.
- Formal Representation for Analysis and Verification: Informal models (such as UML) can be transformed into a formal modeling language in order to add preciseness and formality to the model. This aids in verifying the model for correctness.

Mens and Van Gorp [50] made a distinction between two kinds of model transformations: exogenous and endogenous. Exogenous Transformation is a model transformation where the source and the target metamodel are different and belong to two different domains. In contrast to exogenous transformations, if the source and target metamodel are identical we refer to it as endogenous transformation. Another slightly different classification of model transformation was provided by France and Bieman [8].

They classified model transformation approaches as vertical and horizontal. If the target model is at a different level of abstraction than the source model, the transformation is known to be vertical. On the other hand, if the source and the target model belong to the same level of abstraction, the transformation is known as horizontal. Also a transformation approach may create a new target model that is separate from the source or support an update of the existing source model. Czarnecki and Helson [47] further classified model-to-model transformation approaches based on the manner in which they are implemented. Categories include:

- *Direct Manipulation Approach:* These approaches provide an internal model representation of the source model and some APIs to manipulate it to generate the target model.
- *Intermediate Manipulation Approach:* In this approach, the source model is exported into a standard intermediate representation. An external transformation language or tool is used for applying transformations.
- *Transformation Language support Approach:* This category of model transformation approaches provides a mechanism for explicitly expressing, composing and applying transformations.

One of the most popular types of transformation classified under Endogenous Horizontal transformation is called Software Refactoring. Model refactoring is a special instance of model transformation where the source and the target models are instances of the same metamodel and operate at a higher level of abstraction. This type is shown in the Figure 8 with a darkened rectangle and is elaborated in section 2.4.1. In order for a model refactoring approach to be valuable for practical application, certain set of activities are

required to be specified as part of the approach. These activities, included as part of a transformation framework, are elaborated in section 2.5.

2.4.1 Software Refactoring

Refactoring, a term extensively acknowledged in the discipline of Object Oriented Programming, was defined by Opdyke as an outcome of his PhD dissertation [6]. It is an object oriented alternative to the concept of restructuring categorized as a software maintenance activity by Chikofsky and Cross [5]. According to Chikofsky and Cross, "*Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior*".

Fowler et al. [15] redefined refactoring highlighting its inherent advantages as "*a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.*" Fowler's definition emphasizes program understandability and maintainability. Fowler et al. also provided a comprehensive catalog of refactorings as part of their book [15]. Hence, refactoring is just a way of rearranging code.

The topic of refactoring at the level of source code has been extensively studied. With the growing popularity of MDA and UML, application of refactoring has been elevated to a more abstract level of design models. Hence, the term model refactoring or model-driven refactoring was proposed. The key motivations for shifting the focus of software refactoring from source code to design models can be summarized as follows.

- A model provides an abstract view of the system; hence, visualization of the structural changes required is easier.

- Problems uncovered at the design-level can be improved directly on the model.
- Exploring alternate decision paths is much cheaper at the design-level.

A simple illustrative example of a UML model refactoring is shown in Figure 9. It shows a class diagram in which two classes have attributes of the same type. Model refactoring removes this redundancy by introducing a new super class and moving the common attribute to this super class.

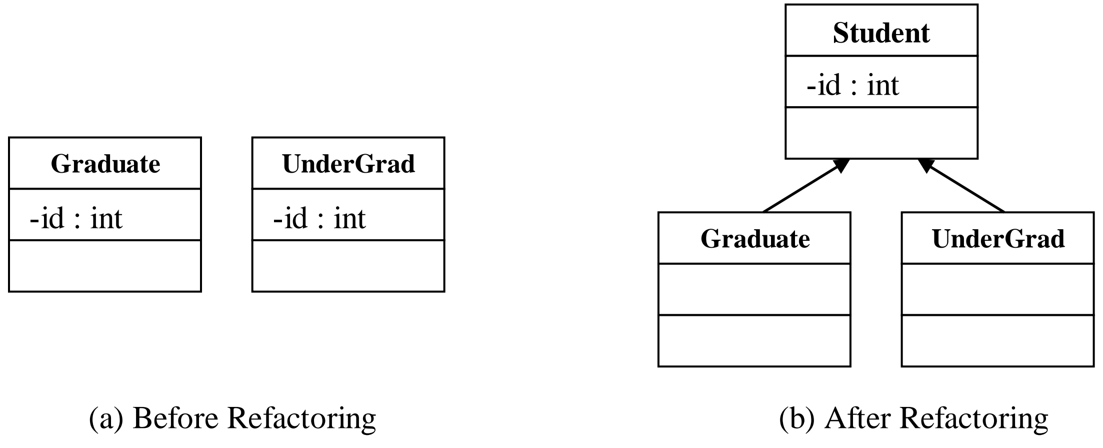


Figure 9 Model refactoring example

2.5 Model Transformation Framework

Model-driven refactoring is a special kind of model transformation that allows us to improve the structure of the model while preserving its internal quality characteristics. Model-driven refactoring is a considerably new area of research that still needs to reach the level of maturity attained by source code refactoring. Based on the information

obtained from source code and model-driven approaches, we identified a list of distinct activities that are essential for a model refactoring approach.

1. *Model Specification*: Select an appropriate language for specifying the model. Either a formal or an informal language can specify models. A formal language apart from specifying a syntax and semantics also provides a proof system for validation.
2. *Model Transformation Language*: A transformation language allows composition of rules that dictate the transformation process. The specification language along with the transformation language forms a Transformation System.
3. *Model Smells:* Model smells are portions within the model that need to be refactored. A number of detection strategies are available in the literature for identifying model smells. They are also referred to as Refactoring Opportunities.
4. *Model Behavior:* One important constraint posed by refactoring is the notion of behavior preservation. Since models are non-executable entities, the concept of behavior has to be defined and verified before and after the application of refactoring(s).
5. *Model Refactoring:* Select suitable refactoring(s) that can be applied at the identified location(s). Refactoring operations are chosen based on the smell identified.
6. *Refactoring Quality:* Evaluate the effect of refactoring on the quality of the software model.
7. *Tool Support:* Application of Refactoring is usually supported by a tool. A refactoring tool can either perform refactoring automatically without user intervention or requires user confirmation before application.

8. *Consistency Management:* Refactoring a model leaves other related models and source code inconsistent. In order to preserve consistency between the refactored model and other software models and source code, model consistency approaches need to be adopted.

2.5.1 Model Transformation System

A model transformation system (MTS) includes both the specification language and the transformation language. UML is a graphical notation designed to specify, visualize and document artifacts of s software system. It is a semi-formal language as its syntax and static semantics are precisely defined but dynamic semantics are not formally defined. The process of model-driven refactoring includes a number of activities such as behavior conservation, verification, synchronization etc. that requires a formal set of both static and dynamic semantics to ensure behavior-preserving transformation. Although many authors use UML metamodel and models as-is for model refactoring, they annotate the model with formal behavioral constraints using OCL. The importance of choosing a proper specification language can be understood clearly from the reasoning provided by Kalleberg [51]:

> *"The effectiveness and applicability of a software transformation system depends to a large extent on how its underlying program model has been formulated. The model determines which transformation tasks will be easy and which will be difficult or impossible. Particularly, the "abstractness" of the representation determines which analyses and transformations are possible – if the model is too abstract, refactoring is not possible, and if the model is too detailed, many analyses become too expensive".*

Apart from the specification language, transformation rules that dictate the transformation from source model to the target model are required to be specified. Languages or formalisms used to describe these rules are known as Model Transformation Languages (MTL). The choice of MTL depends on the selected model specification. A transformation rule is a depiction of how a collection of constructs in the source metamodel can be altered into one or more constructs in the target metamodel. A transformation rule consists of a Left-Hand Side (LHS) component and a Right-Hand Side (RHS) component. The LHS accesses the source model and the RHS component access the target model. Both the LHS and RHS components are described using model fragments (or patterns) with zero or more model elements. Popular model transformation systems include:

- *Graph Transformation System (GTS):* One of the most popular and widely used specification languages to represent UML models is graphs. The use of graphs to represent models is motivated by the fact that models are fundamentally graph-based in nature. A graph consists of a set of vertices (V) and a set of edges (E) such that each edge e in E has a source s(e) and a target t(e) in V. A graph is given as a tuple <V, E, s. t> where s and t are two functions that assign each edge a source and a target node. Graph transformation languages are based on algebraic graph grammars. There exist two paradigms for graph transformation approaches. The conventional paradigm, also known as Algebraic Graph Transformation, defines transformation rules declaratively. Transformation rules in Algebraic Graph Transformation have a Left Hand Side (LHS) graph and a Right Hand Side Graph (RHS). On application of the rule, elements in the LHS are deleted and the elements in the RHS are added. A

transformation rule also consists of an arbitrary number of negative application condition (NAC) [52] graphs. If the rule matches any of its NAC, then the rule cannot be applied. The other paradigm is known as Triple Graph Grammar (TGG). The transformation rules in TGG are always bidirectional. The relationship between the source graph and the target graph is described by a correspondence graph.

- *Logic Based System:* Another popular approach to represent UML models is logic-based representation. Logic is a formal system that allows definition of formulas representing propositions. Formulas can be derived by the use of well-defined rules and axioms also known as theorems. For instance, Boolean logic limits the truth-values of its propositions to two values: true and false. Popular logic based languages include Alloy [53, 54], Z notation [55], Object-Z [56] and Description Logic [57]. Primitive transformations in logic-based systems are formalized as algebraic laws that consist of templates with which the actual declarations match. Each law defines two templates of equivalent models on the left and the right side. Equivalence allows application of the law in both directions.
- *Direct Manipulation:* This approach allows direct manipulation of the metamodel without conversion to any other specification language. One of the main reasons for the popularity of this methodology is the availability of quite a few model-to-model transformation languages such as Query/View/Transformation (QVT) [58], Xpand [59] and the ATLAS Transformation Language (ATL) [60] to describe refactoring rules. OCL is usually used with UML to define pre and post conditions in order to ensure behavior preservation. Although popular, describing model refactoring transformations for UML models is not an easy task due to the complexity and

impreciseness of the UML metamodel. The main reason for the popularity of UML/OCL based approaches is the fact that OCL is both formal and simple when compared to other formal specification languages such as Z.

2.5.2 Model Smells

Martin Fowler et al. [15] were the first to introduce the concept of code smells:

> *"In doing so, we have learned to look for certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring".*

Similar to the concept of code smells, model smells can be defined as elements within the model that are potential candidates for improvements. Models Smells could either be symptoms of design defects or bad alternatives to recurring design problems in OO design also known as anti-patterns. Brown et al. [17] initially defined anti-patterns as structures that although may appear beneficial, but result in having negative consequences on the quality of the OO system. Not all the anti-patterns defined by Brown et al. [17] can be detected at the design level. In our work, we use the term model smells to refer to both design defect symptoms and anti-patterns.

The manner in which model smells are detected (also known as the detection strategy) has resulted in two paradigms of refactoring: *Metrics-Based Refactoring* and *Pattern-Based Refactoring*. Apart from these approaches, a hybrid approach that uses both metrics and patterns to describe smell detection strategy has also gained popularity.

1. *Metrics-Based Refactoring:* One methodology that gained immense popularity for detecting bad smells, proposing refactorings for correction and verification of quality improvements is Metrics-Based Refactoring. Metrics used for detecting model smell

belong to different metric suites [61-63]. An important aspect of using model metrics as a smell detection strategy is the threshold value of the metrics as it has decisive influence on detection accuracy. Marinescu [64] identified three ways of parameterizing threshold values for metrics used for smell detection as follows:

a. Empirical results from metrics' authors and similar past experiences
b. Using a Tuning Machine to find proper threshold values for regulating the detection strategy automatically [65]. This approach uses an examples repository of flaw samples and selects those values that maximize the number of correctly detected samples.
c. Analyzing multiple versions for change stability information or persistency of a design flaw over time [66]. Although this approach does not help in parameterizing a threshold value, it provides a value time perspective for each potential entity and hence improves the accuracy of the detection process.

2. *Pattern-Based Refactoring [67]:* Another popular method to detect refactoring opportunities is to identify problems within the model that can be solved by applying design patterns. Design patterns are defined as solutions that can be reused for a recurring design problem. It typically shows relationships between classes or objects. The concept of using design patterns to solve common design issues in order to speed up the software development process was initiated by Gamma et al. [68]. The field of identifying symptoms for design related problems and using design patterns to solve them is termed as Pattern-Based Model Refactoring. Design patterns in this paradigm are represented as a triple (PM, SM, T), where PM (Problem Model) is a model describing the design problem, SM (Solution Model) is a model describing the

solution and T is a transformation that transforms an input model presenting with an instance of PM and replacing it with the corresponding solution model SM.

3. *Rule-Based Detection [24]:* This smell detection strategy identifies both model smells and anti-patterns using a declarative rule definition. These rules are manually defined to identify the symptoms that characterize the smell. A rule-based method can be perceived as a hybrid approach that uses metrics, structural patterns and lexical information to form rules that query the source model for design defects or anti-patterns. Rule-based detection approaches either use complex queries or algorithms to detect refactoring candidates.

2.5.3 Model Behavior

One important constraint associated with application of refactoring is behavior preservation. By definition, model-driven refactoring is an activity to restructure models in order to improve model quality without changing its observable behavior. In order to demonstrate whether a refactoring operation is behavior preserving, concept of model behavior needs to be precisely defined.

The most popular approach to define model behavior is through the use of model constraints such as pre-conditions and Invariants. Pre-conditions are assertions that a model must satisfy prior to the safe application of refactoring. These conditions characterize valid model transformations. As their name implies, pre-conditions must be checked before refactoring is executed. Invariants are conditions that must remain true before and after refactoring. Usually, preconditions are checked prior refactoring to ensure invariants hold after the refactoring operation. Establishing preconditions and invariants for refactoring properly is very important. Lax definitions will allow

refactoring operation to be executed but may not preserve model behavior. On the other hand, severe unnecessary preconditions may not allow refactoring application even when required.

2.5.4 Refactoring Quality

An important objective of Model-Driven Refactoring is to improve the quality of the software model without changing its behavior. Only a few studies elaborate the concept of Model Quality and address the issue of quality assessment for UML models. One of the most popular approaches to assess the quality of models is using model metrics [69]. Similar to software metrics, model metrics are also used to measure and quantify desirable aspects of the models. Some software metrics can easily be ported to models, especially those that measure object oriented source code.

2.5.5 Refactoring Tool Support

Based on the activities required for Model-Driven Refactoring, it is evident that in order to be completely practical, tool support is necessary to cover the entire range of designated activities. Refactoring tools can be classified based on their degree of automation: Manual, Semi-Automated and Fully-Automated. A fully-automated tool provided automatic detection and correction of design defects without user intervention. Semi-automated tools require interaction with the user throughout the refactoring process. A semi-automated refactoring tool assists the user by proposing refactoring opportunities and their suggested solutions. The decision to perform the actual transformation is left to the user. Manual refactoring tools are UML modeling tools that leave the process of model smell detection and application decision to the user

completely. Manual refactoring tools automate behavior preserving model transformations only.

Another important requirement for refactoring tools is their deployment mode. A refactoring tool can either be a standalone prototype tool or developed as a plugin to an existing Integrated Development Environment (IDE). The importance of integration into an existing IDE on usability of the tool was provided by Egyed [70].

> *"A final challenge is that all of the above should be implemented in model-driven development environments in an as efficient and scalable way as possible, otherwise it will never be adopted by practitioners".*

2.5.6 Consistency Management

Spanoudakis and Zisman [71] defined inconsistency as "*a state in which two or more overlapping elements of different software models make assertions about aspects of the system they describe which are not jointly satisfiable*". They provided an in-depth survey of inconsistency management approaches available to the field of software engineering. With respect to Consistency management due to software refactoring, two kinds of inconsistencies are observed [72].

- *Vertical Inconsistency:* When source code/design model is refactored, corresponding design artifacts / source code becomes inconsistent.
- *Horizontal Inconsistency:* Since a modeling language such as UML is typically composed of many different diagrams, the issue of consistency between all these diagrams needs to be addressed. This need arises when any one of them evolves or refactored.

CHAPTER 3

LITERATURE REVIEW

This section reviews the literature on code based refactoring, model based refactoring and integrated refactoring. It is worth-mentioning that an extensive survey of software refactoring emphasizing on source code refactoring has been conducted as part of an initiative called "*The Refactoring Project*" at the Universiteit Antwerpen [11, 12, 73-75] A similar state of the art survey was also done for model based refactoring by Mens et al. [13].

3.1 Code Based Refactoring

Opdyke [6] introduced the concept of object oriented refactoring in 1992 as "*program restructuring transformation that supports the design, evolution and reuse of object-oriented application frameworks*" as a result of observing the evolution of object oriented programs and Database Schemas [76]. Opdyke compiled a set of twenty-six low-level refactorings and a set of three high-level refactorings assembled from the low-level refactorings. In order to assure behavior preservation, he identified invariants and augmented his refactorings with pre-conditions to ensure that these invariants were preserved even after the refactoring process.

Following upon the foundation laid by Opdyke, Dan Roberts in 1999 [77] supplemented Opdyke's refactoring definition with post-conditions. These post-conditions specify how

the pre-conditions are transformed by the refactorings thereby reducing program analysis effort after the refactoring. Research in the area of code refactoring has been done at length. To simplify the presentation of the research conducted in improving code-level refactoring, we organize it in line with the activities involved in the refactoring process. Wake [16] suggested that a refactoring process should first identify portions within the software that needs refactoring. Then an appropriate refactoring is selected and applied to this portion. Mens et al. [73] added three more steps to provide a complete list of six distinct refactoring activities. These activities are as follows:

1. Identify portions within the software that needs to be refactored (*Bad Smell Identification*).
2. Select suitable refactoring(s) that can be applied at the identified location(s) (*Refactoring Suggestion*).
3. Verify behavior preservation for the applied refactoring(s) (*Behavior Preservation*).
4. Apply the refactoring(s) (*Refactoring Application*).
5. Evaluate the effect of the refactoring(s) on the quality of the software or the process (*Refactoring Effect Evaluation*).
6. Preserve consistency between the refactored code and other software artifacts (*Consistency Preservation*).

3.1.1 Bad Smell Identification and Refactoring Suggestion

Steps 1 and 2 are usually coalesced as studies identify fragments of code in need of refactoring and propose a suitable approach to handle them. Software in this section refers to source code, as refactorings related to software models is discussed in Section 3.2. Portions of code in need of refactoring are referred to as bad smells or code smells.

Martin Fowler et al. defined bad smells in code as "*certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring*" [15]. They identified twenty-two bad smells for code. These bad smells were later classified by Wake [16] into smells within and between the classes.

Looking for bad smells requires analysis of source code that can be done with either static information or dynamic information. Static analysis of source code is preferred over dynamic ones, as the latter requires execution of the source code to obtain its runtime behavior. Static analysis can be done either by lexical analysis of the source code or by projecting the source code over a graphical representation. Well-known graphical notations to represent source code include Abstract Syntax Trees (AST) [78-80], Program Dependence Graphs (PDG) [81-83] and Type Graphs [84, 85].

Analyzing software artifacts to identify structural shortcomings is also one of the strategies used to detect bad smells. Code smells proposed by Fowler et al. [15] fall into this category of smell detection approaches. Studies analyzing structural anomalies use strategies such as search algorithms over code parse-trees [86], analyzing internal method structure [87] and discovering relationships between entities [88, 89].

Code duplication or cloning, one of the bad smells proposed by Fowler et al. [15], is considered one of the worst smells that affects software maintainability. A number of code-clone detection approaches have been discussed in the literature. Popular methods employed for code-clone detection include lexical analysis [90-98], graph-based traversal and slicing [78-83, 99, 100] and pattern recognition [101]. Although code duplication is considered a bad smell, a few studies [102-104] have shown that cloning can sometimes

be reasonable and beneficial to the design. Hill and Rideout [105] proposed the use of machine learning algorithms on near duplicate code segments for automated code completion.

A popular approach to detect refactoring opportunities or code smells is the use of source code metrics. Simon et al. [106] referred to this strategy of code smell identification as *Metrics-based refactoring*. Research studies based on metrics to identify code smells use object-oriented metrics such as coupling, cohesion, inheritance and complexity [64, 107-117]. Although popular, some authors claim that metrics are not sufficient in precise detection of bad smells [118-120]. Improvements to the metric based approach include using code patterns, heuristics and machine learning. Pattern based approaches define bad smells as patterns of source code [119]. Heuristics-based approaches use a combination of traditional object-oriented metrics composed as functions to evaluate software quality attributes. These functions are then used to identify refactoring opportunities [121-126]. Machine learning approaches that use metrics to predict refactoring opportunities is also gaining immense popularity. Prevalent machine learning algorithms employed by smell detection approaches include Naïve Bayes [127-129], C4.5 [130], clustering algorithms [131-136] and Fuzzy Logic [137].

Another popular approach is the identification of opportunities where design patterns can be inserted in the source code. Design patterns [68] are reusable solutions to commonly occurring problems in software design. This strategy is referred to as *Pattern-based Refactoring* [138]. A pattern-based approach initially identifies problem location in a program and then recommends an appropriate design pattern to transform the program. A number of approaches focus exclusively on design pattern based approaches were

proposed [139-143]. Shimomura et al. [144] use genetic algorithms to assess the quality of a program based on design patterns.

A number of studies assessed the effect of bad smells on software evolution. These studies investigate the evolution of bad smells over multiple versions of software systems [145-148]. A state-of-the-art in bad smell detection and refactoring suggestion approaches can be found in these studies [149, 150].

3.1.2 Behavior Preservation

The most important aspect of refactoring and the most difficult one to specify and verify is the notion of behavior. Opdyke [6] introduced the concept of preconditions to handle behavior specification and preservation. Preconditions ensure that provided the same set of input values to the source model before and after refactoring, it should always produce the same result. Although preconditions provide a good notion of behavior, they do not consider the size of the program and hence static checking of these preconditions before the application of refactoring can become very expensive. Roberts [77] augmented his refactorings with post-conditions. Roberts was able to prove theoretically that a set of post-conditions can be translated into a set of equivalent preconditions that later formally proved by Heckel [151]. Post-conditions not only enable specification of invariants that depend on dynamic information easier but also increase the efficiency of the refactoring tool by postponing the evaluation of a constraint.

Mens et al. [85] proposed a relaxed notion of behavior preservation claiming that full behavior preservation is impossible. Based on their notion, a program will perform the same actions before and after refactoring execution if:

- The refactoring is *access preserving*. That is if each method at least accesses the same variables, directly or indirectly, before and after the refactoring.
- The refactoring is *update preserving*. That is if each method at least updates the same variable before and after refactoring.
- The refactoring is *call preserving*. That is if each method at least performs the same method calls before and after refactoring.

Another pragmatic approach to verify behavior preservation is through rigorous testing. Although sensible, this approach cannot definitely claim behavior preservation due to the relationship between code structure and tests. Hence, any modification done to code structure may alter the test results even though refactoring does not alter behavior [152, 153].

3.1.3 Refactoring Application

After identifying the refactoring opportunities, the next step is to correct them by refactoring application. Authors in refactoring literature propose refactoring application in two ways: *implicit* and *explicit*. Implicit approaches loosely associate refactorings and model smells. The basis of selecting a particular refactoring from the provided options is not expressed clearly [15, 73, 130]. In implicit approaches, multiple corrective solutions are possible for the same smell. Explicit approaches associate refactoring operations with different kind of refactoring opportunities and are clearly expressed. Most of the approaches using metrics-based, learning algorithms and pattern-based techniques adopt the explicit approach. Prior to correcting the identified defects, two important issues must be addressed: order of bad smells to resolve and the sequence of refactoring application.

Ranking refactoring opportunities in the literature is proposed based on their impact on software quality [88, 154-156], software faults [157] and based on past source code modifications [158]. Cheng and Liao [159] proposed a taxonomy of code smells based on their semantic relationship from the viewpoint of refactoring application.

An important aspect prior to rule application is to suggest the sequence of refactoring applications. Suggesting a particular refactoring sequence may require an effort that is comparable to the one of re-implementing part of the system from scratch. This suggestion should take into account the dependency and interrelationship between relevant refactorings to produce a practical sequence. A number of approaches to sequence refactoring applications have been proposed in the literature. Prominent ones include critical pair analysis [160] and graph unfolding [161, 162] for graph based representations, simplification rules over Definite Finite Automata representation [163], constraint programming [164, 165], multi-objective optimization [166, 167], set pair analysis [168, 169] and genetic algorithms [170] . Other approaches target improvements over specific object-oriented principles such as cohesion [171, 172] and inheritance [173]. Arcoverde et al. [174] conducted a survey to understand the longevity of code smells and ranked refactorings based on difficulty, priority and frequency of use.

A popular trend with refactoring application is the use of evolutionary algorithms. This paradigm of software refactoring is known as *Search-based Refactoring* [171, 175]. Normal refactoring approaches require manual specification of rules, metrics and patterns to identify refactoring opportunities and suggest refactoring operations to remove these anomalies. These approaches suffer from a number of complications such as right metric combinations, metric threshold values, and calibration and so on. Search-based

refactoring treats refactoring as a combinatorial optimization problem. Hence, the approach tries to find a correct combination of refactoring operations that maximize the number of corrected defects and improve overall software quality. A number of search techniques are used in the literature such as Genetic Algorithms [171, 173, 176], Simulated Annealing [173, 177], Multiple Ascent Hill-Climbing [173], Steepest Ascent Hill-Climbing [173], Steepest and Multiple Descent [177] and Artificial Bee Colony Search [177] to search for an optimal combination of refactoring application.

In order for refactoring application to be feasible and practical, tool support integrated as part of the state-of-the-art Computer-Aided Software Engineering (CASE) tools is essential. Refactoring tools usually support identification of code flaws [24, 64, 178-180], refactoring code [181-183] and both [124, 184-187]. A state-of-the-art in software refactoring tools can be seen at [188].

3.1.4 Refactoring Effect Evaluation

Apart from identifying refactoring opportunities within source code, metrics are also used to measure the effect of refactoring on internal and external software quality. Impact of refactoring on internal quality attributes have been studied by [88, 189-191]. Quite a few efforts have been made to study the impact of refactoring on external quality attributes too. Some prominent attributed studied include Understandability [192-194], Changeability [195], Maintainability [110, 193, 194, 196, 197], Reusability [198] and Testability [193, 194]. Ratzinger et al. [199] used data mining and classification algorithms to evaluate the effect of refactoring on the number of software defects. They claim that the number of software defects decreases as the number of refactorings increase over earlier versions of software.

Quite a few studies have also discussed the impact of design patterns on software quality. A number of authors have studied the impact of design patterns on internal quality attributes (coupling, size and inheritance) [200-202]. Khomh and Gueheneuc [203] and Ampatzoglou [204] studied the impact of design patterns of external quality attributes such as reusability, expandability and understandability. They claim that design patterns do not necessarily improve quality and some patterns in turn decrease some quality attributes. Authors in [205-208] have also investigated the correlation between design pattern introduction and change proneness. Elish and Alshayeb [209] proposed a classification of refactoring methods based on their effect on software quality.

3.1.5 Consistency Preservation

Since consistency preservation is concerned with both the source code and models at higher level of abstraction, the topic is discussed independently in section 3.3.

3.2 Model Based Refactoring

Model-driven refactoring is a special kind of model transformation that allows us to improve the structure of the model while preserving its internal quality characteristics. Model-driven refactoring is a considerably new area of research, which still needs to reach the level of maturity attained by source code refactoring. Refactoring at model level is more multifaceted and challenging than at source code level. This is due to the existence of multiple views. A view is a collection of diagrams that illustrate similar characteristics of the system.

Approaches to model-driven refactoring can be classified into two categories based on the number of views considered when refactoring: *single view* and *multiple views*. Each of these approaches can be further classified as either *operational* or *relational* [31]. The operational approach allows definition of model refactoring and provides methods to automate them whereas the relational approach allows verification for behavior preservation and consistency.

Since then, a few surveys covering state-of-the-art [13], taxonomy [210], open issues and challenges [11, 12, 14, 74, 211] related to model-driven refactoring have been published in the literature. We could discuss the literature in the area of model-driven refactoring along several ways. However, keeping in line with the organization of code based refactoring in the previous section; we chose to organize the literature by model refactoring activities. The refactoring process for models consists of a number of distinct activities (based on the refactoring process for source code refactoring [73]):

1. Select an appropriate language for specifying the model. Either a formal or a non-formal language can specify models. A formal language apart from specifying a syntax and semantics also provides a proof system for validation. *(Model Specification)*
2. A transformation language allows composition of rules that dictate the transformation process. The specification language along with the transformation language forms a Transformation System. *(Model Transformation Language)*
3. Model smells are portions within the model that needs to be refactored. A number of detection strategies are available in the literature for identifying model smells. They are also referred to as Refactoring Opportunities. *(Model Smells)*

4. One important constraint posed by refactoring is the notion of behavior preservation. Since models are non-executable entities, the concept of behavior has to be defined and verified before and after the application of refactoring(s). *(Model Behavior)*
5. Select suitable refactoring(s) that can be applied at the identified location(s). Refactoring operations are chosen based on the smell identified. *(Model Refactoring)*
6. Evaluate the effect of refactoring on the quality of the software model. *(Refactoring Quality)*
7. Application of Refactoring is usually supported by a tool. A refactoring tool can either perform refactoring automatically without user intervention or requires user confirmation before application. *(Tool Support)*
8. Refactoring a model leaves other related models and source code inconsistent. In order to preserve consistency between the refactored model and other software models and source code, model consistency approaches need to be adopted. *(Consistency Management)*

3.2.1 Model Specification

One of the most popular and widely used specification languages to represent UML models is graphs. The use of graphs to represent models is motivated by the fact that models are fundamentally graph-based in nature. As models are required to conform to their metamodel, graphs must conform to the corresponding type graph [212]. A typed graph ensures whether or not a graph is well formed or not. Figure 10 obtained from [213] visualizes the relationship between metamodel, model, graph and type graph.

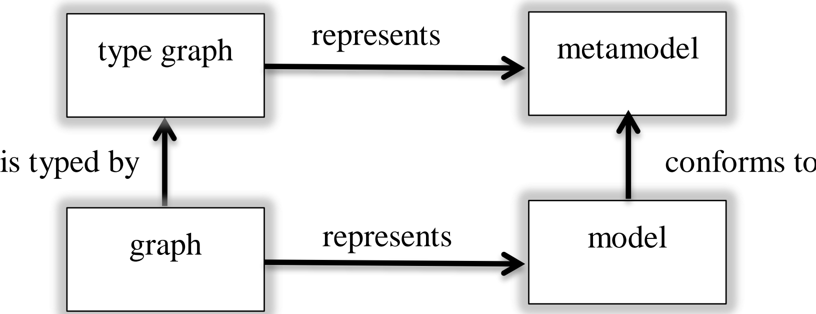


Figure 10 Relationship between models and graph representation

The use of directed typed graphs for model specification is a common approach [161, 213-217]. The graph that contains nodes and edges and relationship between them is similar to the relation between objects and classes in UML. Another popular type of graph used for model specification is the directed attributed type graph to represent the UML metamodel [160, 218-220]. The use of an attributed type graphs allows inclusion of object-oriented concepts such as attributes, relationships and multiplicities to be added to the type graph.

Another popular approach to represent UML models is logic-based representation. A popular logic based language used to represent UML models is Alloy [53]. Alloy is a formal object-oriented language based on first-order relational logic. A relational language is a set of all relational sentences formed from a *relational signature* and a *function*. A relational signature is composed of a set or sequence of constants that can be either objects, functions or relations. An Alloy model is a sequence of *paragraphs*. A paragraph can either be a *signature* that defines new types or formal paragraphs used to record constraints. With the proposition of Alloy 3 [54], inheritance concepts can be integrated as the language allows a signature to extend another signature. A number of

approaches [221-224] used the Alloy language to present a formal type system and semantics for object-oriented UML model specification.

The Z notation [55] is a formal specification language gaining popularity in the field of formal software engineering. Z is mathematical notation based on set theory, calculus and first-order predicate logic and is used for describing and modeling computing systems. Z is a declarative language that describes the system states and models their change under the execution of operations. The main construct of a Z specification is called a *schema*. A schema consists of variable declarations and predicates defined over the variables. Estler and Wehrheim [225] used formal specifications written in Z for refactoring but verification of these refactorings were carried out in their approach using an Alloy based constraint solver. An object-oriented variation of the Z notation known as Object-Z [56] has become popular with UML modeling. Object-Z supports object-oriented concepts such as classes, polymorphism and inheritance. Estler et al. [226] used the Object-Z notation in their approach to represent UML models. A formal notation using both Object-Z and process algebra CSP (CSP-OZ) for modeling the static view and dynamic view was used by Derrick and Wehrheim [227] and Ruhroth et al. [32].

Another popular logic based approach is Description Logic (DL) [57]. Spanoudakis and Zisman [228] highlighted two important limitations of first-order logic based approaches. They pointed out that first-order logic is semi-decidable hence not sufficient to provide semantically adequate inferences and the process of theorem proving is computationally inefficient. Description Logic is a less expressive formalism than first-order logic but provides more reasoning capability and is decidable. Knowledge in DL is represented as concepts, roles and individuals. Individuals are instances of defined concepts and related

to each other by roles. DLs use a small set of constructors (operators) to construct complex concepts and roles. Approaches using DL to represent UML diagrams translate the metamodel into DL concepts and roles. Classes are mapped to concepts and associations are mapped to roles [229-231].

Saadeh et al. [232] also used logic to represent UML models. In their approach, elements in the UML model are represented as logic terms called *Model Element Terms (METs)* and presented as Prolog facts to take advantage of Prolog's search engine and backtracking techniques.

With the proposition of the QVT (Query/View/Transformation) [58] standard by Object Modeling Group, UML metamodels accompanied by OCL are used in the context of a model transformation language. Sunye et al. [233] were the first one to use the UML/OCL model for refactoring. Other studies that followed the trail included [21, 234-236]. Apart from being used as a formal representation for the UML model, OCL expressions have also been used for refactoring application [237-240].

3.2.2 Model Transformation Language

Languages or formalisms used to describe transformation rules are known as Model Transformation Languages (MTL). Although there exist quite a few model transformation languages, we limit our scope to those that have been used in the literature for the purpose of UML model refactoring.

Alloy language can also be used to compose model transformations. A catalog of primitive transformations was proposed by Gheyi et al. [222]. Primitive transformations are formalized as algebraic laws that consist of templates with which the actual Alloy

declarations match. Each law defines two templates of equivalent models on the left and the right side. Equivalence allows application of the law in both directions. The catalog of Alloy laws have been proven to be sound and complete [241] and can be used for behavior-preserving transformations such as model refactoring. Gheyi et al. [221] presented an approach for proving structural model refactorings for Alloy. They presented an example Alloy Law (or model refactoring) to introduce a generalization into an Object Model. Massoni et al. [223] presented refactorings as Alloy transformation done at two levels: program level and the object model level. They extended their contribution in [224] with description of synchronization and proof soundness for the Alloy transformations.

Graph transformation languages are based on algebraic graph grammars. A number of approaches make use of graph transformation languages to define model refactoring rules [214, 215, 217, 242]. Mens [213] specified design models as typed graphs and expressed refactorings as typed graph transformations. They evaluated two concrete graph transformation tools (AGG and Fujaba) for composing model refactoring rules over class diagrams and statecharts. They also proposed the use of critical pair analysis to detect implicit dependencies between refactorings. A critical pair is a pair of transformation that conflict with each other. Set of critical pairs represents all conflicts when applying model-refactoring rules to a model. Based on their previous work, Mens et al. [160] implemented a number of refactorings from Fowler et al.'s catalog as typed graph transformations with NACs. Junbing et al. [243] proposed a conflict resolution algorithm to handle model refactoring conflicts based on critical pair analysis.

Bottoni et al. [244] used the Double Push-Out (DPO) scheme to define model refactoring rules over a number of UML models and source code in an integrated fashion. The DPO scheme for graph transformation ensures that the target graph has no dangling edges after the application of the transformation as opposed to the Single Push-Out (SPO) scheme. Rangel et al. [245] also used the DPO graph transformation scheme for model refactoring. Amelunxen and Schürr [246] also used graph transformation approach to specify dynamic semantics of modeling languages and provided formalization of model refactoring rules over the latest version of UML/MOF 2 metamodel.

3.2.3 Model Smells

Models smells in the literature are identified either by analyzing the source code and then applying them at the model-level [24, 247-250] or by analyzing the model directly. Since the scope of this paper is limited to Model-Driven Refactoring, we list approaches that analyze design models directly to detect refactoring opportunities. The manner in which model smells are detected (also known as the detection strategy) has resulted in two paradigms of refactoring: *Metrics-Based Refactoring* and *Pattern-Based Refactoring*. Apart from these approaches, a hybrid approach that uses both metrics and patterns to describe smell detection strategy has also gained popularity known as rule-based approach.

One methodology that gained immense popularity for detecting bad smells, proposing refactorings for correction and verification of quality improvements is Metrics-Based Refactoring. Most model smells identified in the literature relate to the UML class diagram as it is the most frequently used model in OO software development. Astels [251] was the first one who proposed the notion of UML model smells in the context of

model refactoring. He argued that the visual notation of UML models makes smell structures more evident. Model smells in his work were described informally using the visual notation of UML. Kempen et al. [25] identified the model smell *"God Class"* defining it as a single class with many attributes and/or operations. Threshold values for metrics associated with this model smell were not provided in his work.

Ruhroth et al. [32] and Fourati et al. [19] associated different class model smells and anti-patterns with OO metrics. Specific values for metric thresholds in their approaches were either taken from published empirical results or based on their experiences. A few model smells in their approaches not only covered class models but also considered metrics over the statechart diagram and sequence diagram respectively. Ghannem et al. [22] used an advanced evolutionary approach to model smell detection. Instead of specifying which metrics to use or their threshold values, they used Genetic Programming to choose the best metrics combination from an exhaustive list to detect different model smells.

Mohamed et al. [20] proposed an extension to the UML metamodel incorporating model smell and refactoring meta-classes in order to assist users to create their own smell-refactoring definition. For each model smell, information such as metric-based heuristics required detecting them and the UML diagrams on which these metrics depend on are attached. They demonstrated their approach by detecting the *Blob* anti-pattern using design metrics based on Class and Sequence Diagram. A list of model smells and the UML diagrams they relate to is tabulated in Table 1.

Table 1 List of model smells detected using OO metrics

Model Smell	UML Diagram	References
God Class or The Blob	CD, SD	[19, 20, 22, 25]
Hidden Concurrency	CD, SC	[32]
Unnecessary Behavioral Complexity	CD, SC	[32]
Too Low Cohesion	CD	[32]
Lazy Class	CD	[32]
Too Strong Coupling	CD	[32]
Refused Bequest	CD	[32]
Lava Flow	CD, SD	[19]
Functional Decomposition	CD, SD	[19, 22]
Poltergeists	CD, SD	[19]
Swiss Army Knife	CD	[19]
Poor use of Abstract Class	CD	[22]

Note: CD: Class Diagram, SD: Sequence Diagram, SC: Statecharts

Another popular method to detect refactoring opportunities is to identify problems within the model that can be solved by applying design patterns. France et al. [67] were the first to propose the use of design patterns for model transformation. Kim et al. [252] later investigated approaches for incorporating design patterns into UML models. They defined design patterns in terms of roles at the metamodel level. A role is based on a UML meta-class and associates a set of constraints (well-formedness rules, pre and post conditions and invariants) on the meta-class to adapt to the type of elements they can play the role. Kim [253] used the concept of roles to describe the problem model (called *Problem Specification* in their approach). A problem specification described the problem that suggested the usage of a particular design pattern. To specify the problem, Kim [253] used a methodology they developed earlier [254]. Ballis *et al.* [255, 256] proposed a graphical language to define patterns/anti-patterns either textually or using the graphical notations. An important aspect of their approach is that they allow users to customize existing pattern descriptions or create new from the start.

El-Boussaidi and Mili [257] extended the UML-based representation of the problem models to represent time evolution. In order to detect instances of problem models within a source model, they translated their pattern matching problem into a *Constraint Satisfaction Problem* (abbr. CSP; by extracting variables and constraints from the problem model) and used a CSP solver to find the instance. El-Sharqwi et al. [258] also used CSP to formalize their algorithm for problem model detection. They used XML to represent both the pattern and the software model. Bouhours et al. [259] defined the concept of Spoiled Patterns to specify problem patterns for refactoring. According to them, a spoiled pattern is an abstraction of an alternative solution which is a less optimal solution (optimal being the design pattern) to solve a design problem. Millan et al. [234] proposed an extended OCL language (*pOCL*) and demonstrated its use to find occurrences of an alternative solution within a source input model by using OCL rules.

Rule-based detection approaches either use complex queries or algorithms to detect refactoring candidates. Llano and Pooley [23] provided an informal specification approach to describe anti-patterns that appear in UML diagrams. They exemplified their approach by providing a specification of two popular anti-patterns: *God Class* and *Poltergeists*. Detection strategies for these anti-patterns are specified in their approach by a textual description which can easily be translated into a query for automated detection. Akiyama et al. [260] proposed the refactoring of Class diagram by redistributing class responsibilities in order to obtain a design model of higher quality. Responsibilities in their approach are initially obtained from *Requirements Specification* and assigned to classes based on the GRASP (*General Responsibility Assignment Software Pattern*)

guidelines [261]. Model smells are detected by formalizing the guidelines proposed by GRASP using predicate logic.

Other UML models used for model smell detection using the rule-based method include sequence diagrams and use case models. Dobrzański and Kuźniarz [262] used a rule-based approach to describe the *middle man* model smell. A *middle man* class in their approach is defined as one that has an attribute with at least 2 *Simple Delegating Operations* (SDO). Operation for an attribute is classified as an SDO based on a set of conditions. Liu et al. [263] represented the UML sequence diagram as a suffix tree and proposed a special algorithm to detect longest common prefixes of its suffixes in order to identify duplicated fragments. A fragment of a sequence diagram is a rectangular unit whose edges are parallel to the diagram's axes. El-Attar and Miller [264] identified 26 anti-patterns for use case models and provided a query based approach to formulate the detection of these anti-patterns using OCL language. Stolc and Polasek [236] described their refactoring approach using a graphical definition of the refactoring rules with the smell defined on the left side of the rule and the solution on the right side. OCL queries are generated automatically for the model smell defined on the left side of the rule for smell detection.

OCL is used heavily in the field of Model-Driven Refactoring to fulfill numerous purposes such as: constraint specification, specification of pre and post conditions on operations, specification of well-formedness rules for metamodels and as a query language. Because of these important applications, it is important to ensure that specifications written in OCL are easy to understand and maintain. Hence, the notion of

OCL smells analogous to Model Smells was defined. Correa et al. [240] defined OCL smells as:

> *"Structures present in OCL expressions that might negatively affect the understandability or maintainability of OCL specifications".*

The concept of OCL smells was initially introduced by Correa and Werner [237]. They identified OCL smells that either required refactoring of the OCL specification or in a few cases warranted changing the associated UML Class model. They classified these smell into three categories: those that affect only OCL expressions, those that result from refactoring the underlying class diagram and finally those that require modification of the underlying class model as a result of changes made to the OCL expression. The authors extended their approach with an enhanced list of OCL smells [238] and an empirical study to demonstrate their impact on understandability [239, 240].

3.2.4 Model Behavior

Specification of model behavior and approaches to ensure their preservation after the application of refactoring is considered the most important activity in the refactoring process as refactoring is supposed to *preserve observable behavior*. The most popular approach to define model behavior is with model constraints such as pre-conditions and invariants. Model constraints are assertions that a model must satisfy prior to the safe application of refactoring. Constraints can be in the form of pre-conditions that must be checked before refactoring is executed or post-conditions that are checked after the application of refactoring. OCL is the most popular language used to define model constraints.

Most of the approaches in the literature describing model behavior through pre and post conditions use OCL constraints [21, 215, 224, 233-235, 237, 240, 248, 262, 265-267]. Other scripting languages used to describe constraints include python scripts [268] and ECL [269]. A number of approaches that do not provide prototype tool described model constraints in natural language form [260, 270, 271]. Although easier to comprehend, using informal approaches to describe behavior affects implementation and automatic verification of behavior preservation.

Graph based approaches usually describe pre-conditions using *negative application conditions* (NAC). An NAC is a graph that defines a prohibited graph structure in order to restrict application of refactoring rules. Approaches in the literature that used NAC along with refactoring operations to describe pre-conditions include Bottoni et al. [244, 272], Mens et al. [213] and Hosseini and Azgomi [273].

The other common approach to specify model behavior is through process algebra CSP [274, 275]. Behavior preservation in this approach is verified by proving that the target model is a *refinement* of the source model. The theory of failure-divergence refinement in CSP is used to demonstrate behavior preservation in refactoring. Another similar approach is proving behavior preservation through model *equivalence*. Equivalence is refinement in both directions. Studies proving behavior preservation by model equivalence used different formalisms and tools. Estler *et al.* [226] used Object-Z as the specification language and proved equivalence using the model checker SAL [276]. Derrick and Wehrheim [227] and Ruhroth et al. [32] also used Object-Z but proved refinement using CSP. Other formalisms used to prove equivalence include co-algebra [277] and Alloy [221, 223, 278]. Another approach worth mentioning is the use of

behavioral models to specify behavior for structural diagram refactoring [231] . They formalized Sequence Diagram traces to prove observation and invocation call preservation.

3.2.5 Model Refactoring

The class diagram, apart from being the most frequently used model in object-oriented development, is also the most frequently researched diagram for model-driven refactoring. According to Mens et al. [13], the main reason class diagrams are profoundly investigated is because of their close similarity in representation to object-oriented program structure. Hence most program refactorings [15] can be ported directly to UML class diagrams. Sunyé et al. [233] were the first to try refactoring on UML models. According to Sunyé et al., the primary advantage of UML over other modeling languages is that the syntax is defined precisely by a meta-model. They transposed some of the existing code based refactorings onto the class diagram. Astels [251] also informally defined some class diagram refactorings. His motivation for selecting the class diagram was that it is easier to comprehend the structure when looking at the class diagram rather than the source code. Boger et al. [279] also provided refactorings for class diagrams. Class diagram refactorings in their approach were classified into five basic categories: Addition, Removal, Move, Generalization and Specialization over attributes, methods and associations.

Refactoring over behavioral models is still in its infancy. Sunyé et al. [233] initially applied refactoring to statecharts. They introduced the following refactorings: Fold and Unfold Incoming / Outgoing Actions, Fold and Unfold Incoming / Outgoing Transitions, Grouping States and Moving Atomic States In and Out of Composite States. Phillips and

Rumpe [280] extended the definition of UML refactoring over system structure diagrams and state diagrams. Phillips and Rumpe's approach made use of a new transformation language introduced as part of a project titled Computer-Aided Intuition-Guided Programming project (CIP) [281]. Boger et al. [279] also provided refactorings for state diagrams and activity graphs. Refactoring in activity graphs involved changing the order of activities without altering the overall result.

With respect to refactoring of functional models, only a few approaches address the problem of refactoring in use case diagrams. Rui and Butler [282] were the first to initiate the application of refactoring over use case models. No formal definition of a use case model was provided as they used a three-layer meta-model based on Regnell's use case model [283] to base their refactorings over. Rui and Butler decomposed their model into entities like use case, actor, user, task, goal, service, episode and so on. Refactoring operations were then identified over these entities like creation, deletion, modification and move operation. One interesting refactoring operation was decomposition of use cases to distribute its behavior. Their approach was refined by Yu et al. [270] with the introduction of the concept of "episode tree". Complex episodes were decomposed into one or more child episodes and similar episodes were merged together to form a composite episode. The Episode Tree provided a visualization of the whole episode hierarchy, which made them introduce new refactoring operations like *generalization_generation, inclusion_mergence, extension_mergence* and *precedence_mergence*. This whole set of refactorings introduced in [270, 282, 284] was later put together in the form of a tool to realize use case refactoring [284, 285]. The tool provided additional features by including OCL to define constraints among the entities

and XML to store the model. The missing element of formal semantics in order to validate the behavior preservation property of refactoring for use case models was provided as part of the PhD thesis by Kexing Rui [286].

3.2.6 Refactoring Quality

An important objective of Model-Driven Refactoring is to improve the quality of the software model without changing its behavior. Only a few studies elaborated on the concept of Model Quality and addressed the issue of quality assessment for UML models. One of the most popular approaches to assess the quality of models is using model metrics. Similar to software metrics, model metrics are also used to measure and quantify desirable aspects of the models. Some software metrics can easily be ported to models, especially those that measure object oriented source code.

One of the most widely used design metric suite for OO programs was provided by Chidamber and Kemerer [62] also known as the CK metric suite. Genero et al. [63] proposed a set of complexity measures based on the UML class diagram. Kim and Boldyreff [287] proposed a number of metrics that can be used at early stages of software development. Their metric suite covered class, sequence and use case diagrams. Gronback [288] provided a broad collection of UML metrics to detect aberrations from standard design practices. A few of these practices were derived from style guidelines provided by Ambler [289]. Enckevort [21] used four out of the six metrics from the CK metrics suite to quantify model quality. They also chose the Fan-In and Fan-Out metrics proposed by Henry and Kafura [290]. In their approach to assess refactoring quality, they calculated metrics for the model before and after the application of refactoring.

As mentioned earlier, modeling in UML is multi-faceted. UML diagrams model different views of the system and these diagrams are not mutually disjoint. Multiple views of the UML models provide information not available from program code. Muskens et al. [29] proposed metrics that combine information from multiple views. Lange [291] also postulated that metrics for establishing Model Size requires information from multiple views of the model.

Apart from defining metrics for UML models, correlation between these metrics and external model quality attributes needs to be established. Lange and Chaudron [69] developed a quality model for UML based on the ISO quality model [292] and McCall quality model [293] for software quality. We refer to this model as LC model. The LC model is a hierarchical model with four levels. The first or the highest level defines the primary uses of the model: Maintenance and Development. The second level defines the purpose of the model within its primary application. The third level identifies the characteristics of the purposes and the fourth level defines the metrics and the rules for the assessment of the characteristics. Jalbani et al. [294] proposed an integrated quality engineering approach for UML models. They divided their approach into two parts: Quality Assessment and Quality Improvement. Quality assessment includes the Quality Model for UML based on the LC model and metrics for UML. Quality Improvement includes model smell detection and model refactoring.

3.2.7 Tool Support

Two of the most widely used state-of-the-art UML modeling CASE tools for implementing model refactoring are Eclipse and Poseidon. One of the first model refactoring tool *Refactoring Browser* proposed by Boger et al. [279] implemented

refactoring rules for UML class, state machine and activity diagrams. The refactoring browser was integrated in Poseidon. However, this plugin is not available anymore for Poseidon. RACOoN is another plugin developed for Poseidon by Van Der Straeten and D'Hondt [230]. Model refactoring rules can be implemented and loaded into RACOoN for execution. It's a manual refactoring tool that lets the user select the refactoring. Inconsistencies encountered during application of multiple rules are presented to the user with resolution options.

Eclipse is an IDE from the Eclipse foundation [182] that uses a plugin mechanism to allow integration with various projects. Together Architect for Eclipse is a model transformation environment that supports the QVT standard. It supports only textual notation of QVT and the user invokes each refactoring rule separately. Markovic and Baar [235] used this modeling environment to implement their refactoring rules. Voigt and Ruhroth [295] developed a fully-automated tool (called RMC) as an Eclipse plugin that enables model creation, measurement, diagnosis and refactoring. It also allows users to define their own measures (using OCL queries), threshold values (range of values) and refactoring. Available refactorings are stored in a configuration file that can be extended by the user. They used the tool to propose a quality cycle for software model development [32]. VisTra [236] is a visual oriented tool implemented as an Eclipse plugin for refactoring class models. It provides a rule editor that allows users to define transformation rules graphically. The VisTra framework automatically generates OCL query and transformation script for the transformation rule. End users can invoke transformation rules on the UML model. M-Refactor is another model refactoring plugin for the Eclipse Modeling Environment developed by Mohamed, et al. [20]. It's a semi-

automated refactoring tool that detects model smells on the source model based on the value of the metrics based heuristic. Based on the detected smell, model refactoring solutions are presented to the user. Since their approach is based on an extended UML metamodel, information regarding the smell, metric thresholds and refactoring solutions are represented by meta-classes.

Another popular approach to implement model refactoring is using standard programming language scripts. Porres [268] first proposed the System Modeling Workbench (SMW) toolkit based on Python programming language to implement UML model refactorings. Similar approach was followed by Correa and Werner [238] to refactor OCL expressions using their OCL extension called OCL-Script. Since OCL-Script is an action language as opposed to standard OCL, it can be used to manipulate metamodel-level and model-level instances. The NEPTUNE platform [234] is a prototype tool that allows verification and transformation of models. It uses an extension of the OCL language called pOCL to automate the detection of model fragments that can be substituted by structural design patterns. It is a semi-automatic refactoring tool that suggests the user to substitute the detected problem pattern with the corresponding design (solution) pattern.

Zhang et al. [269] used the Constraint-Specification Aspect Weaver (C-SAW) model transformation engine to describe model refactoring. They proposed a special language, called Embedded Constraint Language (ECL), to specify and implement user-defined refactorings. According to Zhang et al. [269], ECL is an extension of OCL with additional operations for model aggregation and transformation. C-SAW is developed as

a plugin for Generic Modeling Environment (GME), which is a UML meta-modeling environment.

As evident from section 3.2.2, Graph Transformation is one the most popular model transformation language used to express refactorings. Two general-purpose graph transformation tools commonly used to specify model refactorings are AGG [296] and Fujaba [297] .

AGG is rule-based visual programming environment that supports graph transformation. A number of studies made use of the AGG tool to implement their refactoring rules. Kazato et al. [218] formalized model refactoring using graph transformation in AGG. AGG, apart from providing model transformation primitives, also provides advanced mechanisms such as critical pair analysis that can be used for analyzing refactoring rules [160] and an Application Programming Interface (API) that allows programmers to use the transformation engine with other environments. Folli and Mens [216] used the AGG API to develop a model refactoring application in Java.

Fujaba on the other hand implements a controlled graph transformation approach. Transformation rules and their order of application are represented in Fujaba by a compact notation called story diagrams (which is a combination of activity diagram and collaboration diagram). Geiger and Zündorf [298] exemplified statechart refactoring using the Fujaba CASE tool. In their approach, they flattened nested statecharts into plain state machines for the sake of refactoring. Grunske et al. [214] used the Fujaba tool set to implement graph transformation. Mens [213] implemented model refactoring rules in

both AGG and Fujaba for comparison. He identified both positive and negative aspects of using both tools for implementing model refactoring.

Sunyé et al. [233] were among the first to propose refactorings over UML models. They created an initial set of Class model refactoring and statechart refactoring. They implemented their approach in a general-purpose transformation framework called UMLAUT (Unified Modeling Language All pUrposes Transformer) [299].

Prototype tools based on XSLT for model transformation have also gained popularity. This is mainly because most UML modeling environments export model diagrams as XMI. The UML Model Transformation Tool (UMT-QVT) proposed by Oldevik [300] is an open source tool based on XSLT. An alternative approach was proposed by Peltier et al. [301] to use XSLT to execute model transformation on the back-end instead of specification. They used a high-level transformation language (MTRANS) to specify refactoring rules that were later converted to XSLT programs before execution. Ren et al. [284] proposed a prototype tool for use case refactoring based on XML. It is composed of a Refactoring Tool GUI and a Use Case Diagrammer to draw the use case model. Following the approach by Peltier et al. [301] to use XSLT at the back-end, Li et al. [302] proposed an approach to use QVT relations to specify transformations and implement each relation as an XSLT rule template. The main reason specified for using XSLT as a back-end language is due to its low-level syntax.

AndroMDA is another prototype MDA tool that allows generation of complete applications from a UML model. Although its focus is on code generation, Mens et al. [13] demonstrated its use for model refactoring. El-Boussaidi and Mili [257] proposed a

semi-automatic tool for marking models using constraint satisfaction technique and rewriting source models to incorporate appropriate solutions. Their main aim was to propose a framework for detection of problem patterns that can be solved by design patterns.

Apart from these tools, a number of prototype tools have been proposed in the literature to validate their proposed approaches [22, 24, 260, 264]. These tools cannot be classified as refactoring tools as they do not provide complete refactoring functionality. These tools aid the user either in detecting model smells or anti-patterns in models or calculating metrics for quality assessment.

3.2.8 Consistency Management

Another vital aspect of model refactoring is model consistency. Lucas, et al. [303] provided an excellent systematic review of the literature on inconsistency management in software engineering domain. The need of consistency management with model refactoring arises because UML is composed of many different diagrams. Refactoring one diagram leaves the others in an inconsistent state. Since consistency preservation is concerned with both the source code and models at higher level of abstraction, the topic is discussed independently in section 3.3.

3.3 Refactoring Consistency Management

Consistency, as defined by Spanoudakis and Zisman [228] is "*a state in which two or more elements, which overlap in different models of the same system, have a satisfactory*

joint description". Inconsistencies occur if a change in an element is not correctly reflected on all overlapping elements in other models of the same system. Based on the description in section 2.5.6, inconsistencies occur at two levels: *horizontal* and *vertical*.

Horizontal consistency, also known as intra-model consistency, aims at identifying and resolving inconsistencies between models at the same level of abstraction. Intra-model consistency approaches tend to handle both syntactic or structural consistency and semantic or behavioral consistency. Intra-model consistency management approaches can be classified into four classes based on the formalism and technique used for checking consistency: **1)** Direct, **2)** Transformational, **3)** Formal and **4)** Knowledge Representation. A systematic literature review of UML model consistency approaches can be found in [303].

Direct approaches to consistency management use OCL to define consistency checking rules over UML. Chiorean et al. [304] proposed the use of OCL to validate models against well-formedness rules and also it to define inter-model consistency rules. Spanoudakis and Kim [305] conducted a series of experiments to evaluate the impact on the whole model based on an inconsistency involving a particular element. Based on this framework, Spanoudakis et al. [306] proposed a set of significance-ranking rules formalized in OCL based on the impact of consistency violation. Other approaches that used OCL to define model consistency rule include Paige et al. [307] and Sapna and Mohanty [308].

Transformational approaches transform one model into another or to a common notation and apply comparison techniques to establish consistency. Graaf and van Deursen [309]

used ATL transformation language to specify mapping between state machines and scenario diagrams. Egyed [310] proposed a transformation-based inconsistency management approach between class diagrams at different levels of abstraction. Egyed [28, 70, 311], proposed a *model profiler* to establish a correlation between model elements and consistency rules based on the manner in which they are accessed during consistency checks. Since the use of graphs to represent models is a popular approach, many authors propose consistency management approaches based on graphs. Mens et al. [312] used graph transformational analysis and critical pair analysis to identify and establish casual dependencies between alternative resolutions for model inconsistencies. Kuster [313] proposed a graph based approach to handle behavioral model consistency between sub-models of a larger model at the same level of abstraction and models generated during different phases of software development. Fryz, and Kotulski [314] used Conjugated Graphs for representing UML models. Other approaches using graphs for consistency checking and resolution include [315, 316].

Formal approaches convert the UML model into a formal notation to check and resolve inconsistencies. Formal specification languages (FSL) are popularly used as they provide precise descriptions of the software system and can be formally analyzed. Commonly used FSLs in the literature for UML consistency management include Z [317], Petri Nets [318-321], , Symbolic analysis [322], B [323-325], pi-calculus [326], Constraint Programming [327], PVS specification language [328, 329], CSP-OZ [330] and automata [331, 332].

Knowledge representation approaches use logical representation languages such as Description Logic to translate models and use reasoning for consistency management.

Van Der Straeten et al. [230, 231, 333] used description logic to represent UML models as collection of concepts and roles and logic rules to detect and suggest means to resolve inconsistencies. Other logic based representations include Temporal Logic [334] and Maude [303, 335].

Vertical consistency aims at identifying and resolving inconsistencies between models and source code. Massoni [336] identified three common approaches for handling code-model consistency. These are as follows:

- *Simple forward engineering:* Models are used only in the initial stages of development and discarded later. Hence, changes are made only on the source code that renders the consistency issue useless.
- *Successive reverse engineering:* Source code is the primary artifact and models are generated as physical images of the source code. Reverse Engineering tools are used to maintain consistency.
- *Round-trip engineering:* Models are used to generate source code during implementation. Once a stable version of the source code is available, reverse engineering tools are used to ensure model consistency.

Since the approaches highlighted by Massoni rely heavily on the use of reverse engineering tools to reconstruct models based on source code modifications, a number of alternate approaches to handle the issue of vertical inconsistency have been proposed. Bottoni, et al. [244] used distributed graphs for model transformation. A distributed graph consists of a network graph. Each network node is refined by a local object graph and network edges are refined by graph morphisms on local object graphs. The graph

morphisms describe how the object graphs are interconnected. Their main objective in using distributed graphs was to describe synchronized transformation on distributed models: *diagrams* and *code*. Code was represented by a flow graph and diagrams were represented by a typed graph in their approach. Common interface parts are represented using an interface graph. For instance, the interface between class diagrams and flow graphs will present *Method*, *Variable* and *Type* nodes. Refactorings are then described by a set of coordinated graph transformations, which is instantiated on code modification and applied to an appropriate model affected by the change. Van Gorp et al. [248] proposed the idea of source consistent refactoring to handle vertical consistency. Since UML models do not model statements in method bodies, Van Gorp et al. constructed their own metamodel called *GrammyUML*. This metamodel added eight extensions to the UML 1.4 specification allowing them to model statements in method bodies.

3.4 Metamodel Extension

UML models are described by a metamodel. A UML metamodel is a qualified alternate of the UML models and is a representative of any diagram that can be expressed with it. UML provides well-defined ways to extend the metamodel. These extension mechanisms allow designers to customize and extend the syntax and semantics of the model elements. The two extension mechanisms provided are 1) by augmenting the metamodel itself (heavyweight extension) or 2) by constructing a profile (lightweight extension). A UML profile is a predefined set of stereotypes, tagged values and constraints to support modeling in specific domains. Profiles give a well-defined manner of adopting the

standard UML model to a particular domain. Since a profile is not a new element, its expressiveness is constrained by the model element it specializes. Augmenting an element to metamodel allows the addition of a new model element or meta-class to the standard UML abstract syntax.

Extending the UML metamodel has been a common practice in the literature in order to enhance the expressive power of UML to model object-oriented designs. Most of these proposed extensions stem from the motivation when the existing UML specification fails to represent the semantic meaning of the design. The extension mechanisms provided by UML have been utilized in numerous applications. These applications include modeling OO frameworks [337], integrating software architecture descriptions [338-340], agent oriented systems [341-343], design composition [344], aspect oriented system [345, 346], modeling variability in families of systems [347], adding business goals to activity diagrams [348], representing XML Schemas [349], secure systems development [350] and web applications [351].

Although most of the works mentioned above concentrated on adding domain-specific structural information to the UML metamodel, there were also few initiatives made to extend the behavioral elements of the UML metamodel. Metamodel for sequence diagrams has been the primary focus of extension to integrate domain specific behavioral information. da Silva and de Lucena [352] enriched the UML sequence diagram with explicit information to represent the exchange of messages between agents. Based on their earlier work [343] on adding structural elements to the UML metamodel, interactions between these elements to model the dynamic aspects of a Multi-Agent System were proposed. Padilla et al. [353] proposed a notation to specify multiplicities

over a classifier in a sequence diagram. In order to provide an interpretation of multiplicities, they extended the UML metamodel and demonstrated how interaction operators behave in the presence of this additional information. Harel and Maoz [354] extended and defined a subset of the UML language called Modal UML Sequence Diagrams (MUSD). This extension allows fragments or part of fragments to be either mandatory (universal) or optional (existential).

Apart from proposing new behavioral elements for the UML metamodel, proposals redefining current behavioral constructs were also suggested. Refsdal and Stølen [355] proposed the addition of risk related information to the UML Sequence diagram. They proposed an operator "*palt*" that adds probabilistic choice to the existing "*alt*" operator. Haugen et al. [356] proposed an external mandatory operator "*xalt*" that specifies that one of the alternate cases in that fragment must be possible.

Heavyweight extensions have also been applied to the other UML models in the literature. Metamodel for use case diagrams has been the primary focus of extension to add narrative information to the model. These modifications involve extension of the metamodel to incorporate the behavioral properties as described in the textual descriptions of the use case model.

An extension to the UML metamodel for use cases was initially proposed by [357] for their XML-based requirements verification approach. They proposed a simple extension wherein a use case is composed of a sequence of steps. Each step refers to an optional condition, set of exceptions or an action. The metamodel defined as part of their approach

distinguished between different actions such as actor's actions (by the actor), system's actions (by the system) and use case actions (inclusions and extensions).

Rui and Butler [282] proposed a use case metamodel based on a single use case modeling notation. Elements in their metamodel are divided into three levels. Environmental level is similar to that of the UML use case metamodel, which includes actors, use cases and other feature based information such as goals, services and tasks. At the structural level, use case from the previous level is further decomposed into a series of episodes along with preconditions and post-conditions. In the event level, each episode from the structural level is further decomposed into events. An event is further classified as stimulus, response or an action.

Diaz et al. [358] proposed a use case specification metamodel as an extension to the use case package of the UML metamodel. Each use case in their proposed extension includes a specification element, which is composed of two different paths in a textual specification: basic and alternative. Each path is composed of a sentence which is classified as either a simple sentence or a special sentence (extends, include and control).

Metz et al. [359] did not propose an extension to the use case metamodel but provided an in-depth explanation of the different types of alternative flows in a use case description. They focused on unifying specific notational issues such as alternative flow types in use case modeling. This concept of use case variability specification was later integrated into a use case metamodel extension proposed by Bragança and Machado [360]. They extended the use case metamodel with new model elements in order to clarify the use case relationships (extend and include). The Extend meta-class from the UML metamodel

was extended to include extension fragments. They associated a rejoin point (the return location within the base use case after execution of the extension fragment) with each extension fragment.

Hoffman et al. [361] recently proposed a narrative metamodel for textual use case descriptions specifying the behavior of use cases in a flow-oriented manner. The main motivation behind their approach was to ensure consistency between UML use case model and its descriptions. Each use case from the UML use case model is described as flow of events, which is easily comprehendible by both technical and non-technical stakeholders.

Zelinka and Vranic [362] proposed a precise definition of different use case templates so as to allow a consistent application. Their goal was not to unify the UML use case model with its textual description, but to map the common and variable part among different template descriptions. This allowed flexibility of using a single notation or a combination of several use case description notations.

Somé [363] proposed a use case specification metamodel which is formally defined as an extension to the UML metamodel specification. He provided a set of constraints that ensure consistency between use case descriptions and use case models. He also enhanced the degree of expressiveness by introducing control flow structures for iteration and concurrency and definition of variable custom traits.

The most recent extension proposed for a use case metamodel was by Repond et al. [364]. Particularly they modeled the generalization relationship within a use case behavior, which was not provided by earlier proposed extensions. They also defined the

concept of use case scenarios that represents a specific path among all the possible flows of the use case. Hence, each use case consists of multiple scenarios where each scenario has a sequence of steps that model a specific flow path. Although the concept of scenarios was put forward earlier by [282], it was not properly explained in their work.

Extensions proposed by [282, 357, 359, 360] fail to include the concept of flows (or scenarios) within use case descriptions. Extensions proposed by [358, 361-363] explicitly modeled flows as a set of steps within a use case description. Although these works modeled use case flow, the lowest level of abstraction in their work is a use case step of which a flow is composed. In our proposed extension to the use case metamodel, we considered different form of steps within a use case flow and each action step further is modeled to the level of fine grain system-user interaction. Apart from this, we modeled the concept of use case transactions useful for applications such as effort estimation and use case analysis.

Almost all extensions proposed to the UML use case metamodel do not model the generalization relationship except for the metamodel proposed by Repond et al. [364]. Their work introduced a *GeneralizationPoint* where specialized use cases can add additional behavior. In our proposed extension, a specialized use case cannot only add additional behavior, but it can modify or replace the steps of the generalized use case. Also the concept of *GeneralizationPoint* within the generalized use case defeats the purpose of generalization (i.e. allowing the generalized use case to have knowledge of what all use cases specializes it and where they add additional behavior).

Finally, all proposed extensions in the literature cannot be used for use case analysis and evaluation due to lack of information modeled such as different actor types, use case transactions and structure for use case constraints. In our proposed extended use case metamodel, we incorporated all required information for use case analysis and evaluation enhancing the usability of a use case model while maintaining the level of stakeholder comprehension. This extension allows easy integration of the use case metamodel with the metamodels of sequence and class diagrams.

CHAPTER 4

INTEGRATED METAMODEL

This chapter discusses the construction of the Integrated metamodel. Information regarding the UML metamodel, its contents and extension mechanisms supported are introduced initially. The diagrams that form the elements of the integrated metamodel are then discussed individually and finally the composition method employed to build the integrated metamodel and the metamodel itself is discussed.

4.1 UML Metamodel

Software development is classified as a methodology. Methodology as defined by Henderson-Sellers [365]

> *"A methodology has several constituent parts including a full lifecycle process, a comprehensive set of concepts, a set of rules, heuristics and guidelines underpinning appropriate development techniques, a set of metrics, information on quality assurance, a set of coding and other organizational standards and advice on reuse and project management"*

In simpler terms, methodology can be defined as a systematic approach to getting work done in a particular discipline. Following this line of definition, Software Development Methodology can be defined as a systematic approach to design and development in the

software engineering discipline. Methodologies can either be expressed in natural language description or by a modeling language if the underlying methodology is complex and non-trivial. Since software design is composed of diagrams and elements that refer to each other in complicated manner, it is more viable for it to be expressed by a modeling language rather than described through natural language.

Due to the popularity of the Object-oriented (OO) paradigm, UML has been adopted as a modeling language to express OO development methodology. Gonzalez-Perez and Henderson-Sellers [366] defined a relationship between a methodology, model and a metamodel as

> *"If a methodology is a model, creating that methodology is modeling, whereas creating the language concepts used to describe the methodology is metamodeling".*

The Object Management Group currently defines the UML language using a metamodel. The UML specification document [10] defines the metamodel in three different parts. These parts are

1. ***Abstract Syntax:*** A class diagram describes the abstract syntax of UML, which is composed of meta-classes and meta-associations. A meta-class describes each model element (e.g. Class, Attribute, Lifeline, Use case etc.) and meta-associations describe the interrelationships between these meta-classes. Syntax of UML is well defined and unambiguous.
2. ***Well-formedness Rules:*** Specification of constraints on instances of the meta-classes (that represent the UML language constructs) is through a set of *well-formedness*

rules. These constraints for well formedness are semi-formal specified by a combination of OCL expressions and an informal description.

3. **Semantics:** Semantics describe the meanings of the meta-classes introduced in the abstract syntax. Semantics of the metamodel consists of natural language description of the language constructs and their collaboration. Although the use of natural language makes them easier to understand, it also includes some incomplete and ambiguous information.

One of the main reasons of why UML is popular among OO developers is because it allows extension or even modification of the base language metamodel in order to adapt the language to a specific situation or domain. Categories of extension mechanisms provided by UML include: (1) Lightweight extension mechanism and (2) Heavyweight extension mechanism.

- ***Lightweight extension mechanism:*** Lightweight extension mechanisms are termed as lightweight because they do not add new model elements to the UML metamodel. UML profiles are used to implement these types of extensions. A UML Profile [10] is a collection of extensions that are packaged together to customize UML for a particular domain. It specifies a set of standard elements, well-formedness rules and semantics, beyond those specified by the UML metamodel. A UML profile consists of stereotypes, tagged values and constraints. Tagged values allow association of user defined variables or metadata to a model element. A tag value is represented by a name-value pair and must be compatible with the constraints of the base class of the model element. Constraints on the other hand, allow addition of semantic restrictions to the model elements. Constraints, similar to UML semantics, are written in OCL

and must also be compatible with the constraints of the base class of the model element. Tagged values and constraints are grouped under a meaningful name that forms a stereotype. Stereotypes are defined as extension to the UML model elements which implies that the tagged values and constraints it contains are associated with the model element implicitly. The keywords <<*stereotype*>>, <<*TaggedValue*>> and <<*Constraint*>> are used when including them in the extended metamodel. The relationship between stereotypes, tagged values and constraints as part of a UML profile metamodel is shown in Figure 11.

- ***Heavyweight extension mechanism:*** Adding new elements in the form of meta-classes, defining suitable metadata and meta-associations is referred to as heavyweight extension. These extensions are guided by the Meta-Object Facility's (MOF) meta-metamodel language [45]. The MOF meta-metamodel is a standard provided by OMG for specifying, interchanging and extending the UML metamodel. The metamodel constructed by using the heavyweight extension mechanism is more expressive but might end up with an exceedingly complex notation. Both these approaches have their share of advantages and disadvantages. Using the lightweight extension mechanism allows the availability of standard UML notation and hence generic UML tools could be used. On the other hand, the stereotypes must adhere to the constraints of the base element it extends which severely limits its expressiveness. Using the heavyweight extension mechanism makes the metamodel incompatible with UML-compliant tools, as the notation would not conform to UML standard. However, using this extension mechanism allows addition of any desired feature to the metamodel.

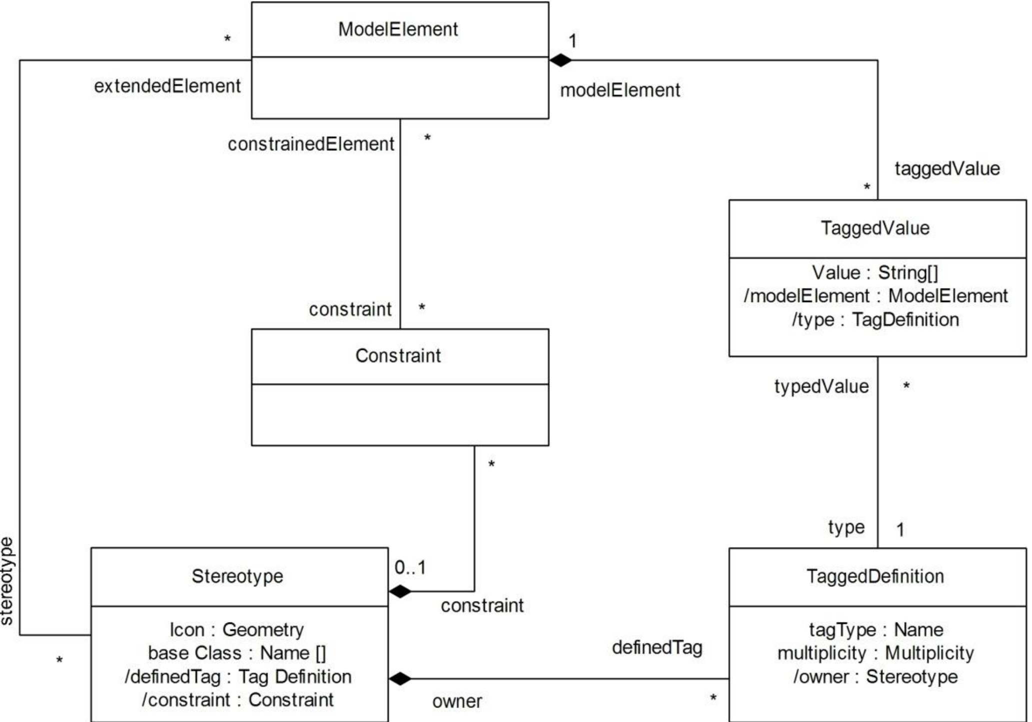


Figure 11 UML Profile Metamodel

Instances of the UML metamodel form a suite composed of all the UML models. UML models are classified into three categories based on the aspect of the system they describe. These categories are referred to as *views*: structural view, behavioral view and functional view. The structural view consists of diagrams that capture the physical organization of the basic elements (classes, objects etc.) in the system. It describes the static structure of the system. The behavioral view consists of diagrams that focus on the interactions between the elements in the system. This view represents how elements work together, interact, and respond to the environment. The functional view is a collection of diagrams that depict how a system is supposed to work, modeling the workflow and business processes. It captures information about the system from the user's perspective. Figure 12 shows the classification of the UML diagrams into views.

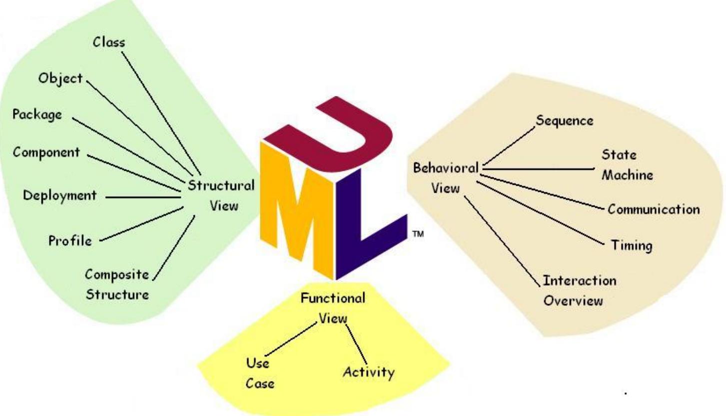


Figure 12 Classification of UML Diagrams into Views

The Integrated Metamodel proposed in this chapter is composed of one model from each view. Class diagram from the structural view, sequence diagram from the behavioral view and use case diagram from the functional view are used as core models for composing the integrated metamodel. A metamodel description of the models selected from each view is provided and then the integrated metamodel is discussed. Although UML metamodel does not differentiate between model elements, subsets of UML metamodel are referred to here as class diagram metamodel, sequence diagram metamodel and use case diagram metamodel. These subsets include all model elements that are used when constructing respective models.

4.2 UML Class Diagram

Class diagram represents the structural view of an object-oriented system. It consists of a set of classes designating important entities of the system modeled. Along with classes, a class diagram also consists of relationships between these classes. It is the most common diagram and considered as the backbone for modeling object-oriented systems.

A formal syntax for class diagram along with semantics is provided by Meng and Aichernig [367]. Utilizing their work with minor modifications (to incorporate features introduced in UML 2 specification), a formal definition of the UML Class diagram metamodel is provided here. Formally, a class diagram can be defined as:

Definition 4.1: A class diagram is a 4-tuple $CD = \{\mathbb{C}, \mathbb{A}, \mathbb{R}, WF_{CD}\}$ where

- $\mathbb{C}$ is a non-empty finite set of classes
- $\mathbb{A}$ is a finite set of associations
- $\mathbb{R} \in \mathbb{C}\ x\ \mathbb{C}$ is the relationship between classes
- WF_{CD} is a set of well-formedness rules on the Class Diagram CD

4.2.1 UML Class Diagram Metamodel

The UML specification document describes the UML abstract syntax in the form of a class diagram representing the UML metamodel and well-formedness rules. The UML class diagram metamodel is composed of a number of meta-classes. Some of these meta-classes may not be useful for the intended application of refactoring; hence, a subset of the UML Class diagram metamodel to be used for the integrated model is given in Figure 13. A detailed description of the abstract syntax and well-formedness rules of UML class diagrams is provided in Appendix 1.

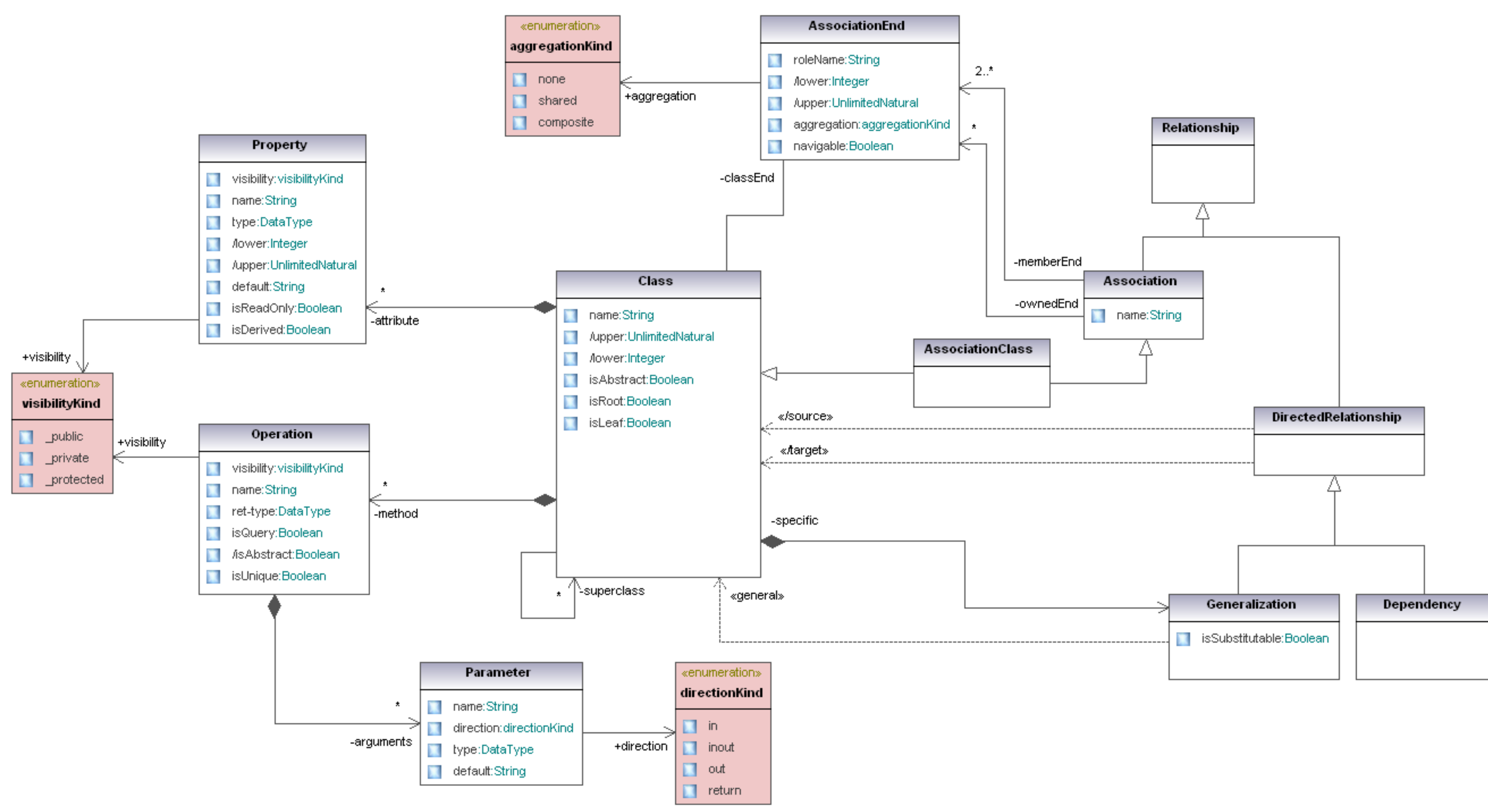


Figure 13 Subset of the UML Class Diagram Metamodel

4.2.2 Class Diagram Metamodel Extension

The metamodel for class diagram is used as-is without any extension. The reason for not extending the class diagram metamodel is that it is extensively and precisely described in the UML specification.

4.3 UML Sequence Diagram

Sequence diagram represents the dynamic view of an object oriented system. The main purpose of a sequence diagram is to capture dynamic behavior of a system. This is realized by modeling flow of events leading to a desired result.

Formally, a sequence diagram can be defined as:

Definition 4.2: A sequence diagram is a 7-tuple $SEQ = \{\mathbb{L}, End, Mes, \mathbb{E}, \leq, fragment, WF_{SEQ}\}$ where

- $\mathbb{L}$ is a finite set of lifelines
- End is a finite set of end locations
- Mes is a finite set of message labels
- $\mathbb{E} \subseteq End \times Mes \times End$ is the relationship (event) between lifelines
- $\leq \subseteq End \times End$ is a partial order providing the position of ends within each of the lifelines
- $fragment$ is an ordered set of fragments in the sequence diagram
- WF_{SEQ} is a set of well-formedness rules on the Sequence Diagram SEQ

4.3.1 UML Sequence Diagram Metamodel

Similar to that of the Class diagram, the UML Specification document also describes the Sequence Diagram metamodel by an abstract syntax in the form of a class diagram and the well-formedness rules. A subset of the UML Sequence diagram metamodel to be used for the integrated model is shown in Figure 14. A detailed description of the abstract syntax and well-formedness rules of UML sequence diagrams is provided in Appendix 1.

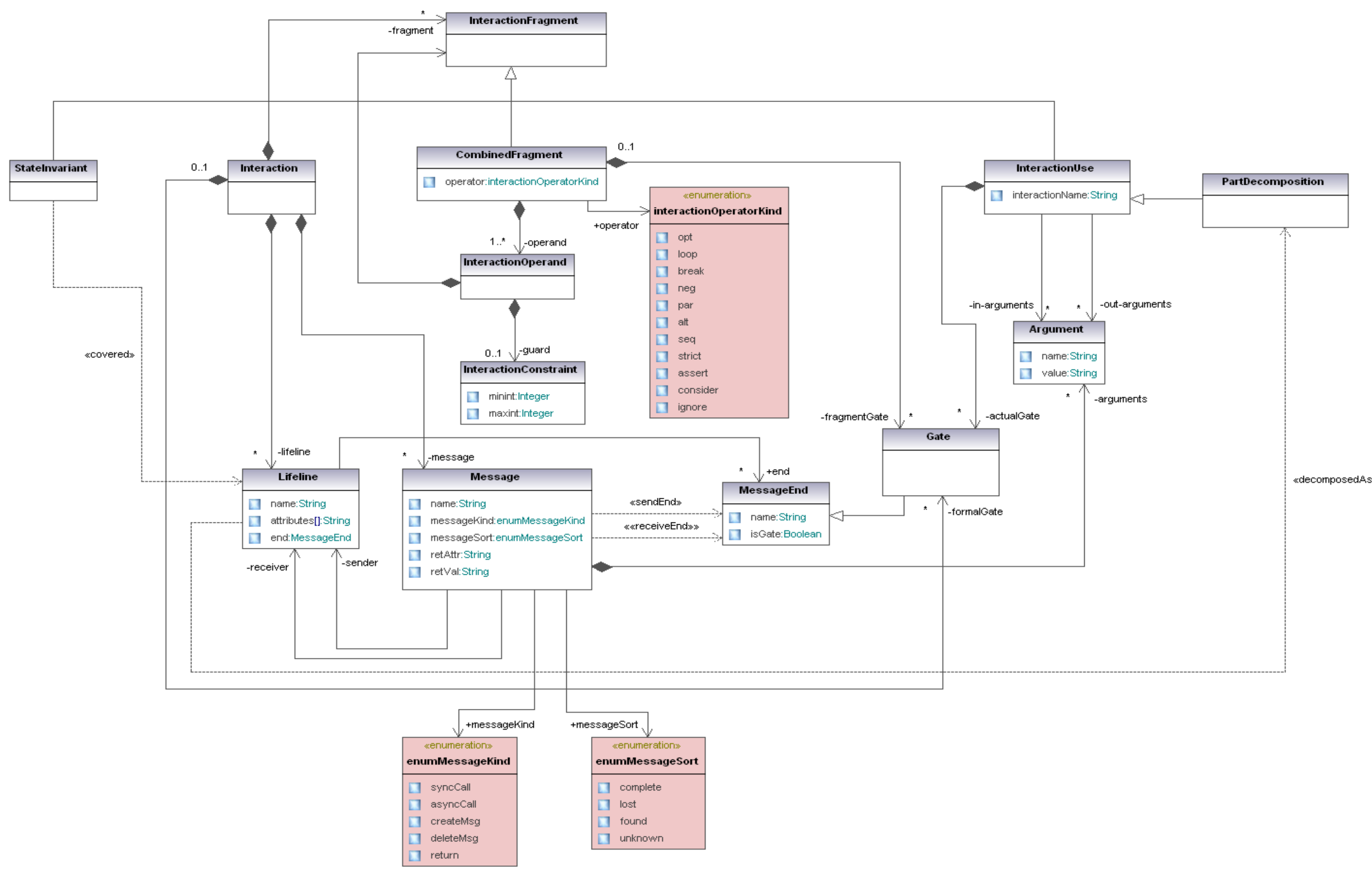


Figure 14 Subset of the UML Sequence Diagram Metamodel

4.3.2 Sequence Diagram Metamodel Extension

The main motivation for proposing an extension to the UML metamodel for sequence diagram is to make it easier to integrate with the other metamodels. Apart from this, the extended metamodel accommodates future extensions made to sequence diagram notations. By extensions we mean either integration of domain-specific information or modeling syntactic variability due to difference in comprehension. It will allow advanced UML modelers to define domain-specific extensions to the sequence diagram in a precise and usable manner. Furthermore, this modification also provides ease of mapping program code to sequence diagrams thereby providing a means of validating consistency between them.

Extensions to Sequence diagram notations and metamodel have been proposed quite a few times in the literature. The approaches are discussed in section 3.4 as part of the literature review for metamodel extensions.

The UML sequence diagram metamodel described in the previous subsections contains a meta-class called "*CombinedFragment*". The UML specification provides twelve types of combined fragments that are given by an enumerated attribute called "*InteractionOperatorKind*". The extended sequence diagram metamodel proposed in this work restructures the combined fragment logic by suggesting a change to the abstract syntax and well-formedness rules of the metamodel elements.

Initially two new meta-classes *SingleOperand* and *MultiOperand* are introduced. The motivation behind the inclusion of these meta-classes is to remove the well-formedness rule ($\mathrm{WF}_{\mathrm{SEQ}}$ *Rule 6*) enforced through constraints on the sequence diagram in the UML

specification. Based on this rule, all the sub-classes of the meta-class *SingleOperand* can have only one operand in its body. These two meta-classes are defined similar to the manner of meta-class description in the UML specification as follows:

***SingleOperand* Metaclass**

- **Description**
  - SingleOperand is an abstract meta-class, which declares a combined fragment with only one single operand in its body definition. SingleOperand is a specialization of CombinedFragment.
- **Associations**
  - InteractionOperand –the operand of the fragment

***MultipleOperand*Metaclass**

- **Description**
  - MultipleOperand is an abstract meta-class, which declares a combined fragment with more than one single operand in its body definition. MultipleOperand is a specialization of CombinedFragment.
- **Attributes**
  - isStrict – if false, the messages between different operands can be interleaved but messages within a single operand should be ordered; the default is false
- **Associations**
  - InteractionOperand –the set of operands of the fragment

The proposed metamodel modifies one class declaration from the standard metamodel. The "*CombinedFragment*" meta-class will no longer have the attribute

"*InteractionOperatorKind*". In addition, the association of this class with the "*InteractionOperand*" meta-class is also removed.

Apart from the above modifications, a single meta-class for each "*InteractionOperandKind*" was also added to the extended metamodel. These meta-classes are then made subclasses of either the *SingleOperand* or the *MultipleOperand* meta-class. The *Opt, Loop, Break* and *Neg* meta-classes are made subclasses of *SingleOperand* as they require only one operand. The remaining *Par, Alt, Assert, Strict, Seq* and *ConsiderIgnore* meta-classes are made subclasses of the *MultipleOperand* meta-class. The main motivation behind this modification is because a number of suggestions have been proposed in the literature to modify the semantics of some combined fragment operators such as "*alt*", "*neg*", "*assert*" and so on. In order for the above-mentioned proposed operators to be added as metamodel extensions, existing combined fragment operators need to be treated as model elements. Our proposed extended metamodel allows researchers to define their modifications in a usable manner by making use of lightweight extensions. In order to illustrate this, we take an example of the extension proposed by Refsdal and Stølen [355] to include probabilistic choice to the existing "*alt*" operator. They proposed an operator "*palt*" (probabilistic alternative), in which the choice between alternatives is expressed as probabilities between two or more operands. This extension is depicted in Figure 15 using a stereotype "*palt*".

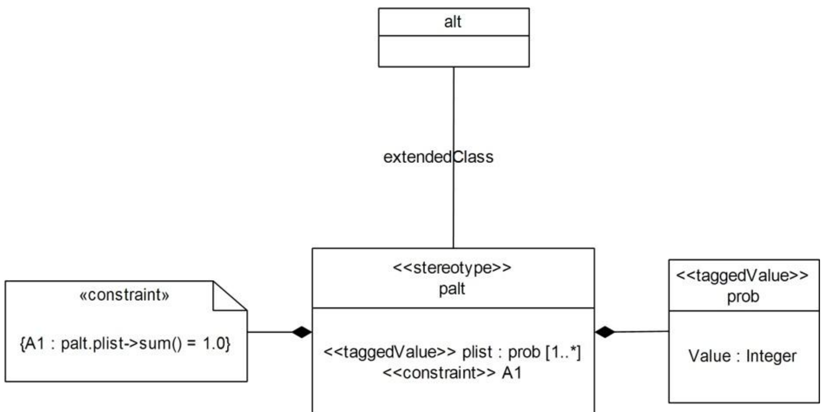


Figure 15 An example lightweight extension of "alt" fragment

Apart from deprecating the well-formedness rule $\mathrm{WF_{SEQ}}$ Rule 6, another rule $\mathrm{WF_{SEQ}}$ Rule 7 is also removed. This is because the conditions minint and maxint are included as metadata in the loop meta-class and removed from the *InteractionConstraint* meta-class. This ensures that they are valid only when the loop fragment is used. The "*Loop*" meta-class can hence be defined as

Loop Metaclass

- **Description**
  - Loop is a meta-class, which declares a combined fragment representing a loop. The single operand in the fragment body will be repeated a number of times as specified by constraint attached to it. Loop is a specialization of SingleOperand.

- **Attributes**
  - minint – the minimum number of iterations of a loop
  - maxint – the maximum number of iterations of a loop

- **Well-formedness rules**
    - If minint is specified, then the expression must evaluate to a non-negative integer.
    - If the maxint is specified, then the expression must evaluate to a positive integer.
    - If both minint and maxint are specified, the value of maxint must be greater than or equal to the value of minint.

The descriptions of all other meta-classes are left for future improvements and extensions to the UML Metamodel. The proposed extension component for the Sequence Metamodel along with its related meta-classes from the original UML Sequence Metamodel is shown in Figure 16. Figure 17 presents the complete extended sequence diagram metamodel. The extended sequence diagram metamodel along with promising applications apart from metamodel integration is provided by Misbhauddin and Alshayeb [368].

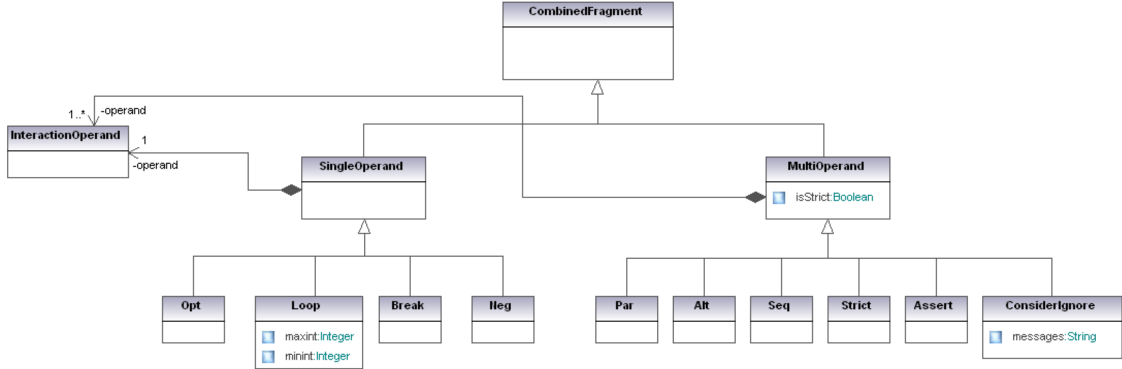


Figure 16 Extended Component of the Sequence Metamodel

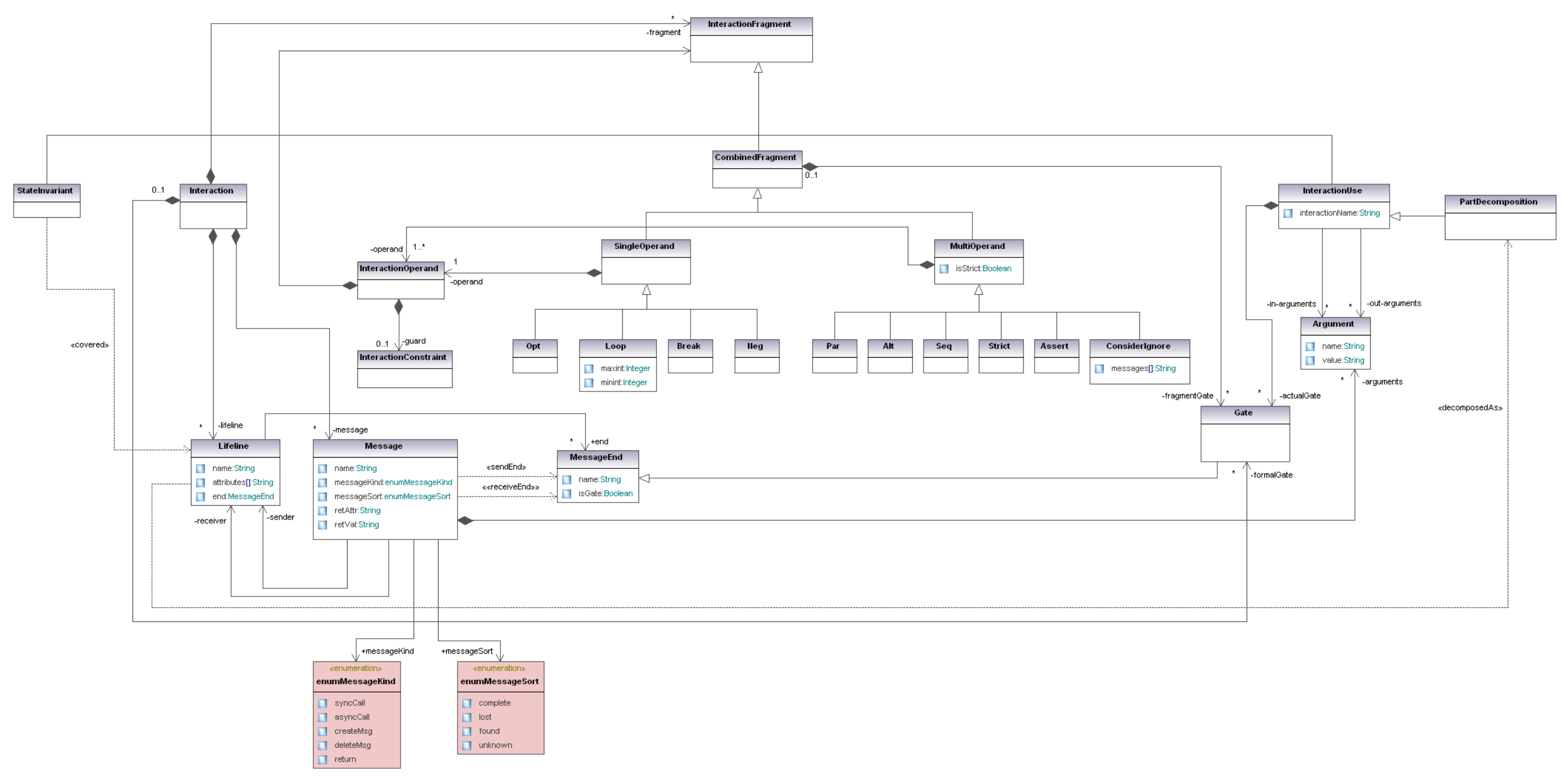


Figure 17 Extended Sequence Diagram Metamodel [368]

4.4 UML Use Case Diagram

Use case diagrams were initially introduced by Jacobson [39] and later adopted by the OMG to be part of UML. A use case diagram represents a functional view of an object-oriented system and plays a vital role in modeling the system's functional requirements. To model these requirements, the use case diagram represents them as a set of use cases. Each use case is a specification of a set of operations between the system and actors resulting in an output valuable to actors or stakeholders of the system. Formally, a use case diagram can be defined as follows:

Definition 4.3: A use case diagram is a 5-tuple $UC = \{\mathbb{U}, \mathbb{a}, \mathbb{m}, \mathbb{r}, WF_{UC}\}$ where

- $\mathbb{U}$ is a finite set of use cases
- $\mathbb{a}$ is a finite set of actors
- $\mathbb{m} \subseteq \mathbb{a} \times \mathbb{U}$ is a finite set of associations
- $\mathbb{r} \subseteq \mathbb{U} \times \mathbb{U}$ is the relationship between use cases
- WF_{UC} is a set of well-formedness rules on the Use Case Diagram UC

4.4.1 UML Use Case Diagram Metamodel

A use case model represents the functional view of an Object Oriented (OO) system and plays a vital role in modeling the system's functional requirements. The use case model represents the functional requirements as a set of use cases. Each use case is a specification of a set of operations between the system and the actors resulting in an output valuable to actors or stakeholders of the system. UML use case diagram models use cases and their relationships with actors and other use cases. Behavior of each use

case is typically documented either through other UML models (sequence [369-371] or activity diagrams [372-374]), formal modeling languages [375-378], or as natural language text.

UML models are described by a metamodel detailed out in its specification document [10]. A UML metamodel is a qualified alternate of the UML models and is a representative of any model that can be expressed with it. Since the UML metamodel includes information for all the diagrams in the modeling suite, a subset of the UML metamodel that includes all elements related to modeling a use case diagram is shown in Figure 18. A detailed description of the abstract syntax and well-formedness rules of UML use case diagrams is provided in Appendix 1.

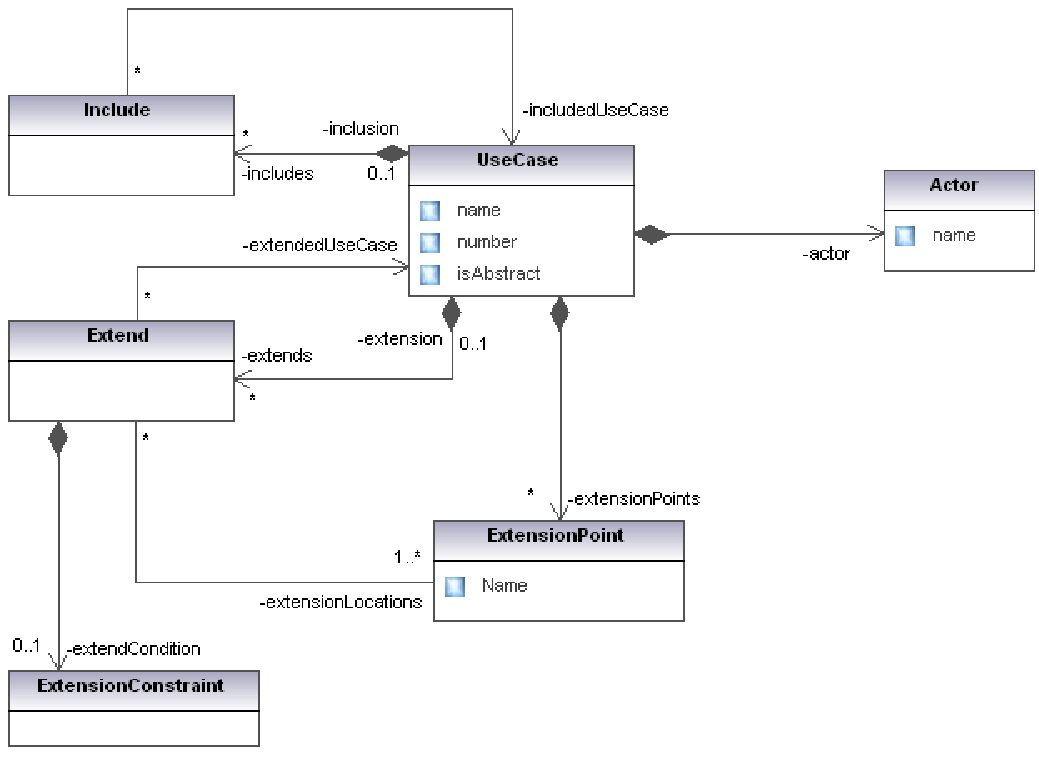


Figure 18 Subset of the UML Use case diagram metamodel

4.4.2 Use Case Diagram Metamodel Extension

The use case model that is part of the UML specification describes only its structural view. The structural view defines the services provided by the system without divulging its internal structure. The internal structure presents the behavioral aspect of the use case. A use case, once initiated by an actor, performs a number of operations to provide a meaningful output to the invoking actor. These set of operations constitutes a use case's behavior. There are a number of ways in which the behavioral information can be presented. A classification of these approaches is given in Figure 19 below:

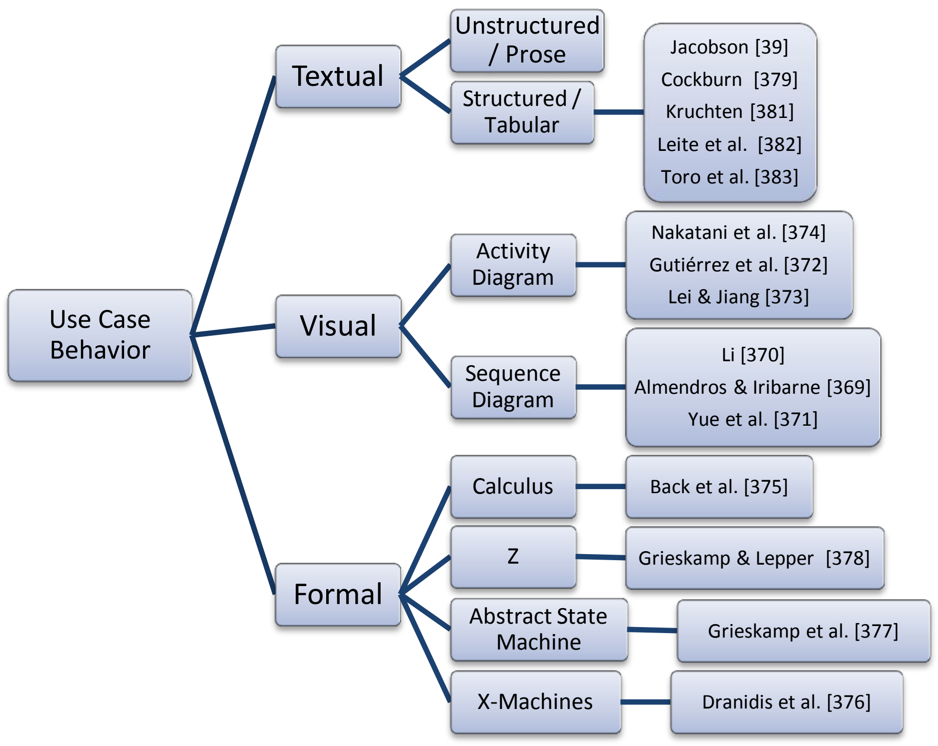


Figure 19 Use Case Behavior Description Approaches

Which of these approaches to use depends on the nature of the use case behavior as well as the intended reader? It is suggested by Cockburn [379], Kulak and Guiney [380] and many other practitioners that non-technical stakeholders usually understand use case behavior written in the vocabulary of the problem domain better than any other notation. Hence the text-based approaches gained immense popularity. Two major advantages available by selecting the text-based approach for behavioral specification are:

1. Understandable by both technical and non-technical stakeholders.
2. Minimum use of UML vocabulary.

One major trade-off when selecting textual specifications to model use case behavior is that they are prone to mistakes and incompleteness. Although using formal models and other UML diagrammatic notations for requirements elicitation and use case description allows for better structure and validation, it also introduces a high participation hurdle for customer involvement which is the main goal for use case specification. In order to circumvent the issues posed when using the text-based approach, we propose a metamodel that extracts useful information from the text and maps it to the metamodel elements for further analysis. A number of different notations or templates for composing them have been proposed in the literature. Table 2 shows a number of prevalent initiatives that describe a use case template descriptions in the form of a structured template.

Table 2 Template elements from different notation proposed in the literature

Template Elements	Cockburn [379]	Jacobson [39]	RUP [381]	Leite [382]	Toro [383]
Name: *Unique name assigned to a use case*	√	√	√	√	√
Number: *Unique ID assigned to a use case*	√			√	
Goal: *Statement of goals expected from the use case*	√				
Scope: *System being considered black-box under design*	√				
Level: *Level of use case description*	√				
Description: *Brief summary of use case purpose*		√	√		
Primary Actor: *Actor that initiates the use case*	√				√
Secondary/Supporting Actors: *Actors that participate within the use case*	√				
Offstage Actors: *Non-interacting actors concerned with the outcome of the use case*					
Special Requirements: *List of non-functional requirements*		√	√		
Preconditions: *Expected state of the system prior to use case execution*	√	√	√	√	√
Post-conditions (Success): *State of the system upon successful completion of the use case*	√	√	√		√
Post-conditions (Failure): *State of the system if goal is abandoned*	√				
Performance Target: *The amount of time this use case should take*	√				
Priority: *How critical to the system / organization is the use case*	√				
Frequency: *How often is it expected to happen*	√				
Open Issues: *List of issues about this use case awaiting decisions*	√				√
Due Date: *Date or release of deployment*	√				
Main Flow: *Steps of the scenario from trigger to goal delivery*	√	√	√	√	√
Sub Flows: *Sub-variations that will cause eventual bifurcation in the flow*	√	√		√	
Alternate Flows: *Conditional variations that will cause eventual bifurcation in the flow*	√	√		√	
Extension Points: *List of extensions each referring to a step in the main flow*		√	√		
Exceptions: *Conditional variations that will cause unsuccessful termination of use case flow*				√	√
Super Use Case: *Name of use case that this one specializes*	√				
Sub Use Case: *Links to all use cases that specialize this use case*	√				

As observed from Table 2, a number of variations exist in the elements for use case description template. Despite these differences, each approach has two major parts of information: description and dynamics depicted in Table 2 separated by a thick line. The description part includes elements such as name, number, goal, scope, level, description, actors (primary and secondary), preconditions, post-conditions (success and failure), priority, frequency, open issues, due date and special requirements. The dynamics part captures the use case's flow of execution. Flow of execution of a use case includes a sequence of steps that can either be events (messages exchanged between actors and use case objects), or anchors (that disrupt the main flow by allowing access to sub flows, alternate flows, use case extensions and inclusions).

The main objectives in proposing an extension to the use case metamodel can be summarized as follows:

1. The original metamodel is an essential subset of the extended metamodel so that information can be utilized from both depending upon the requirement of the user.
2. The extended metamodel should take into consideration information from all published templates. But information that is useful for further analysis of the use case model should be included as meta-classes so other tools can access and extend it easily and other information can be included as meta-attributes of the respective meta-classes.
3. Information for use case analysis, model evaluation, and model interchange should be readily available and accessible from the metamodel.

4. The extended metamodel should provide an integrated global modeling environment for tools and users and provide seamless transition from requirements to system modeling.

For the sake of clarity of presentation, we construct the metamodel in pieces. A complete metamodel is presented towards the end of this section. Each modeling element from the use case diagram is analyzed and extended.

4.4.2. (a) Actors

Actors are used in the use case diagram to model users of the system. The UML Specification defines actors as entities that can communicate with several use cases. In this proposed extension to the use case metamodel, we classified actors based on two criterions: the role they play in a use case and the role they play in the system. Many authors define different types of actors based on their role in the use case. According to Larman [384], an actor can be classified into three types:

1. **Primary Actor:** An actor that initiates the use case and helps realize its goal.
2. **Supporting Actor:** An actor that participates in a use case that helps realize a primary actor's goal.
3. **Offstage Actor:** An actor that does not interact with the system but has needs that should be addressed in the system. Offstage actors are considered as stakeholders of the system under development.

The actor's type may differ from use case to use case. Based on the above classification, we added three associations between the *UseCase* meta-class and the *Actor* meta-class to denote the role an actor plays in a use case. Popularity of the use of use case modeling as a de facto standard for requirement modeling in the field of software engineering was further enhanced with the establishment of a software estimation technique known as Use Case Points (UCP) [385]. UCP became a good candidate for early estimation of software size and effort because of its simplicity and ease of use. The main activity of UCP is to estimate the complexity of actors and use cases. The complexity of actors is identified based on the role an actor plays in the system (as opposed to in a use case as discussed above).

In order to incorporate this information in our extended metamodel, we categorized actors based on information from both the original UCP model presented by Schneider and Winters [385] and the enhanced model known as iUCP presented by Nunes [386]. Based on this, we classified the actors into the following categories:

1. **System Actor:** This type of actor is another system interacting with the base system through an application programming interface (API). For example, the ATM system reads the credit card information directly from a credit card reader. In this case, the credit card reader is outside the system and accessed through an API; therefore, the credit card reader is a system actor.
2. **Network System Actor:** This type of actor is another system interacting with the base system through a protocol or data store. For example, the ATM system verifies the credit card information from an accounting system. In this case, the accounting

system is outside the system and accessed through a network. Therefore the accounting system is a network system actor.

3. **Human Actor:** This type of actor is a person or a user who will use the system. It is the most common type of actor. For example in the ATM system, a customer will ask the system to perform a transaction and therefore, the customer is a human actor.

The iUCP model differs from the original UCP model as it is based on the usage-centered design method [387] in contrast to the conventional use case model for classifying actors. The main reason behind this is because of the richness of the information conveyed by the usage-centered method regarding the complexity underlying each actor. Human actors are divided into simple, average and complex based on the number of roles they play in the system. In the usage-centered design method, the concept of actor is expanded through user roles that represent the relationship between users and a system. A user role is characterized by the "*context in which it's performed, the characteristic manner in which it's performed, and the design criteria for the role's supporting performance*" [386]. The number of roles supported by each human actor provides an important way to infer the complexity associated with each actor. In order to incorporate this, we added a meta-attribute called *num_roles* to the *Actor* meta-class. Since this attribute is associated with human actors only, a default value of 1 is used for system and network system actors.

Actors in a use case model can be associated to each other using the generalization relationship. It is the only kind of relationship that exists between actors. The actor modeling the common role is referred to as the parent actor and the actors using the

common role are called the child actors. In simple terms, a child actor inherits the capability to communicate with the use cases its parent actor is associated with. The metamodel representation with the modified *Actor* meta-class and its relationship with the *UseCase* meta-class are presented in Figure 20.

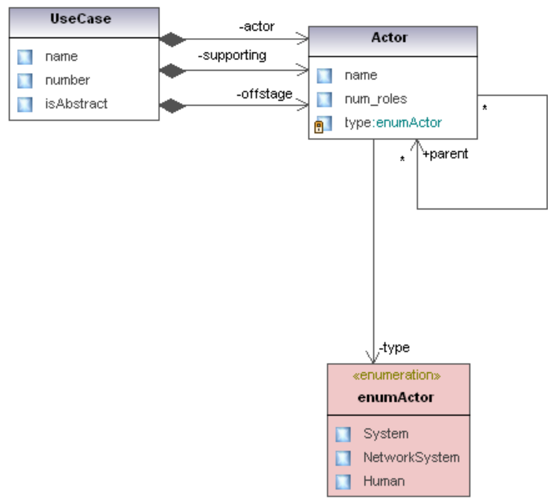


Figure 20 Addition to the extended UML metamodel for Actor

4.4.2. (b) Use Case

A use case within a use case model consists of number of information elements as shown in Table 2. Despite the difference of information portrayed by different templates, each template has two major parts of information: the description part and the dynamics part. In this section, we discuss the description part of a use case.

Information within the use case description can be classified into two categories; information that is used for "mere" documentation purpose and information that will be used for use case analysis at later stages of software development. Keeping in lieu with

the above mentioned criteria, we decided to separate these elements and depict them independently in the enhanced metamodel as follows:

1. Use case description elements that will be used for its documentation will be represented as meta-attributes in a separate meta-class called Description. (See Figure 21)
2. Use case description elements that will be used for analysis will be represented as separate meta-classes and elaborated and justified later in this section.

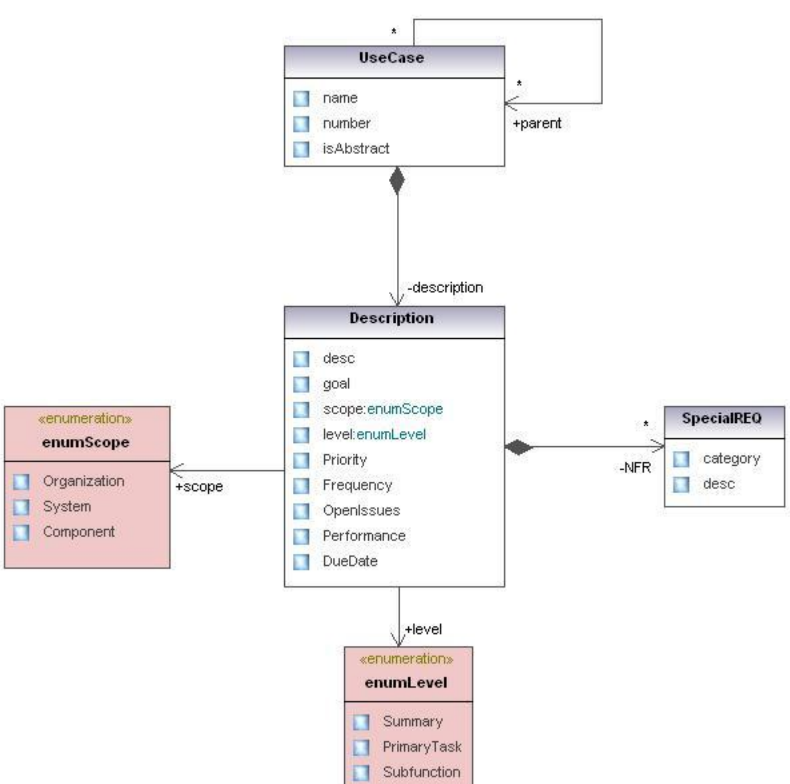


Figure 21 Addition to the extended UML metamodel for Use Case

4.4.2. (c) Use Case Relationships

UML defines three types of relationships between use cases: ‹‹include››, ‹‹extends›› and generalization. When describing these relationships through a metamodel, we need to discuss the relationship depiction on the use case structural view and within the use case flow of execution (its behavioral view). In this section, we discuss the impact of use case relationships on metamodel elements that depict the use case's structural view. We provide a coherent description of these relationships derived from the literature and extend the use case metamodel based on these descriptions. The manner in which these relationships are depicted in the use case's flow of execution are discussed later.

- **Include Relationship**

  Two use cases are related by the ‹‹include›› relationship if one use case (known as the base use case) uses the functionality offered by the other use case (known as the included use case). Two main reasons for using the ‹‹include›› relationship in a use case model according to the UML specification are: to fragment Complex Use Case into manageable ones [384, 385] and to reuse use Cases [384, 385, 388-392]. Apart from this, some authors recommend the use of ‹‹include›› relationship for conditional behavior [384, 389, 391] and to handle asynchronous events [384]. The main motivation behind the use of ‹‹include›› relationship for conditional behavior by the above-mentioned authors is that this relationship is much easier for most people to understand and use than other relationships such as ‹‹extends›› and generalization. Also the use of ‹‹extends›› is restricted to cases where the base use case is locked or "closed for modification". Since it is difficult to gauge when a use case is closed for modification, we adopted the semantics of the ‹‹include››

relationship as outlined in the UML specification and accepted by majority of the authors [393] and leave the concept of conditional behavior to the ‹‹extends›› relationship. We do not modify the meta-classes related to the ‹‹include›› relationship in the extended metamodel.

- **Extend Relationship**

  Two use cases are related by the ‹‹extends›› relationship if one use case (known as the base use case) implicitly incorporates the behavior of another use case (known as extension use case) at a specified location. The extension use case is executed only when some particular condition is satisfied in the base use case. There have been many reasons proposed in the literature for the use of the ‹‹extends›› relationship in the use case model. These can be summarized as follows:

  1. Optional or Exceptional Behavior: Behavior that is optional to the base use case can be separated and defined in an extending use case. Most authors agree with this usage of the extend relationship [1, 385, 388-391].
  2. Asynchronous Events: An asynchronous event is one that can be called at any point in the base use case. Use of the extend relationship to describe asynchronous events is supported by Constantine and Lockwood [1] and Cockburn [379].
  3. Defer Behavior Implementation: Armour and Miller [389] suggested the use of extend relationship to separate behavior from the base use case that can be developed later in order to assign it a lower priority.

The semantics of the ‹‹extends›› relationship has created a lot of disagreement among authors. In this section, we attempted to resolve these concerns by extending the metamodel to incorporate necessary information in order to ensure consistency in semantics of this relationship.

Since the extend relationship is optional and controlled by an execution condition, it requires the specification of the following elements:

- Extension Point: The point in the behavior of the base use case where an extended use case can be inserted is known as the extension point.
- Extension Constraint: This is an optional constraint that specifies the condition that must be true for the extension use case to be invoked from the base use case.

When the extension point in base use case scenario is reached, the extension constraint is evaluated and control is switched to the extension use case. After the execution of the extension use case, the control is resumed just after the extension point in the base use case scenario [39, 379, 389]. But in order to use the extend relationship to model exceptional behavior, the control should be allowed to return to any point in the base use case flow or be allowed to end the use case resulting in a failure or alternative success scenario. In order to handle these situations, Metz et al. [359] defined five types of alternative sequences. These are summarized in Table 3 below.

Table 3 Summary of Alternative Scenarios

Alternative History: The control in this type of alternative sequence never returns to the base use case scenario. The success post-condition in this case can either be the overall success post condition of the base use case or its subset.	Success
Alternative Insertion: The control in this type of alternative sequence returns to the point just after the extension point in the base use case.	Success
Use Case Exception: The control in this type of alternative sequence never returns back to the base use case scenario. In contrast to alternative history, the use case exception is always a failure scenario and results in a failure post condition.	Success Success
Alternative Fragment: The control in this type of alternative sequence returns to any point after the extension point in the base use case.	Success
Alternative Cycle: The control in this type of alternative sequence returns to any point before the extension point in the base use case.	Success

In order to accommodate sequences mentioned in Table 3, the concept of rejoin point was proposed [359, 360]. A rejoin point allows the control to return to separate point in the main flow after performing the steps specified in the extension use case. We followed a similar approach in our extension of the use case metamodel and added a meta-class called *RejoinPoint*. When the rejoin point is equal to the extension point it leads to an alternative insertion fragment. When the rejoin point is a point that occurs either before or after the extension point, then the alternate scenario leads to an alternative cycle or alternative fragment respectively. Finally when the rejoin point is not specified, it leads to a use case exception.

In order to complete our extension to the Use Case metamodel for ‹‹extends›› relationship, we considered an interesting premise put forward by Laguna and Marqués [394]. An extension point in the base use case can be extended by several use cases. An issue arises when this extension point is reached and a decision is to be made if whether only one or at least one among these extension use cases are to be selected. In order to complete and clarify the behavior of the base use case and to aid in the process of elicitation of requirements, Laguna and Marqués [394] added multiplicity attributes to the extension point meta-class. Following their approach, we added the lower and upper meta-attributes to the *ExtensionPoint* meta-class to clarify the behavior of extend relationship in case of multiple use case extensions. A multiplicity of 0..1 states that the extension use case can be executed when the constraint is true (equivalent to the original UML extend semantics), a multiplicity of 1..1 states that only one of the possible

extension use case can be selected and finally a multiplicity of 1..* allows more than one use case to be inserted.

In addition, following Constantine and Lockwood [1] in our metamodel extension, we have considered the concept of asynchronous extensions in which an extension use case can be called asynchronously at any step of the use case flow. Asynchronous extensions are defined in our metamodel as a separate meta-class called *AsyncExtend*. It is defined separately as it lacks an extension point and extension location. For example, a customer can press cancel at any time during his usage of the ATM Machine. Figure 22 shows the extended metamodel for ‹‹extends›› relationship.

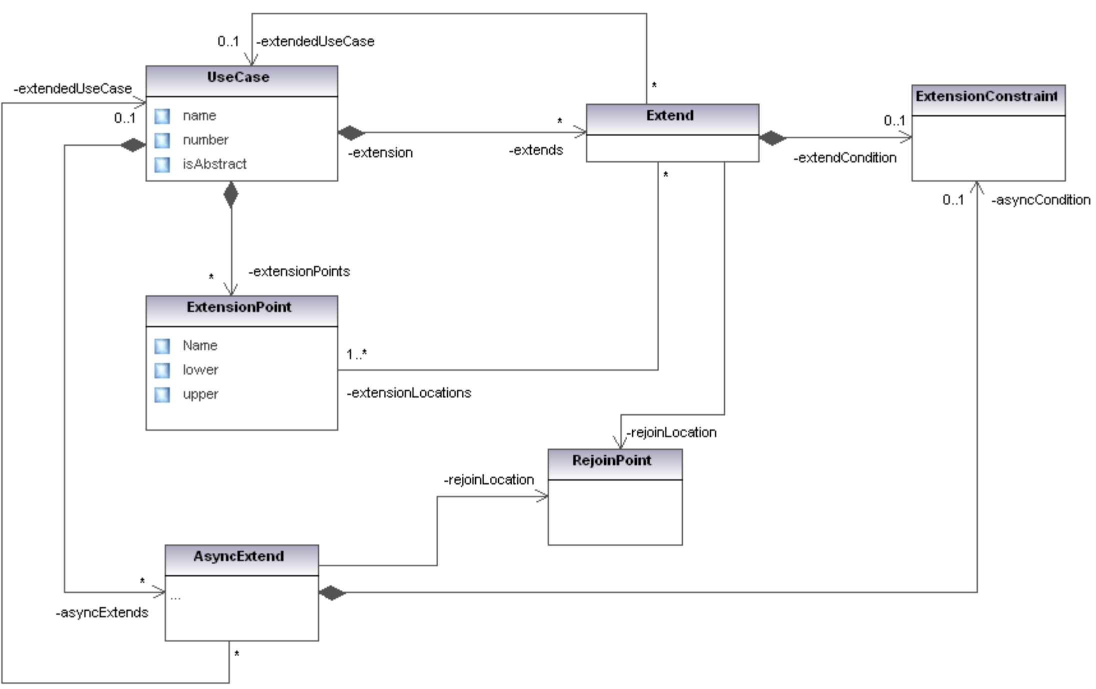


Figure 22 Addition to the extended UML metamodel for extend relationship

- **Generalization Relationship**

  The generalization relationship in a use case model allows a given use case to be defined as a specialized form of an existing use case. Common behaviors, constraints and assumptions are factored out into a general use case (also known as the parent use case) which can then be inherited by a specialized use case (also known as the child use case). The concept of generalization and specialization gives rise to two types of use cases:

  - Abstract Use Case: An abstract use case is an incomplete use case that can only be invoked by another use case. An actor cannot directly invoke it. Jacobson refers to the generalized use case as an abstract use case.
  - Concrete Use Case: A concrete use case is a self-contained complete use case one that can be directly invoked by an actor. A concrete use case provides an implementation to an abstract use case. Jacobson refers to the specialized use case as a concrete use case.

Most authors agree with the definition and usage of the generalization relationship. Figure 21 depicts the use case metamodel for generalization. Although the structural representation of this relationship is straightforward, its usage within a use case scenario description is vaguely described in the literature. A metamodel for generalization within a use case description is discussed in this section.

4.4.2. (d) Use Case Flows

From the many forms of composing the dynamics part of the use case specification, Bittner and Spence [391] provided the most promising one. They expressed the use case dynamics through a sequence of steps. These steps are grouped to form behavioral fragments called flows. A single use case consists of multiple flows as shown in Figure 23, but the flow of events that is initiated when the use case is executed by an actor is called the main flow. Apart from the main flow, a use case can also have multiple sub flows and alternate flows. These flows are initiated from the main flow. A sub flow is used either to describe complex logic associated with a particular step or to factor out redundant steps described in a flow. Alternate flows include behavior that is alternate to the use case. This could be optional or exceptional behavior Steps within a flow are usually atomic events, the content interpretation of which will be discussed later. Usually unconstrained natural language is used to describe the steps within a flow.

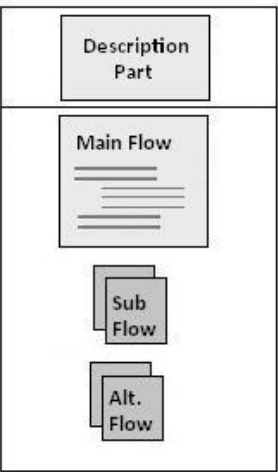


Figure 23 Structure of a typical text based use case description

Following the flow composition architecture described in Figure 23, we initially added a meta-class called Flow to the extended use case metamodel. Different types of flows are then represented as specialized meta-classes of the Flow meta-class: *MainFlow*, *SubFlow* and *AlternateFlow*. Apart from terminological differences and elements used, there are some noteworthy semantic differences between the templates mentioned earlier in Table 2. In order to ensure deterministic initiation of use cases and their completeness, we describe the semantics that our extended metamodel is built upon as follows:

1. Restrict the number of main flows to only one (as described by Cockburn [379] and opposed to Jacobson's [39] notation that allows multiple main flows).
2. Allow sub flows and alternate flows within sub flows and alternate flows.
3. Allow multiple extension points (as described by Jacobson's [39] notation and opposed to Cockburn's [379] notation that does not allow extensions at all).

In order to allow sub flows and alternate flows to have sub flows and alternate flows within them, we added another level of inheritance between the Flow meta-class and SubFlow and AlternateFlow. This intermediate meta-class is called OtherFlow. Most authors define use case flow as a composition of a sequence of steps [363, 364, 379]. Since one of our main goals for extending the use case metamodel is to use the instantiated use case model for analysis, we used the concept of transactions. Our main motivation in the use of transactions to describe flows is because transactions are mainly used as a complexity metric within the use case point method. A transaction is a shortest sequence of use case steps starting from an actor's request and ending in a system response [395]. Hence, a use case flow is composed of a number of ordered transactions included in the metamodel by the Transaction meta-class. Each transaction is then

composed of a sequence of steps modeled by the Step meta-class. Figure 24 shows the excerpt of the extended metamodel for the use case flow of events.

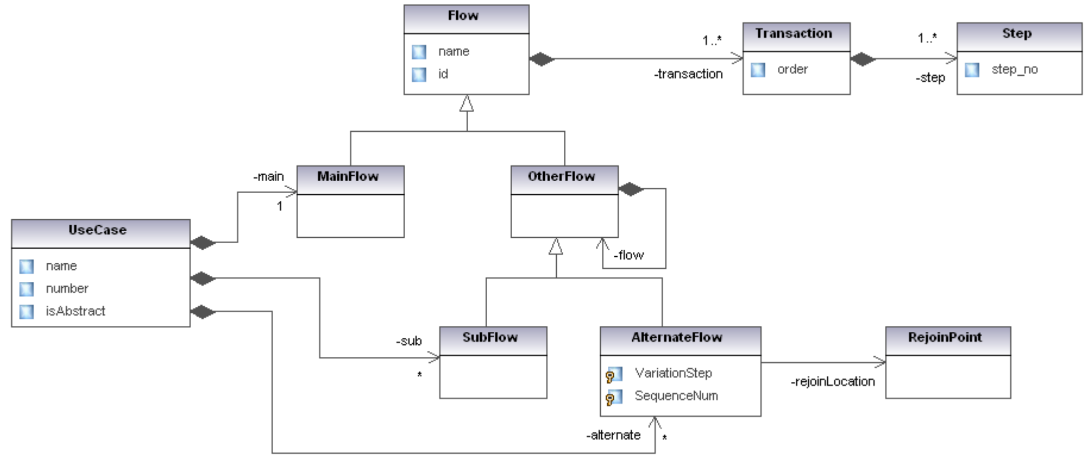


Figure 24 Excerpt of the Extended Metamodel for the Use Case Flow of Events

- **Use Case Action Steps**

  In a flow description, a step can be classified as either an action step or a branching step. A step that performs a certain action (from the actor to the system or vice versa) is referred to as an event. A branching step is a step that alters the sequential order of the flow by invoking the behavior of another flow of events. Branching steps are discussed in the next subsection. Natural language sentences are used to describe an event. A number of approaches that make use of the grammatical structure of the natural language and natural language processing (NLP) techniques, to analyze and extract relevant information, have been proposed in the literature [396-404]. As far as the metamodel is concerned, we focused on the elements that make up a typical event sentence. An event allows a

sender to communicate with one or more receivers through a message (action) that may or may not include additional parameters (arguments). Hence, it is safe to assume that an event is composed of a sender, multiple receivers, an action and zero or more arguments.

Since a step can either be an event or a branching action, it is specialized by two meta-classes called *Event* and *Anchor*. The *Event* meta-class is further extended to include *Sender, Receiver, Action* and *Argument* meta-classes based on the above mentioned reasons. In addition, following Diev's transaction definition [395] and the transaction model proposed by Ochodek and Nawrocki [405], we enumerated four types of actions relevant from the use case transaction point of view. This is shown through an enumerated meta-attribute called *actionType* in the *Action* meta-class. Excerpt of the metamodel depicting the meta-classes relevant to a use case step is shown in Figure 25.

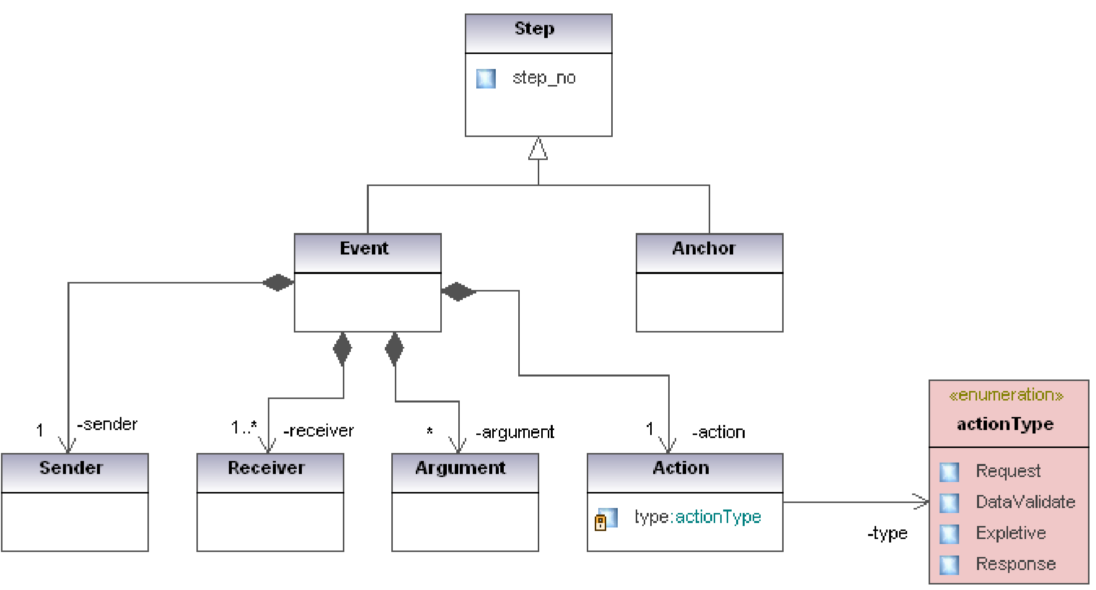


Figure 25 Excerpt of the Extended Metamodel for the Use Case Flow Steps

- **Use Case Branching Steps**

As mentioned earlier, a step can either be an event or a branching action. We refer to the branching action step as anchors as they are mere placeholders or locations within the main flow that invoke another flow or use case. The natural order in which steps occur within a flow is sequential from top to bottom. This concept of sequential ordering can be altered by including the behavior of another flow in the main flow. A flow may include another flow in its execution. This insertion can either be conditional or unconditional. Unconditional insertions of a flow are referred to as Inclusion. A flow may include another flow which is part of the same use case description (also known as sub flows) or may include a flow defined in another use case description (i.e. use cases related to each other by the UML include relationship). These two inclusions are referred to as Internal Inclusion and External Inclusion respectively. An internal inclusion anchor specifies the name of a sub-flow (bolded out to differentiate) [39] whereas an external inclusion anchor is composed of the keyword include followed by the name of the use case to be included [39, 379].

Use case descriptions, apart from allowing unconditional insertions, also provide a means of including another flow based on a condition. Conditional insertions of a flow are referred to as a Variation. Similar to that of Inclusion, a flow may include a variation flow part of the same use case description (also known as alternate flows) or may include a flow defined in another use case description (i.e. use cases related to each other by the UML extends relationship). These two

variations are referred to as Internal Variation and External Variation respectively.

Internal Variation anchors usually do not include branching information. Information about an alternative flow is specified in the alternative flow itself. An example of an internal variation scenario is shown in Figure 26.

Main Flow

1. -----------
2. -----------
3. The Customer enters the withdrawal amount.
4. -----------

Alternative Flow

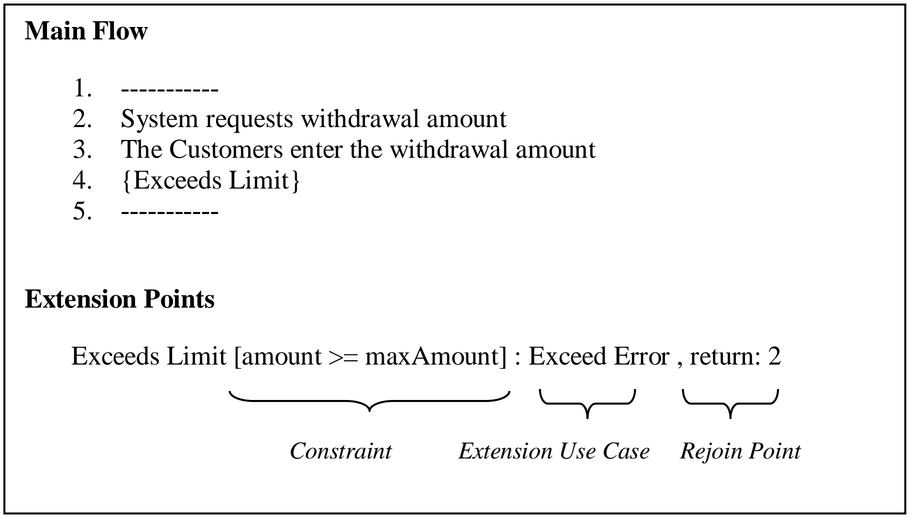
3 (a) ATM System has no currency

1. The system notifies the customer that the ATM is out of cash.
2. The use case ends.

Figure 26 UC Description example depicting the use of Alternative Flow

Based on the example illustrated above, we modified the *AlternativeFlow* meta-class shown in Figure 24 with the following meta-attributes: *VariationStep* and *SequenceNum* (for cases when a single step in the main flow can result in multiple alternative flows). Since the internal variation is a conditional branch, a constraint element needs to be added to the extended metamodel. All discussions related to constraints are deferred towards the end of this section. In addition, since the alternation scenarios depicted in Table 3 are applicable to alternative flows, an association is added between the *AlternativeFlow* meta-class and the *RejoinPoint* meta-class.

An external variation anchor specifies the name of the extension point. Information regarding the extension use case to invoke, condition and location is included in the extension point. An example of the use of an extension point and its description is shown in Figure 27.



Main Flow

1. -----------
2. System requests withdrawal amount
3. The Customers enter the withdrawal amount
4. {Exceeds Limit}
5. -----------

Extension Points

Exceeds Limit [amount >= maxAmount] : Exceed Error , return: 2

Constraint *Extension Use Case* *Rejoin Point*

Figure 27 UC Description example depicting the use of Extension Points

Figure 28 illustrates how the concepts mentioned above can be included as specialized meta-classes of the *Anchor* meta-class mentioned in Figure 25.

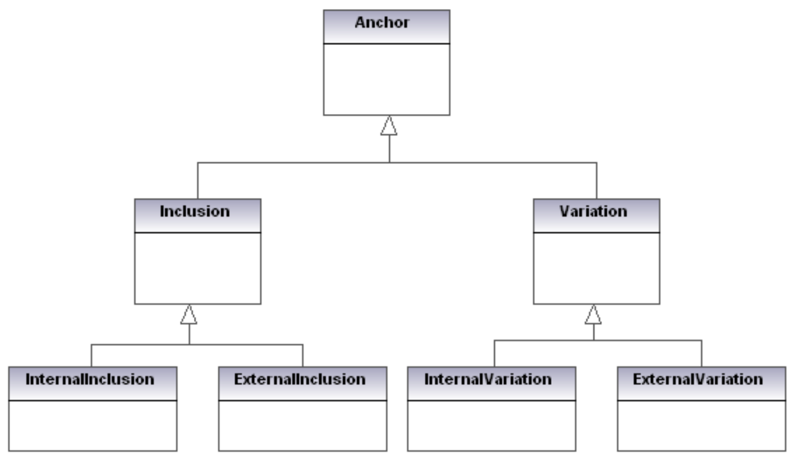


Figure 28 Metamodel for the Anchor meta-class mentioned in Figure 25

- **Use Case Generalization**

One area when describing textual use case metamodels that has been given least attention is how a specialized flow of a child use case is specified. Hoffmann et al. [361] were the first to discuss generalization within use case flow. They introduced the concepts called general narrative description and specialized narrative description to differentiate between original use case flow and inherited use case flow. Although the formalization provided by them has its own merits, inheriting all elements of the general narrative description within the specialized description causes redundancy and makes the behavioral model difficult to maintain. The only other work to discuss generalization in use case flow was carried out by Repond et al. [364]. In their work, a generalized use case is required to define points (called Generalization Points) where the specialized use cases can add additional behavior. Two main problems with their approach are:

- A specialized use case can only add additional behavior but cannot modify or replace the steps of the generalized use case.
- The use of "*Generalization Point*" within the generalized use case defeats the purpose of allowing the generalized use case not to care about what specialization use cases exist.

In this section, we clarify the semantics of use case generalization and provide an extension to the use case metamodel. We used the terms parent use case to refer to the generalized use case and child use case to refer to the specialized use case. The two main functions of the child use case when inheriting from a parent use

case are: modifying existing behavior and adding new behavior. The child use case replaces a portion of actions, conditions and rules of the parent use case. The steps to be replaced are rewritten; steps not rewritten are executed as in parent use case. Apart from this, new actions, conditions and rules can be added, thus enhancing the behavior of the child use case. Since the flow description of a child use case will be either adding new behavior or inheriting existing behavior from the parent use case, we included it as a separate meta-class called *ChildFlow* inheriting from the *Flow* meta-class. Since the use case can either have a *MainFlow* or a *ChildFlow* depending on whether it is a parent use case or child use case, we modified the multiplicities on these two associations in the metamodel to 0..1 instead of 1.

Steps in the child use case flow can be defined locally (added behavior) which is handled by association between the super meta-class *Flow* and *Transaction* in the metamodel. Inherited behavior can either be modified or executed and used as-is. Similar to the manner we handled Alternative Flow in describing use case branching steps; we define a new meta-class *redefinedStep*. This meta-class has a meta-attribute *inheritedStep*, which references the step number inherited from the parent use case. Hence, a child flow is composed of regular steps and redefined steps. A redefined step can be rewritten; hence, we add a relationship between the *inheritedStep* meta-class and the *Step* meta-class to facilitate this information. A modified version of the use case metamodel extension depicted in Figure 25 that handles use case flow generalization is shown in Figure 29.

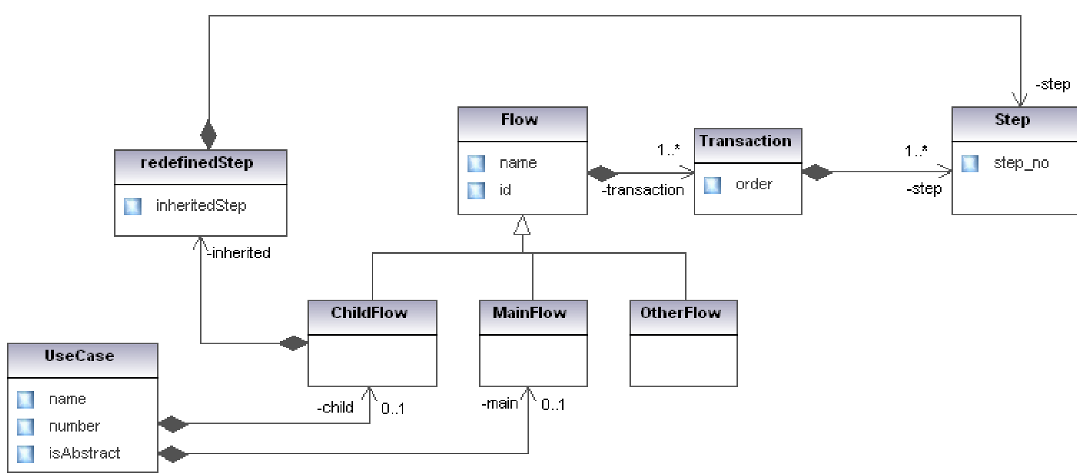


Figure 29 Excerpt of the Extended Metamodel for UC Flow with Generalization

Figure 30 shows exemplarily how the main flow of use case *Reservation* is redefined in the child use case *Reserve Conference*. We used the keyword "*super*" to differentiate between a regular step and inherited step within the child flow description. Hence, our proposed extension not only allows reusability of actions that do not require rewriting, it also allows child use case to modify actions inherited from parent use case flow.

Use Case: Reservation **Main Flow**	**Use Case: Reserve Conference** **Child Flow**
1. The system displays a list of options available for reservation. 2. The customer selects an option. 3. The system displays the total cost. 4. The system displays the reservation confirmation number. 5. The use case ends	1. super: 1 2. super: 2 a. The customer selects to reserve a conference room. 3. The customer selects the room size, duration and additional equipment required 4. The system computes the cost. 5. super: 3 6. super: 4 7. super: 5

Figure 30 UC Flow Generalization example

4.4.2. (e) Use Case Constraints

A use case model is composed of a number of constraints related to different model elements. We briefly describe these constraints prior to defining the metamodel extension. Constraints within a use case model include:

1. **Precondition:** Preconditions indicate circumstances that must be true prior to the execution of the use case behavior. A precondition on a use case explains the state the system must be in for the use case to begin.
2. **Post-condition:** A Post-condition indicate circumstances that must be true after execution of the use case behavior. A post-condition on a use case explains the state the system will be at the end of its execution. Based on the concept of alternate scenarios presented in Table 3, a use case can result in one of many states depending on the execution path (scenario) followed. Hence, a use case can have a single successful post-condition and multiple failure or alternate post-conditions. This concept is explained appropriately by the illustration in Figure 31 adopted from [1].
3. **Extension/Alternate Flow Constraint:** Execution of use case alternate flows or extension use cases require a condition to be satisfied. This condition is referred to as a flow constraint.

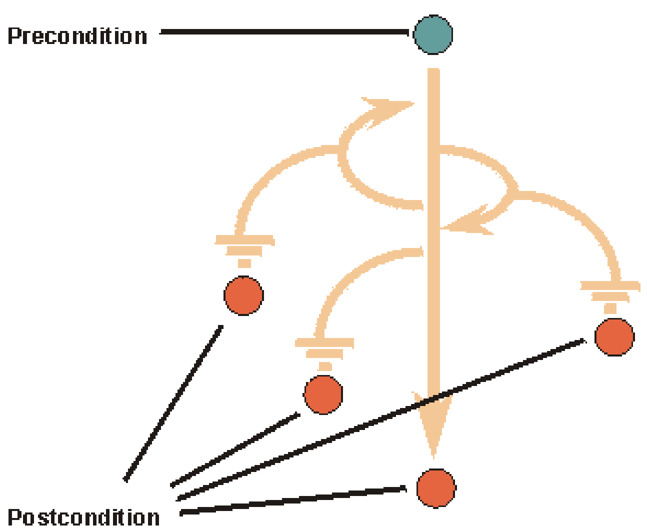


Figure 31 Multiple Use Case Scenarios adapted from [1]

All approaches that provide extensions to use case metamodel make use of a single meta-class called Constraint to handle use case constraints. Recent advancements in the field of use case modeling prompted the necessity of a structured storage and representation mechanism for constraints. Two main research proposals that make use of the use case constraint structure are: (1) Inferring use case sequencing relations from preconditions and post-conditions for requirements verification [406], use case synchronization [407] and test scenario generation [408]; (2) Enhancing software effort estimation process by assigning weights to preconditions, post-conditions and exceptions [409].

Prior to describing the use case metamodel extension with use case constraints, we included a meta-class in the metamodel called *Entity*. An entity, what most use case modeling tools refer to as Vocabulary or Glossary, refers to the systems under consideration, use cases, actors of the system and their attributes. For instance, Customer and Transaction are entities of an ATM System use case model.

A use case constraint can be either atomic or compound. A compound constraint is composed of multiple atomic constraints constructed using Boolean operators (and, or and not). An atomic constraint is a 3-tuple <E, R, V> where E is the entity, R is the relational operator and V is the value. Values assigned to entities of the system can be either units such as "*logged in*" or numeric. For instance a use case precondition "System is Active" can be written as <System, =, Active>. In order to incorporate this structure in the use case metamodel, we add the following meta-classes: *Constraint, Atomic, Compound, Value, Relation, Numeric* and *Unit*. Figure 32 shows the excerpt of the use case constraint metamodel.

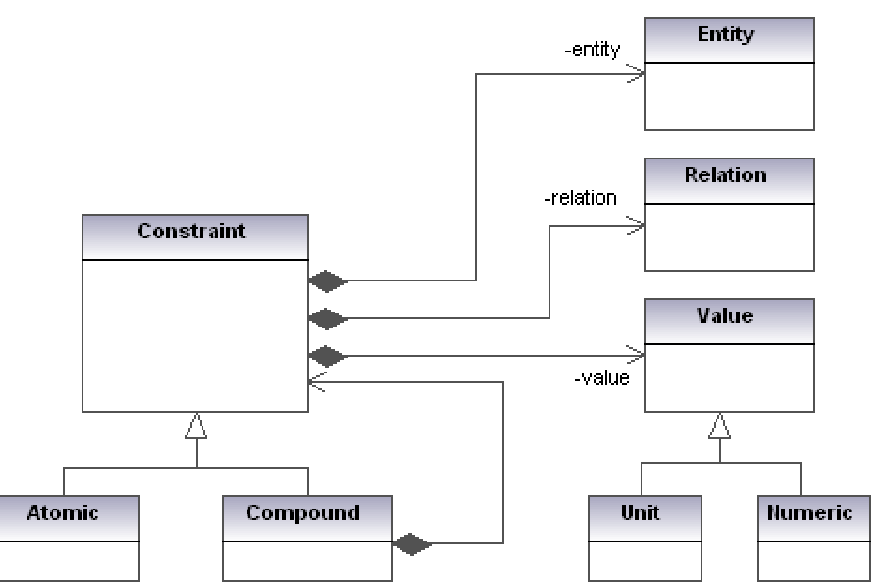


Figure 32 Excerpt of the Extended Metamodel for Constraint

The complete extended use case metamodel is shown in Figure 33. Due its overwhelming size and the fact that the complete diagram is composed from figures previously included

in this section, the diagram in Figure 33 is annotated with the figure numbers it is composed of for reference. The diagram included in Figure 33 is only meant for visualizing the completeness and the connectivity between the components. Meta-classes highlighted in red-color are enumeration classes. The extended use case diagram metamodel along with other encouraging applications such as Effort Estimation for use case analysis and application for metamodel interchange among UML tools are provided by Misbhauddin and Alshayeb [410].

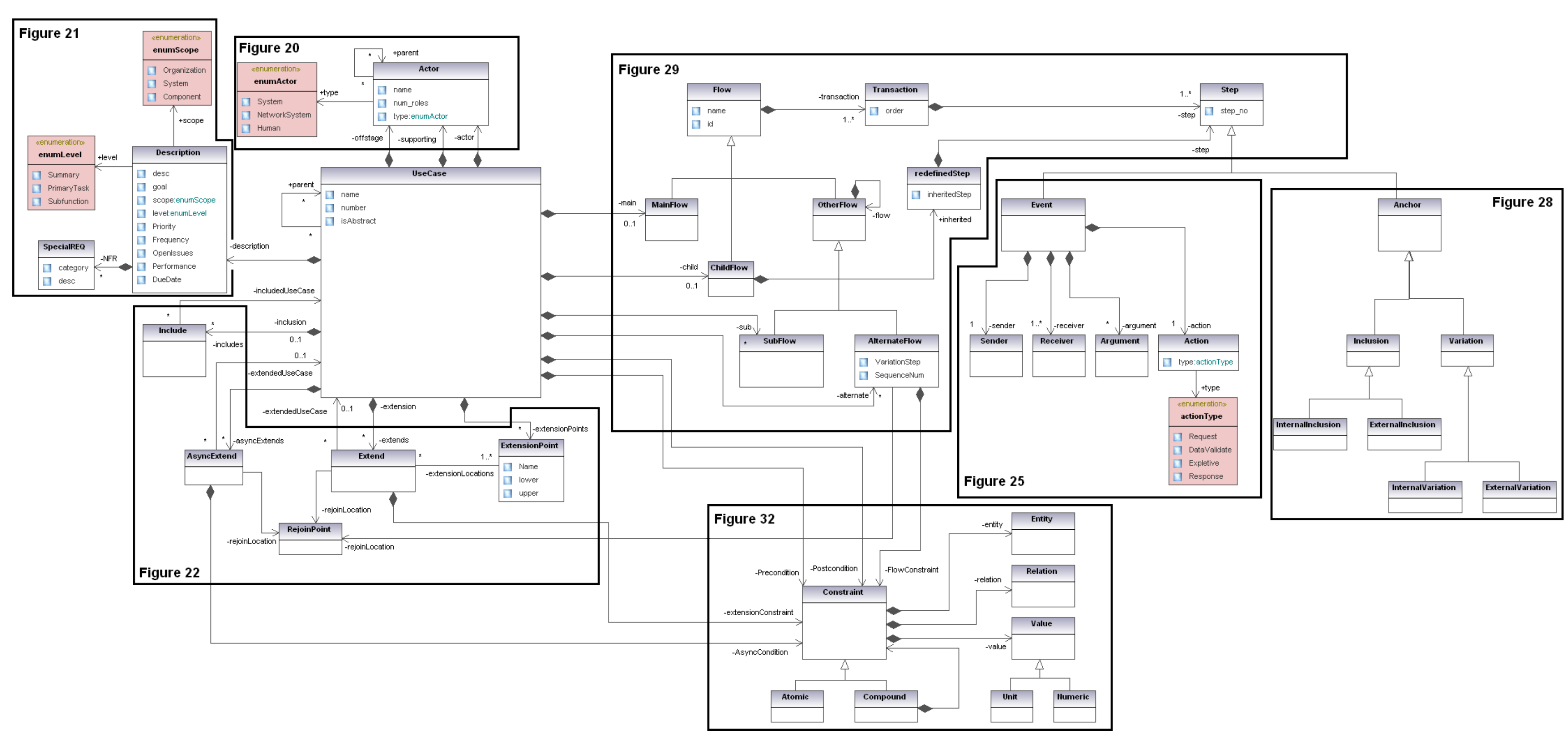


Figure 33 The Complete Extended Use case diagram Metamodel

Since the extended metamodel for use case diagram adds behavior, we need to augment the formal definition of the use case. A formal definition of a use case flow is given below:

Definition 4.4: A use case flow is a 6-tuple $\mathbb{U}_{flow} = \{\mathbb{a}, action, step, \leq, anchor, WF_{\mathbb{U}_{flow}}\}$ where

- $\mathbb{a}$ is a finite set of actors
- $action$ is a finite set of action labels
- $step \subseteq \mathbb{a} \times action \times \mathbb{a}$ is a finite set of steps in a use case flow
- $\leq$ is a partial ordering between steps and anchor
- $anchor$ is a set of location anchors part of the use case flow causing inclusion or variation
- $WF_{\mathbb{U}_{flow}}$ is a set of well-formedness rules on the Use Case Diagram UC

A use case step $s \in step$ consists of the following components:

- $\mathrm{sender}(s) \in \mathbb{a}$ is the actor initiating the action event.
- $\mathrm{acc}(m) \in action$ is the action event performed by the use case step.
- $\mathrm{arg}(m)$ is a list of arguments.
- $\mathrm{receiver}(m) \in \mathbb{a}$ is the actor receiving the action event.

Anchors in a use case flow are classified into two different categories: Inclusion and Variation.

- **[INCLUSION]** An inclusion anchor $\mathrm{inc} \in \mathrm{anchor}$ consists of a name and a body. The body of an inclusion anchor is given by another flow $\mathbb{U}_{\mathrm{include}}$.

- **[VARIATION]** A variation anchor $\mathrm{var} \in \mathrm{anchor}$ consists of the following components:
    - $\mathrm{name(var)}$is the name of the alternate flow or another use case.
    - $\mathrm{constraint(var)}$is the condition at which the variation is invoked.
    - $\mathrm{rejoin\ (var)}$ is the rejoin point from the variation.

    The body of a variation anchor is given by another flow $\mathbb{U}_{\mathrm{extend}}$.

4.5 Object Constraint Language (OCL)

OCL is a specification language and not an action language for UML. It is mainly used to write queries to access model elements and their values and state constraints on model elements. UML model elements are annotated with OCL constraints to ensure their proper usage and validity of the whole model.

4.5.1 OCL Metamodel

The OCL Metamodel part of the UML OCL 2 specification is given in Figure 34. As with the UML diagrams described in the previous subsections, all of them have constraints associated with one or more of their elements. The main reason for including the OCL metamodel, as part of the integrated model, is to have a consistent structure for all the constraints provided by the UML diagrams.

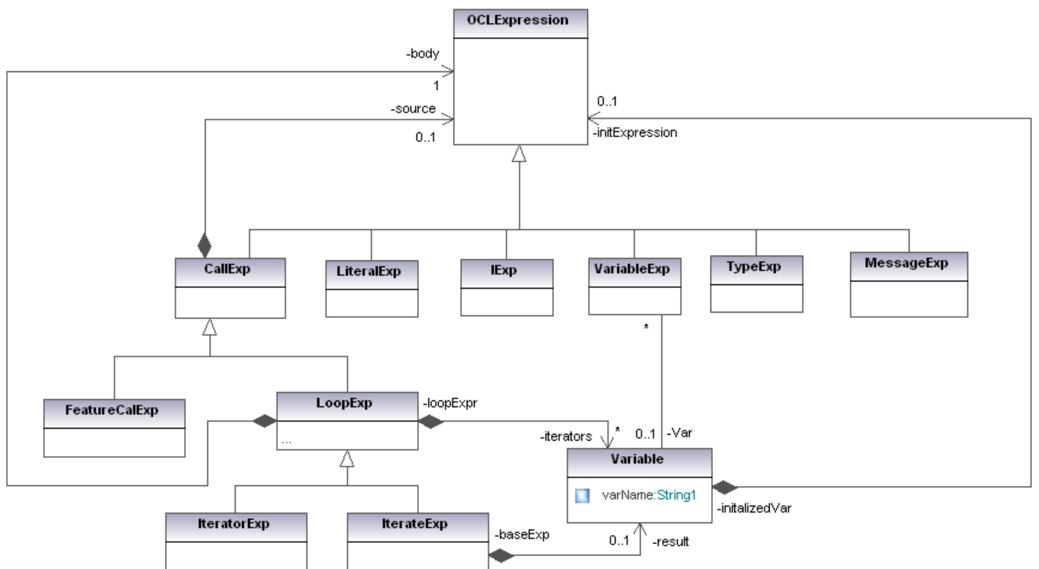


Figure 34 OCL Metamodel

A constraint in OCL is composed of a context and a set of expressions.

- **[Context]** The context contextof an OCL constraint consists of:
  - $name(\boldsymbol{context})$ is an optional name to address the context within the constraint's body of expressions. Alternatively, the *"self"* is also used.
  - $element(\boldsymbol{context}) \in \{\mathbb{C} \cap \mathrm{Attr} \cap \mathrm{Op}\}$ refers to the model element on which the constraint is defined.
- **[Expression]** An expression exp of a constraint consists of the following components:
  - $type(\boldsymbol{exp}) \in \{\, inv, pre, post, init\}$
  - $name(\boldsymbol{exp})$
  - $body(\boldsymbol{context})$

4.5.2 OCL Metamodel Extension

Although the OCL metamodel proposed by OMG is complete, it is rather comprehensive. Not all meta-classes included in the metamodel are used when describing constraints over the diagrams considered in our work. To make the OCL metamodel usable for describing constraints from class, sequence and use case diagrams in a structured yet simple manner, we adopt the extension proposed by Ramalho et al. [411]. They developed their metamodel from three sources: 1) The UML metamodel [10] to ensure integration with the latest UML standard, 2) the OCL EBNF (Extended Bacchus-Naur Form) grammar and 3) the OMG OCL Metamodel. The excerpt of the OCL metamodel considered for our work is shown in Figure 35. The *Constraint* meta-class consists of one or more expressions (*Expression* meta-class) and is associated with a *Context* meta-class.

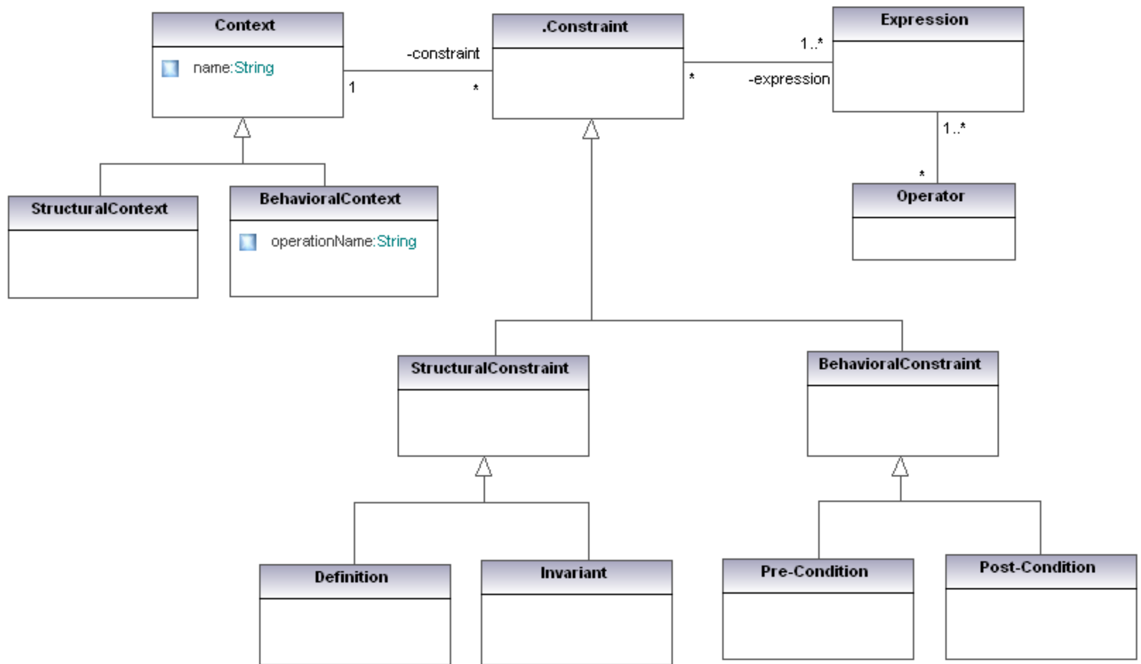


Figure 35 Excerpt of the Extended OCL Metamodel

4.6 Integrated Metamodel

Modeling a complex system requires the software designers to concentrate on multiple different aspects of the system. Designers have to take into account the static structure (attributes and operations), the dynamic behavior (scenarios, invariants), and its functional behavior (requirements, access rights) etc. Often complex metamodels are decomposed into a number of views particularly for multi-perspective metamodels such as UML. Designing models that conform to these metamodels often face consistency and integration problems between the different views. Usually, different views of the same metamodel share a common core. This common core inter-relates different views both at syntactic and semantic level. The UML specification provides only the syntactic commonality between views through high-level packages. With the advent of MDA, a number of approaches to integrate multiple views synthesizing semantic information have been proposed in the literature. In this section, we identify available approaches to link multiple views and use one of them to propose an Integrated Metamodel for refactoring multiple UML views.

Model Integration can be defined as the creation of links between previously separated models, services or processes. Although referenced by multiple terms such as Model Composition, Model Synthesis, Model Weaving and Model Merging, the concept of model integration has been applied to the domain of Model-driven software engineering for numerous applications. Some of the prominent applications include integrating formal approaches to visual modeling languages [412], integrating complementary information

[27, 413], merging/synthesizing models [414, 415] and interoperability with other enterprise metamodels [416-418] .

In order to link models at the same or different levels of abstraction, MDA provides two model integration approaches [9]:

1. **Model merging-based integration:** Two or more models are merged together to produce a model at the same or lower level of abstraction.
2. **Metamodel-based integration:** A mapping is defined between the metamodels of the models to integrate.

In this work, we use the metamodel-based integration approach to propose an Integrated metamodel. Integrating models at the metamodel level allows efficient use of Model-Driven Architecture techniques such as model weaving and model transformation. The main motivation for integrating metamodels in this work is to propose model-driven refactoring over multiple views of UML. Two main advantages of using an integrated metamodel for refactoring are:

1. **Interoperability:** The flow of information between multiple views can be visualized and aids in establishing techniques on how to extract or understand the information in order to process them.
2. **Inter-navigability:** Navigating across multiple models to identify refactoring opportunities can be very difficult. An integrated metamodel provides inter-navigability that allows accessing related information for smell detection and model refactoring.

The UML specification provides numerous different diagrams that allow designers to model the structural, behavioral and functional aspects of the system under development. The Integrated Metamodel proposed in this work is developed incorporating one diagram from each UML view. These diagrams cover structural, behavioral and functional concepts of UML. This restriction is introduced for a single primary reason: to avoid unnecessary complication in metamodel integration and model-driven refactoring. However, the approach can be extended and applied to the entire suite of UML diagrams.

To allow smooth integration of the metamodels, we initially identified missing information required to synthesize these metamodels. This information is depicted pictorially in Figure 36.

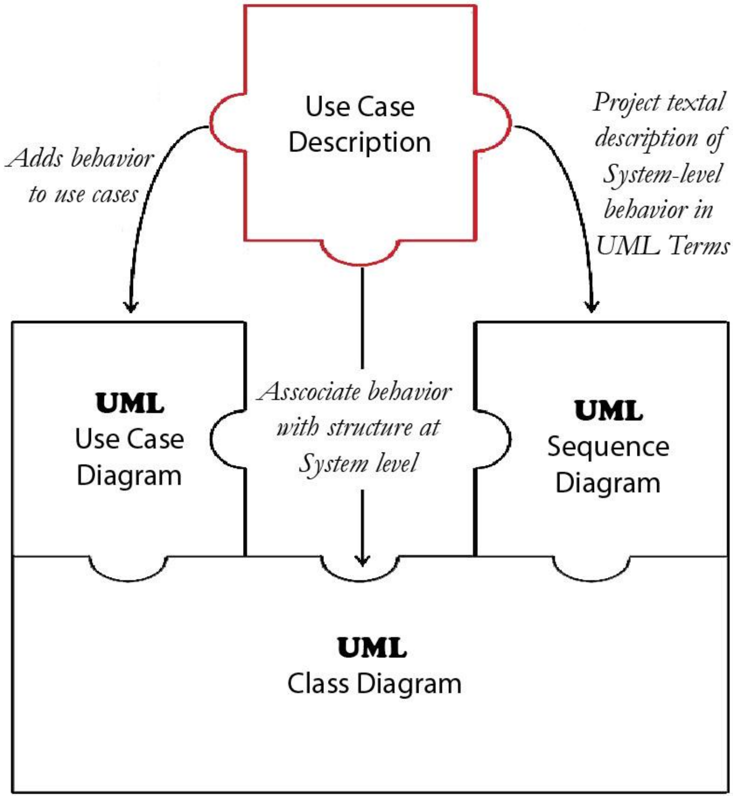


Figure 36 UML Model Integration Elements

In order to facilitate integration of the missing information, we extended the UML metamodels of Use Case diagram and Sequence diagram to ensure seamless integration. The Integrated metamodel is composed of metamodel of the class diagram (see Figure 13) that represents the structural view, extended metamodel of the sequence diagram (see Figure 17) that represents the behavioral view and the extended metamodel of the use case diagram (see Figure 33) that represents the functional view. In order to ensure complete modeling of information, the Integrated metamodel also incorporates the OCL metamodel so that constraints (from class diagrams), invariants and guards (from sequence diagrams) and pre and post conditions (from use case diagrams) are structurally represented.

In order to ensure that the integrated approach is unobtrusive, we followed the integration principles proposed by da Silva and Paton [413]. These principles are briefly summarized below.

- Standard UML should be retained as a subset in which existing constructs keep their roles and semantics.
- Integration should support complete applications, so links between integrated models and existing UML models should be well defined and close.
- Integration should introduce as few new model elements into UML as possible.

In order to obtain the integrated metamodel, we follow a stepwise model composition approach. The metamodels for use case and sequence diagrams are initially composed and then this *resultant metamodel* is composed with the class diagram metamodel.

Finally the OCL metamodel is added to get the Integrated metamodel. Based on the composition semantic defined in [419], the integrated metamodel composition approach is shown in Figure 37. The *receiving metamodel* is a term used to specify the metamodel into which the other metamodel is composed inside. The resulting metamodel is a term used to specify the metamodel obtained after the composition has been performed. Based on existing methodologies [420-422], metamodel integration mechanism involves three basic steps:

1. **The Comparison Step:** Correspondence between elements of the metamodel are identified and stored as a set of rules known as correspondence rules (also called comparison rules, mapping rules or matching rules).
2. **The Integration Step:** Models mapped in the previous step are integrated in this step based on an integration strategy. The integration strategy defines which elements will appear in the integrated model and how these elements will be organized.
3. **The Consistency Step:** The main objective in this step is to discover design errors, adverse properties and conflicts.

In the following sections, we will elaborate the comparison and composition mechanism following the stepwise construction of the integrated metamodel shown in Figure 37.

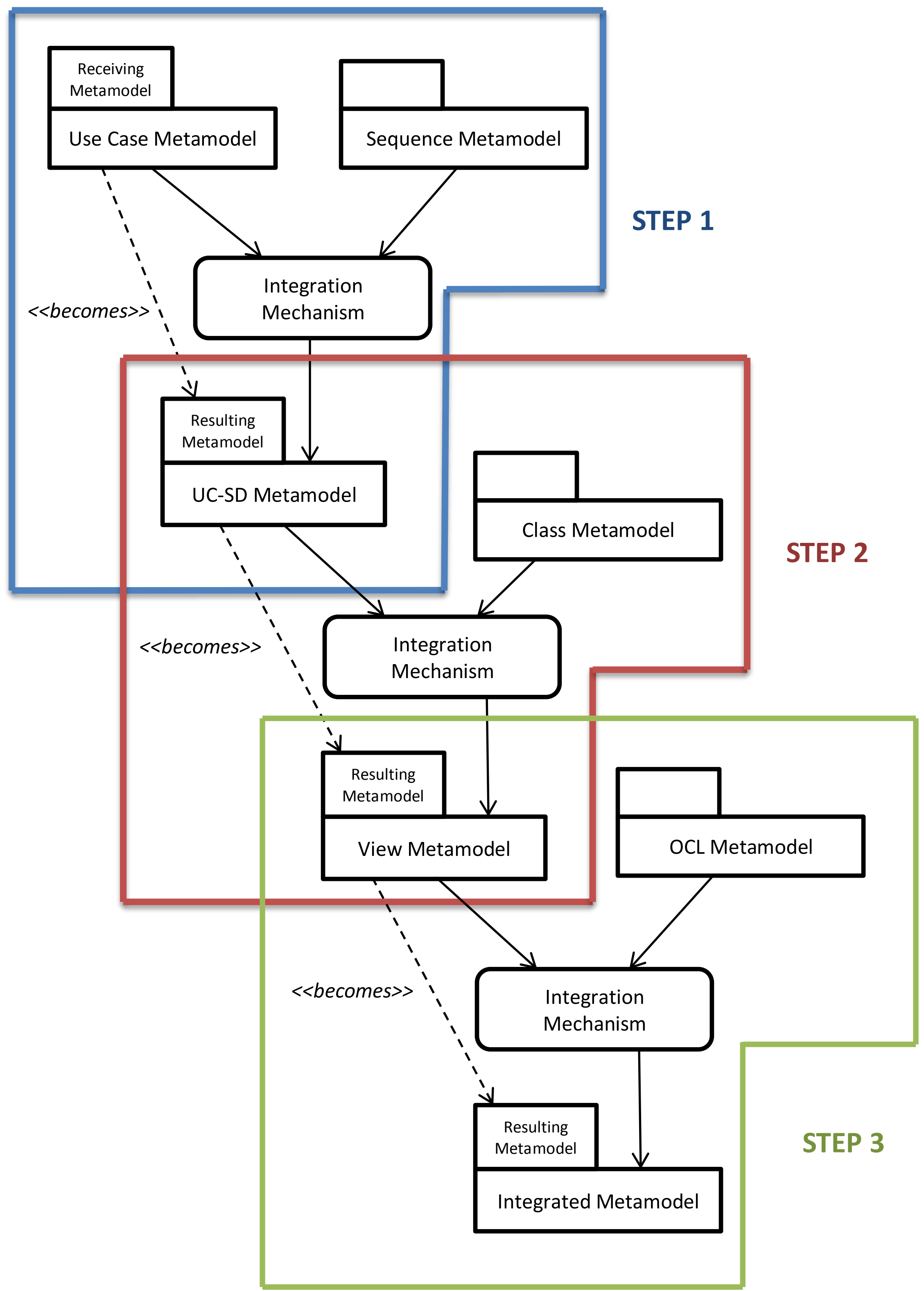


Figure 37 Model Integration Framework

4.6.1 STEP 1: Sequence and Use Case Metamodel Composition

In each step, we first identify correspondence between elements of the two metamodels. In order to identify correspondence, we generate a traceability matrix that highlights the mapping links between the two metamodels. The traceability matrix identifies the following types of correspondence links between the metamodel elements.

Syntactic Similarity (SYN): This correspondence relationship indicates that the two meta-classes related to each other by this link are syntactically equivalent. Usually, syntactically similar meta-classes are specializations of a common super-class in the UML Specification. Syntactically similar meta-classes are merged together in the resulting metamodel.

Semantic Similarity (SEM): This correspondence relationship indicates that the two meta-classes related to each other by this relation are semantically equivalent. In order to integrate semantically similar meta-classes in the resulting metamodel, correspondence rules are defined.

Inclusion (INC): This mapping link indicates that the meta-class is included in the resulting metamodel although no similarity exists between this meta-class with other meta-classes. Correspondence rules are defined to describe the association of this meta-class with other meta-classes in the receiving metamodel.

Exclusion (EXC): This mapping link indicates that the meta-class is excluded from the resulting metamodel. Typically, the main reason for exclusion is its relevance to the application of the Integrated Metamodel.

Dependency (DEP): This mapping link indicates that the two meta-classes related to each other by this relation are dependent. Meta-classes related by this link are usually kept in the resultant model and a directed dependency link is added between them.

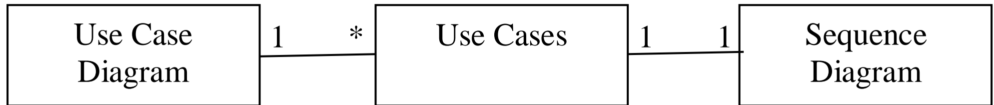


Figure 38 Abstract Relationship between Use Case and Sequence Diagram

The use case metamodel included in the UML specification provides only its structural elements. This is the reason why the use case metamodel was augmented with behavioral information by integrating use case flows or scenarios. Hence, this augmentation has made the use case diagram more similar to the sequence diagram. An abstract relationship between the use case and sequence diagram is shown in Figure 38. Based on this information, the use case metamodel is considered the receiving metamodel as it is composed of sequence diagrams.

In order to keep the size of the traceability matrix to a manageable dimension, the inclusion and exclusion meta-classes are listed separately in Table 4. Another important observation is that the *Constraint* and *StateInvariant* meta-class are added to the Integrated metamodel as-is in this step until the final step of OCL metamodel integration. Another important decision is to decide which meta-class to include in the Integrated metamodel in case of Structural Similarity. Based on the principles of integration summarized in the previous section, meta-classes closer to the UML standard are retained.

Table 4 Inclusion and Exclusion Meta-classes in Step 1.

Mapping Link	INC	EXC
Use Case Meta-classes	Use Case	Description
	Constraint	
	Include	SpecialREQ
	Extend	
	AsyncExtend	RejoinPoint
Sequence Meta-classes	StateInvariant	PartDecomposition
		ConsiderIgnore

The meta-classes for Include and Extend are added to the Inclusion list as they merely list the use cases included or extended by the base use cases. Their use in the behavior is provided by anchors (inclusion and variation) included in the traceability matrix. Based on the traceability matrix shown in

Table 5, a set of correspondence rules were generated that can be used for composing the use case and sequence diagram metamodel. The intermediate resulting metamodel (referred as UC-SD metamodel) is shown in Figure 39.

Table 5 Traceability Matrix for Use Case and Sequence Metamodel Composition

Sequence Diagram Meta-classes	Use Case Diagram Meta-classes													
	Actor	Extension Point	Flow				Event Step				Inclusion Anchor		Variation Anchor	
			Main	Child	Sub	Alt	Sender	Receiver	Action	Argument	Internal	External	Internal	External
Lifeline	DEP													
Message									SYN					
Message End							SEM	SEM						
Interaction			SYN	SEM										
Opt						SEM							SEM	
Loop						SEM							SEM	
Break						SEM							SEM	
Neg						SEM							SEM	
Par					SEM						SEM			
Alt						SEM							SEM	
Seq					SEM						SEM			
Strict					SEM						SEM			
Assert						SEM							SEM	
Critical						SEM							SEM	
Gate							SEM	SEM						
Interaction Use		SEM										SEM		SEM
Argument										SYN				

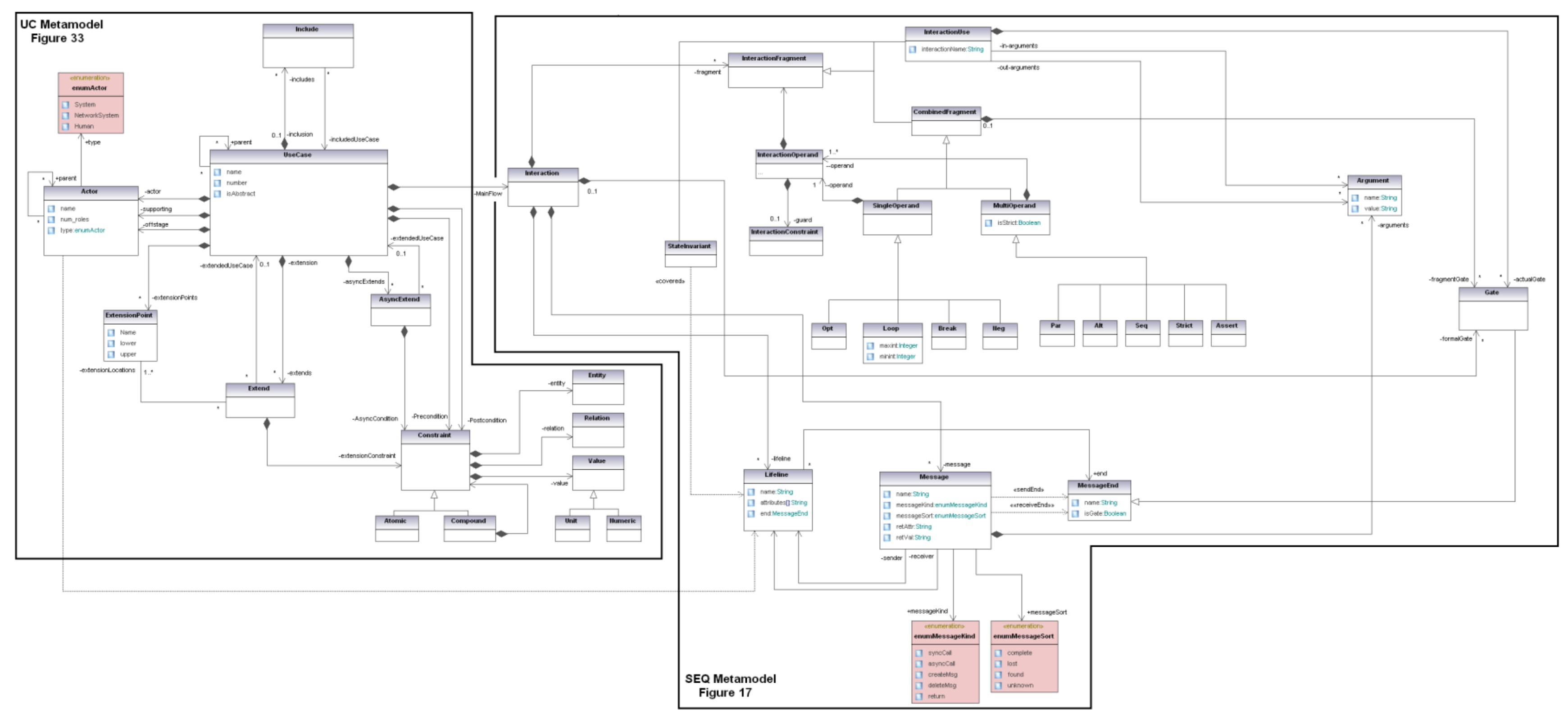


Figure 39 Step 1: The UC-SD (Intermediate) Metamodel

4.6.2 STEP 2: Class Metamodel Composition

Integrating the class diagram metamodel is simpler than the use case and sequence diagram metamodel integration. Although most of the traceability links between the class diagram metamodel and the UC-SD metamodel are structural similarity, we discourage its use due to the principles followed in the integration process. Hence, instead of merging the structurally similar meta-classes, we add the Dependency relationship between the related meta-classes. Thus, the structure of the class diagram remains intact for model evaluation and the dependency relation aids in navigating related information for model smell detection and refactoring. The mapping links between the class diagram meta-classes and the UC-SD meta-classes is shown in Table 6.

Table 6 Traceability Mapping between Class and UC-SD metamodel classes

Class Diagram Meta-classes	Mapping Links	UC-SD Meta-classes
Class	**DEP**	Lifeline
	DEP	Actor
Property	**INC**	
Operation	**DEP**	Message
Parameter	**SYN**	Argument
AssociationEnd	**INC**	
AssociationClass	**INC**	
Association	**INC**	
Generalization	**INC**	
Dependency	**INC**	

Operation meta-class in the class diagram metamodel is structurally similar to the *Message* meta-class in the UC-SD metamodel. In order to keep the semantics of the class diagram intact, a dependency link between the *Operation* meta-class and the *Message*

meta-class is added. Because of the above-mentioned composition, the *Parameter* meta-class is merged into the *Arguments* meta-class in the UC-SD metamodel. Since the *Parameter* meta-class has an attribute called *direction*, the association relationship between the *Arguments* meta-class and the *InteractionUse* meta-class is modified. Initially there were two associations differentiating between the input and the output arguments. These associations are now replaced with a single association and the *direction* attribute will handle the type of the argument (i.e. in or out).

A dependency relation is added between the Class meta-class and the Lifeline and Actor meta-class. This relationship is justified by the fact that any lifeline included in a sequence diagram needs to be available as a class instance in the class diagram. Similarly, an actor in the use case represents the role, which is usually transforms into an entity class within the class diagram. Hence, a dependency link between the Class and Actor meta-class is also added to the Integrated Metamodel.

A partially integrated metamodel that integrates the class diagram metamodel with the UC-SD metamodel is given in Figure 40. We refer to this intermediate resulting metamodel as the View metamodel as it integrated the three views of UML.

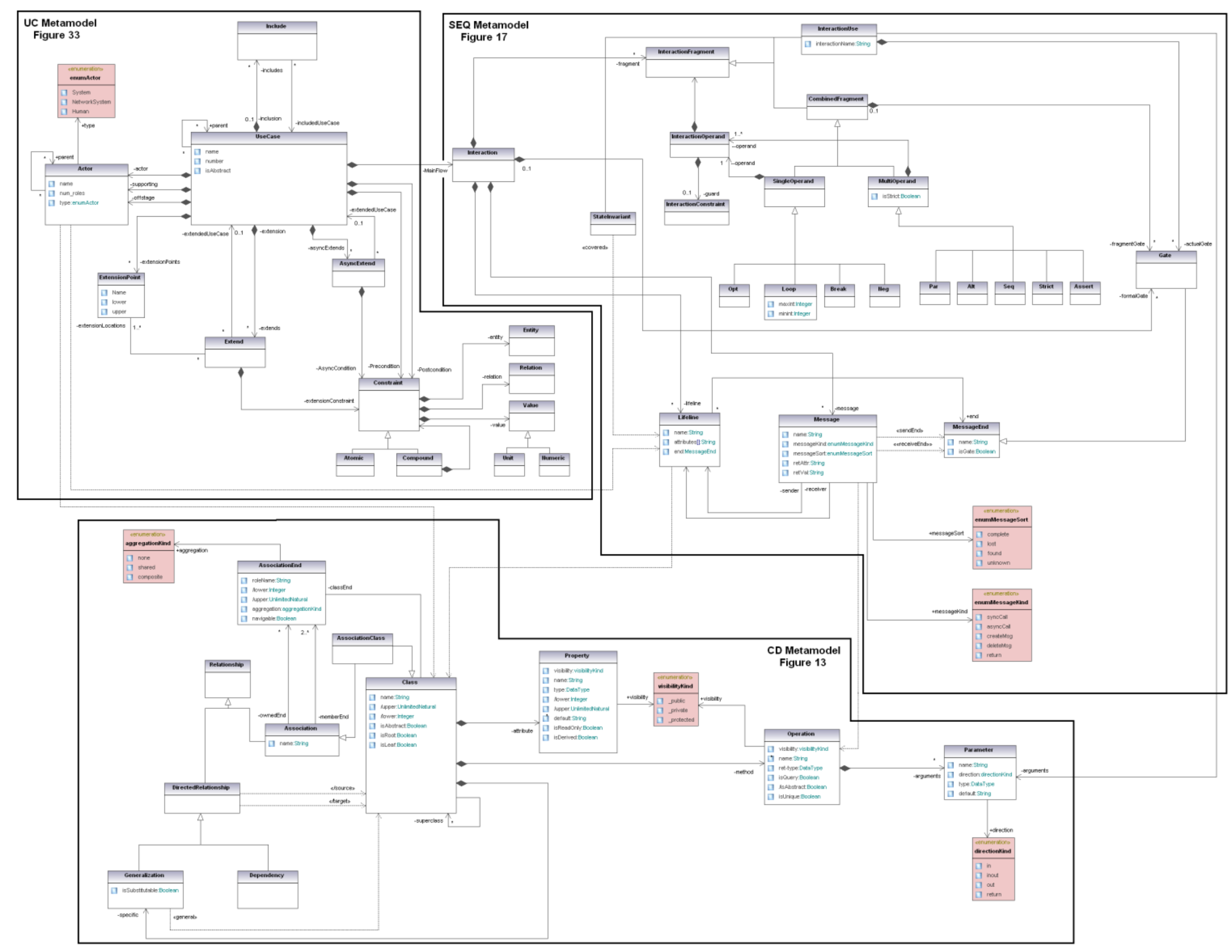


Figure 40 Step 2: The View (Intermediate) Metamodel

4.6.3 STEP 3: OCL Metamodel Composition

The final step in the stepwise composition of the metamodels is the inclusion of the OCL metamodel. The OCL metamodel defines a structure for describing the various constraints and invariants provided by the different views. The main meta-class in the OCL metamodel is the Constraint meta-class.

Since the context will be directly related to the *Constraint* meta-class in the Integrated metamodel, the meta-classes Context and its specialized classes *StructuralContext* and *BehavioralContext* are excluded. Based on the extension proposed for the Use Case constraints in the extended use case metamodel, a mapping was established between Constraint metamodel (from Use Case) and the OCL Metamodel as shown in Figure 41. Hence, as a result the constraints from the use case metamodel are mapped directly as context to the Constraint meta-class provided by the extended OCL metamodel.

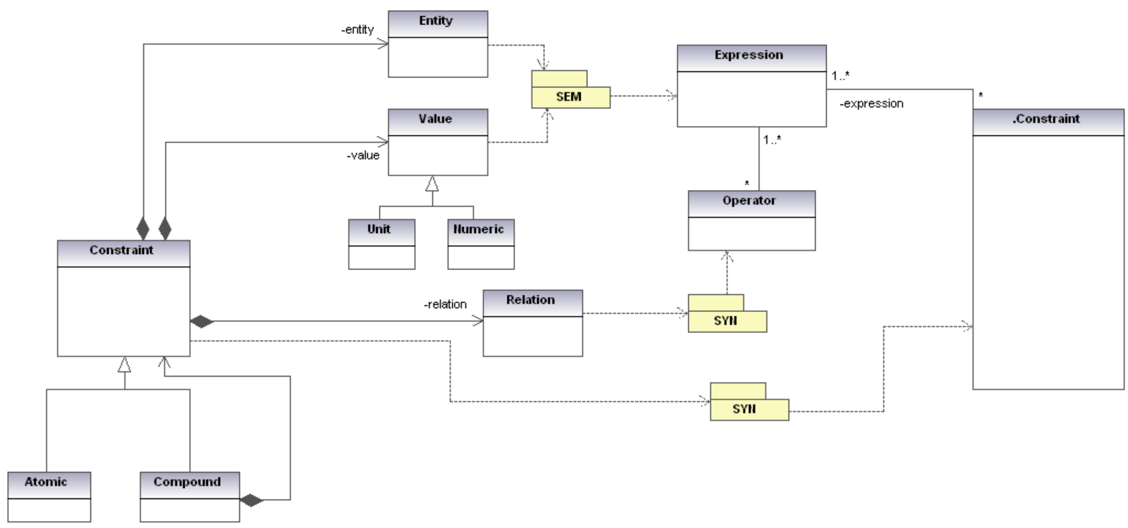


Figure 41 Traceability Mapping between UC Constraint and OCL Metamodel

The *StateInvariant* meta-class is replaced with a composition relation between the Constraint meta-class and the Lifeline meta-class. Although the *StateInvariant* meta-class was a subclass of the *InteractionFragment* meta-class in the View metamodel, the context of the invariant is the lifeline. Hence, the path "*Interaction (composition) Interaction Fragment (super-sub) StateInvariant*" was reduced by directly relating it to the Lifeline meta-class. Another constraint from the sequence metamodel is the Interaction Constraint that guards the Interaction Operand. Similar to the above mapping, a composition relationship is added between the *InteractionOperand* meta-class and the *Constraint* meta-class excluding the *InteractionConstraint* meta-class from the integrated metamodel.

The relationship between the *Constraint* meta-class and the *Class, Property* and *Operation* meta-classes is borrowed from the works of Warmer and Kleppe [188] and Lano [423]. Below we describe how these relationships can be exploited to create a translation mapping between the OCL metamodel and the View metamodel.

1. The most important way in which an OCL expression with type as context can be used is as an invariant. An invariant can be defined as a Boolean expression that evaluates to true if the invariant is true. Associating an invariant with a Class in a model means that any system made according to the model is faulty when the invariant is not met. This is represented in the integrated metamodel by the composition relationship with role-name *inv* between the *Class* meta-class and *Constraint* meta-class.
2. An initial value for a property can also be given by an OCL expression. An initial value is the value that the instance of the class will have on creation. This is

represented in the integrated metamodel by the composition relationship with role-name *init* between the meta-classes *Property* and *Constraint*.

3. An attribute may also have a derivation rule. Attribute is an instance of the meta-class *Property* in the Class Metamodel, and the derivation rule is an instance of the meta-class *BehavioralConstraint*. The fact that the rule describes the derivation for attribute is represented in the integrated metamodel by the composition relationship with role-name *derivation* between the meta-classes *Property* and *Constraint.*
4. Constraints attached to an operation that defines what properties should be true at initiation of the operation and at termination of the operation when it executes normally are represented by preconditions and post-conditions. This is represented in the integrated metamodel by the composition relationship with role-name *pre* and *post* between the meta-classes *Operation* and *Constraint*.

A complete diagram of the Integrated metamodel is depicted in Figure 42.

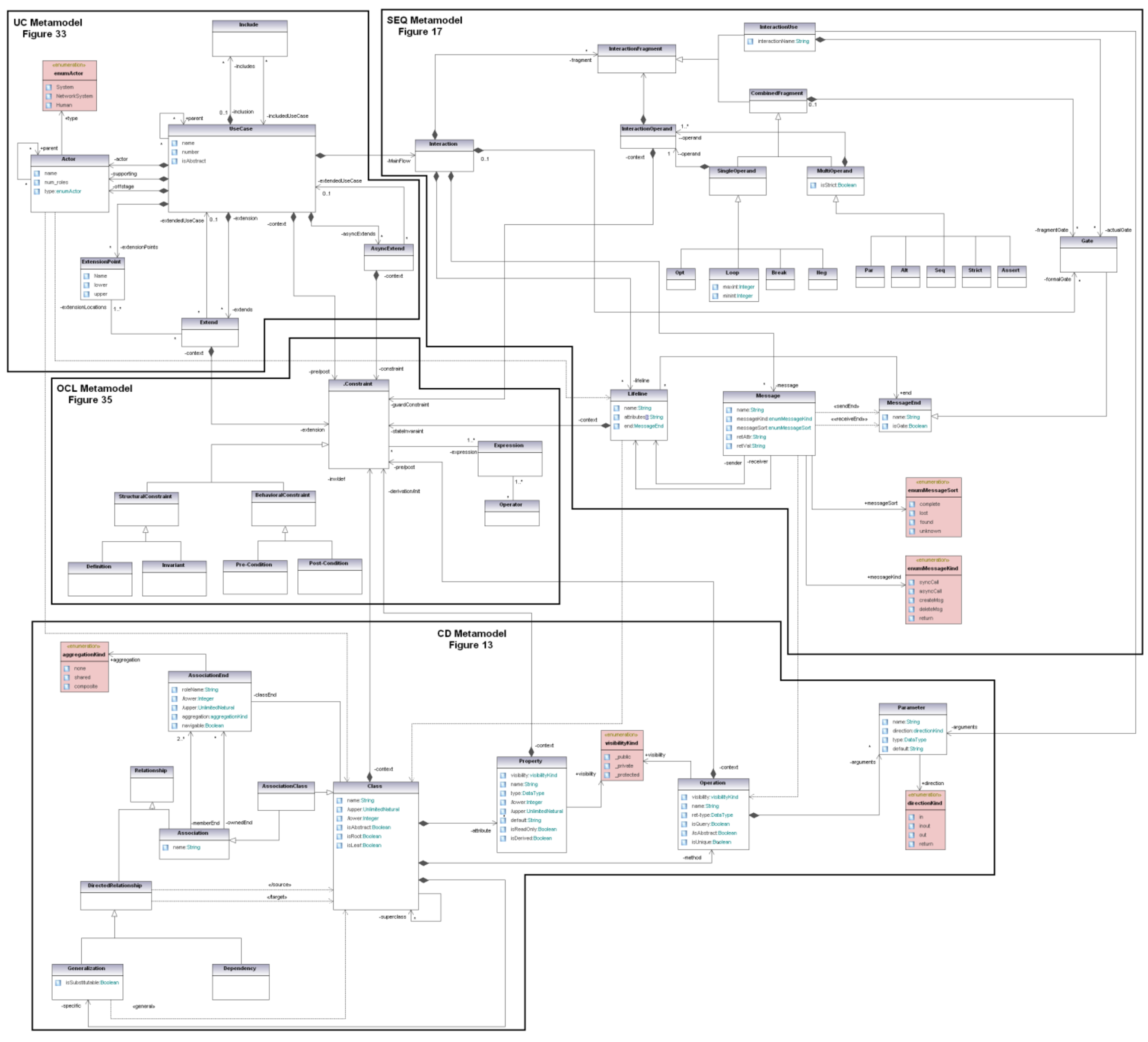


Figure 42 The Complete Integrated Metamodel

CHAPTER 5

INTEGRATED MODEL REFACTORING

Refactorings are usually defined in two ways. The first style is to identify and describe a refactoring opportunity (or bad smell) first and then propose a set of refactorings that either removes or alleviates the effect of this smell (also known as *Smell-Based Refactoring*). The second style is to describe a refactoring first and then provide a list of instances in which this refactoring can be applied. Fowler et al. in [15] used both of these ways when defining refactorings. For instance, Fowler et al. identified *Lazy Class* as a bad smell that occurs when a class is not handling enough responsibility in a system. In order to remove this smell, they proposed either using the *Collapse Hierarchy* refactoring (if a subclass) or *Inline Class* refactoring (if not a useful component). In another section, Fowler et al. first defined the refactoring like *Extract Method* and then provided motivations (*Long Method* or *Complex Method*) regarding when to use this refactoring (a.k.a. Bad Smell).

We use the former method of defining and describing refactorings in this chapter over the integrated model. The reason for this selection is two-fold:

- All model elements in the integrated model are similar to the model elements provided by UML. Hence refactoring operations over these elements (add, modify or remove) are already proposed in the literature. We make use of these primitive refactorings and propose a composite refactoring to handle the refactoring

opportunities identified in this work. A catalog of primitive refactorings defined over the UML model is provided in Appendix 2.

- Structuring refactoring definitions around bad smells increases comprehension and readability.

This chapter is organized as follows: Section 5.1 describes the standards and approaches used in our work to describe model-driven refactoring. Section 5.2 describes a template that will used in the remainder of the chapter to describe models smells and refactoring solutions. Section 5.3 describes a running case study used throughout the chapter to demonstrate the effect of refactoring. Section 5.4 describes eight integrated model smells proposed as part of this work in detail following the template described in section 5.2.

5.1 Model Refactoring Strategy

In Section 2.5, we identified and described a set of activities pertinent when proposing refactoring over models. In this section, we describe the formalisms and methodologies used in our approach to propose model-driven refactoring over the integrated UML model.

5.1.1 Model Transformation System

In order to select an appropriate model specification and transformation language, we identified a set of criteria to compare all available model-driven refactoring approaches. These criteria, proposed in the form of a comparison framework in [424], allows researchers and practitioners in selecting an appropriate approach suitable to their

specific needs and trade-offs. We selected the text-based (XMI) approach because of the following major advantages:

1. **Portability:** Models created in any UML CASE tool can be used for refactoring with minimal translation effort.
2. **Ease of Use:** Models represented in XMI are easier to follow as they are based on well-structured XMI Schemas. Simplicity of structure plays an important role when it comes to implementing complex refactoring operations.
3. **Expressiveness:** XMI-based standards provide numerous ways in which important refactoring activities can be expressed. For instance, XPath or XQuery can be used to describe refactoring opportunity detections and so on. Complete lists of standards used in this work are described briefly in Appendix 3.

Apart from numerous advantages, using text-based approaches such as XMI introduce a number of challenges. A major trade-off with XMI is the lack of formality. In order to overcome this issue, a lot of effort was invested in the design and implementation of parsing and model checking algorithms to ensure behavior preservation and model consistency. Two other relevant challenges posed are the amount of deep nesting and cross-referencing when working with XMI based approaches [425]. We circumvented these issues by mapping original XMI representations of UML models onto a simpler schema (an intermediate XMI representation) which resolves cross-referencing by replacing IDREF's with relevant information for model analysis and transformation. The intermediate schema also reduces the depth of tag nesting to a maximum of three, which aids in model navigation for smell detection algorithms.

5.1.2 Model Smell Detection Strategy

The focus of model smell detection is to fulfill the requirements regarding the description of the smell patterns. The core requirement for smell description is to describe them in a general and comprehensive manner. Smells are queries, which on execution must be able to detect their instances in the representation format of the model. The most well-known, widely used and standardized XML-aware query language is XQuery.

XQuery is a functional and declarative language that supports concepts of user-defined functions and modules which allows grouping of related functions into independent packages. In our approach, we use XQuery to describe models smells over the integrated model. More information on XQuery is included in Appendix 3.

5.1.3 Model Refactoring Application

Several techniques are available to perform refactoring application over models. These techniques have been classified into different top-level taxonomies, below is a list of some popular approaches:

1. **Direct Manipulation Approach:** Direct manipulation approaches use an internal representation of the model and a programming interface to manipulate the model. Tools that follow this approach make use of general programming languages like Java, C++ etc. providing a minimal infrastructure to organize the transformations. Transformation rules, behavior preservation primitives and scheduling in this approach are mainly done from scratch. The advantage of using a direct manipulation approach includes control over the internal representation of the model for model traversal and reorganization. But since transformation rules are implemented by the

user from scratch in this approach, it makes the transformation process cumbersome and hence affects reusability.

2. **Generic Transformation Approach:** Generic approaches use tools and languages such as XSLT or graph transformation tools [426]. Although a number of languages are available in the literature [427] for XMI-based representations, XSLT is considered the most popular of them all. Implementing model transformations using generic approaches such as XSLT seem attractive as models are serialized using XMI. Model refactoring using XSLT usually leads to non-maintainable implementation because of the verbosity and poor readability of XSLT. Peltier et al. [301] proposed an alternative approach to use XSLT to execute model transformation on the back-end instead of specification. Li et al. [302] also proposed an approach to use QVT relations to specify transformations and implement each relation as an XSLT rule template. The main reason specified for using XSLT as a back-end language is due to its low-level syntax. However, these approaches overcoming the previously listed problems also suffer from poor efficiency, as the pass-by-value semantics of XSLT require a large amount information copying.
3. **Template Based Approach:** Template based approaches separate the process of transformation rules description from the rule engine. A template usually consists of the target model containing splices of meta-information to access model elements from the source and perform model transformation. The source model accessing logic in this approach can be implemented in numerous ways. For instance, the logic could be a java code accessing the API provided by the internal representation of the source model or it could be declarative queries.

In this work, the direct manipulation approach is used to define and apply refactoring over the XMI representation of the UML models. The motivation behind this selection is mainly due to the use of the Integrated metamodel proposed in this work to represent the source model. The use of a direct approach allows complete control over the internal representation of the model for model traversal and transformation. Although fairly popular, XSLT and the template-based approach is not considered mainly because of the amount of information copying required between source and target models after each refactoring application and the high dependence of transformation engine tools respectively.

5.1.4 Model Behavior

As with other model-driven refactoring approaches proposed in the literature, we make use of pre-conditions and post conditions to ensure behavior preservation after application of refactoring. Each primitive refactoring operation is associated with pre and post conditions. Although an algebraic framework is used to describe these constraints, these are converted into programming language routines by the direct manipulation approach.

5.1.5 Refactoring Process

To demonstrate how the overall approach works, we discuss briefly the model refactoring process.

1. **Model Parsing and Integration:** To start, one model from each view specifically the class diagram, set of sequence diagrams and the use case diagram (along with use case descriptions) comprise the input layer of the approach. Each of these diagrams

are serialized using XMI and are imported by the prototype tool. Before the integration, each diagram is checked for structural and semantic well-formedness based on the rules provided in Appendix 1. Models are then unified into a single integrated model following the composition rules discussed in Section 4.6. The resultant model conforms to the integrated metamodel proposed in this work.

2. **Model Traversal and Smell Detection:** The integrated model is internally represented in the form of a Document Object Model (DOM) tree and traversed using an XMI parser. Model smells in the form of XQuery modules are then applied over the integrated model one-by-one. If a model smell exists within the model, the refactoring module is invoked.
3. **Model Refactoring:** The refactoring module invokes applicable rules from the repository and applies it over the model. Each refactoring rule in the repository is associated with two constraints *(T_{pre}, T_{post})*. If the pre-condition is satisfied, refactoring operations are applied over the source model. After refactoring, the post conditions are checked over the target model. If not satisfied, the refactoring operations are rolled-back and the source model is returned without any transformation.

5.2 Model Refactoring Template

In this section, we describe the template that is used to describe the refactoring opportunities proposed as part of this work.

1. **Description:** A description of the situation in which the refactoring opportunity is likely to occur.
2. **Rationale:** Reasons why the pattern described above is considered a model smell and is in need of change.
3. **Target Quality Improvements:** Quality aspects violated if this smell occurs. These usually include object-oriented principles, concepts and good design practices.
4. **Smell Detection Strategy:** Description of model smells using XQuery is the actual core of the Refactoring Engine. As the framework is customized for the detection of model smells, this section demonstrates how XQuery is used to describe and detect bad smells in the Integrated Model. An algorithm of the detection strategy is included in this section whereas the XQuery functions that realize this algorithm are included in Appendix 4.
5. **Refactoring Mechanics:** Refactoring operations can be classified into three categories based on their level of granularity: Primitive, Composite and Fine-Grain. Primitive refactoring is an atomic refactoring operation that cannot be split into more than one refactoring during application [6]. A sequence of primitive refactorings is known as composite refactoring. Composition of refactoring allows application of sequential refactoring operations on the model as a single unit [428]. This section includes composite refactoring rules (mainly due to the use of primitive refactoring operations for class, sequence and use case diagrams from the literature) to handle the detected model smell. This subsection is structured into four parts as follows: Name, Preconditions, Mechanics and Post conditions. Behavior preservation in the target model is ensured with the help of preconditions and behavior-preserving

transformations (Mechanics). A list of post conditions, which should be valid after a refactoring, are also specified. Post conditions are useful in building tool support.

6. **Example:** A simple example to illustrate the applicability of the model refactoring is included. Of course, such examples can only show certain aspects of the usability of model refactorings. They cannot demonstrate their complete functionality and the variety and flexibility of possible applications. Since there is no visual representation of the integrated model, the examples include separate class, sequence and use case diagrams.
7. **Post Refactoring Improvements:** The effect of refactoring on each view of UML considered is discussed to highlight the expected improvement.
8. **Side Effects:** Refactoring sometimes lead to violation of user-implemented strategies. Any side effects as a result of refactoring application are included in this subsection.

5.3 Running Case Study

In order to make the presentation more concrete, we demonstrate the proposed refactoring application throughout this chapter over a running case study: Net Banking System (NBS). The following description sets up the context of the running example.

NBS is designed for financial institutions such as banks to provide their basic banking operations over the internet. The system allows customers to open accounts, perform online transactions like transferring money, paying bills and viewing account summaries. The system also allows bank operators and

administrators to perform updates to the system online and handling other online operations.

The services provided by the system are summarized below and all functional requirements of the NBS system modeled through a use case diagram are shown in Figure 43.

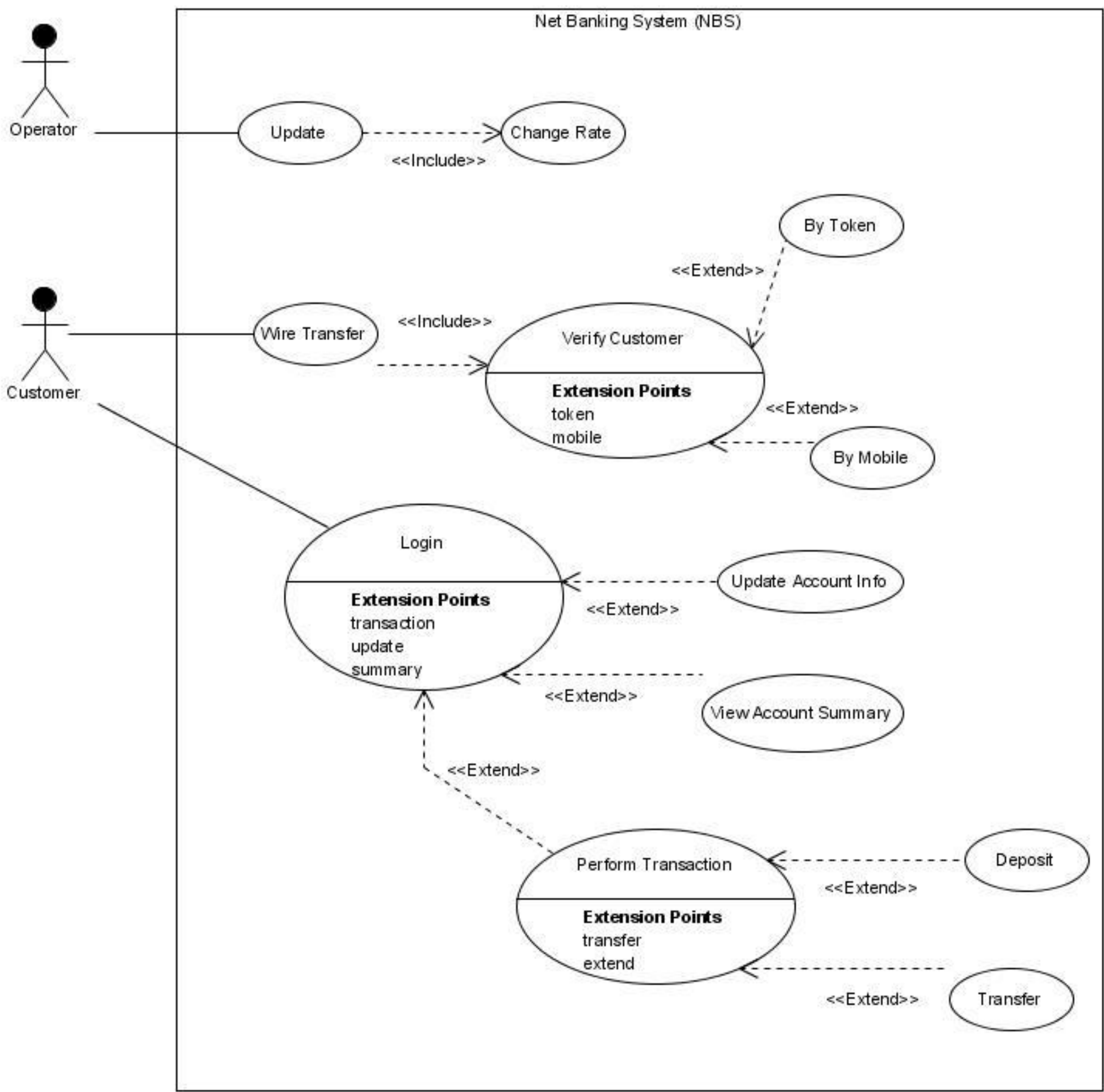


Figure 43 Use Case Diagram of the Running Case Study

1. **Open a new account:** New customers and existing customers can open a new account.
2. **Bill Payment:** Customers can use one of their accounts for bill payment. Popular agencies that can receive bill payments are already stored in the system. A customer can either enter the amount to be paid or pay the full retrieved amount based on the vendor account information provided. Regular auto-pay service for recurring monthly payments can also be setup. Bills can be marked as Favorite to avoid entering information each time a payment is made to the same agency.
3. **Transfer Funds:** A customer can transfer money between his accounts. Transfers to other accounts require a Beneficiary setup prior to the transaction. If a transfer is made to an existing beneficiary, the setup process is by-passed.
4. **Account Summary:** A customer can get an account summary for all his accounts.
5. **Transaction History:** A customer can get transaction history for all his accounts. This includes all transactions performed between a selected duration of time.
6. **Admin Services:** The system allows bank operators and system administrators to perform system updates, conflict resolution, account management and update through the NBS.

5.4 Integrated Model Smells

5.4.1 Creeping Featurism

5.4.1 (a) Description

Functional decomposition is a design methodology in which functionality provided by the system is modularized for fine-grain control over implementation and ease of understanding. Although useful in understanding the modular nature of a larger-scale application, functional decomposition is considered an anti-pattern when applied to object-oriented domain [429]. Functional Decomposition in use cases is caused by separating analytical use cases into functions that yield a set of smaller use cases that are naturally easier to implement. This structuring, if not controlled, will result in many small use cases that offer little or no value to the system's users if executed individually. Hence, the use case structure creeps directly into the design of the system making it look like use cases completely obscuring the concepts of objects and their relationships. This is referred to as *Creeping Featurism* Model Smell [430].

Use cases in UML are structured using pair of relationships between them: *include* and *extend*. Functional decomposition most commonly occurs due to the misuse of the include relationship. The effects of functional decomposition do not simply stop at the functional level; it disperses into the structural and behavioral level as well. A high degree of functional decomposition will result in behaviorally rich classes manipulating a number of dumb data classes. This indicates that responsibility is improperly distributed among classes. Data classes are classes that have only attributes, getter operations and setter operation [15]. Since getter and setter operations may be omitted by convention, a

data class is just a collection of attributes which defeats the purpose of Object-Oriented design methodology.

5.4.1 (b) Rationale

The anti-pattern of functional decomposition has been addressed recently in the literature in the context of UML model refactoring. Three out of four detection approaches propose the use of class diagrams to detect functional decompositions [19, 22, 24]. One major side effect in these propositions is the use of lexical analysis of class names to classify them as *Functional* classes. El-Attar and Miller [264] are the only ones who described the functional decomposition pattern over use case diagrams. They simply merge the functionally decomposed use case into the base use case without further analysis. The Creeping Featurism Model smell detects the occurrence of functional decomposition over use case, sequence and class diagrams.

When working with use case diagrams, it can sometimes be the case that a number of use cases delegate smaller tasks to other use cases by making use of the include relationship. Although this helps in managing the complexity of the use case, it renders the whole use case model difficult to comprehend. Another drawback is when this logic results in the creation of smaller, less useful classes in the class model just to handle to the small task initially created in the use case model. These small classes will also increase the complexity of the sequence model by allowing behaviorally rich classes to use them as data placeholders and increasing the message communication traffic for simple get and set operations. Identification and removal of this model smell is beneficial to the user in

order to manage the modularity and complexity of the class, sequence and use case models and to ensure proper usage of object oriented design methodology.

5.4.1 (c) Target Quality Improvements

- Management of Use Case Complexity
- Behavior Distribution
- Modular Design/Cohesion

5.4.1 (d) Model Smell Detection Strategy

Initially, we define a use case that performs small tasks and provides little or no value to other use cases or actors. We refer to this use case as a *Lazy Use Case* (based on the naming of a class that does nothing in a class model proposed by Fowler et al. [15]).

> **Definition 5.1** Lazy Use Case: A use case is termed as a lazy use case if
> - It is an inclusion use case
> - It has no actors associated with it
> - Included only once by another use case

To identify the availability of this model smell in an integrated model, a lazy use case needs to be identified. The interaction part of this use case is then examined to look for data classes. A class is termed as a data classes if it has only attributes and getter/setter methods. The pseudo code given below describes the steps required for automated detection of the creeping featurism model smell.

```
: ALGORITHM: CREEPING FEATURISM
: start
:   read Model
:   for (each use-case in the Model)
:     read UC
:     if (UC inclusion count is 1) and (UC has no actor)
:       parent = Including use-case of UC
:             diff = (lifelines in parent) ∩ (lifelines in UC)
:       if (diff is a data class)
:                   return diff
:             end if
:           end if
:   end for
: stop
```

5.4.1 (e) Model Refactoring Mechanics

Name: Remove Functional Decomposition

Parameters: Usecase *uc*, Usecase *inc*, Class *c* and Class *d* where,

- uc is the lazy use case
- inc is the use case that includes the lazy use case
- c is the data class
- d is the behaviorally rich class that manipulates the data class c

Preconditions:

i. Class c is not abstract.
ii. Class c and d has no common attributes.
iii. There is an inclusion relationship between use cases inc and uc. The use case inc includes the use case uc.

Mechanics:

1. Remove Data Class (Part of the inclusion use case). This is done by identifying the class that has maximum interactions with the data class. Then use *Inline Class* refactoring to merge the data class into the identified class.
2. *Substitute Lifeline* refactoring is then used to remove all references to the old data class from all interaction diagrams and replace it with its merged class.
3. *Collapse Fragment* refactoring is then used to insert the interaction fragment of the inclusion use case into the interaction diagram of the including/base use case at the point of inclusion (ref fragment).
4. Finally, *Merge UC Inclusion* refactoring is used to merge the inclusion use case into the including use case.

Figure 44 shows the ordering of the composite refactoring *Remove Functional Decomposition*.

Post Conditions:

i. All association ends with class c in the previous model are replaced with class d in the refactored model.

ii. Class c is removed from the model

iii. The interaction fragments for use case uc is collapsed and merged into the interaction diagram of use case inc by inserting it at the point of inclusion.

iv. Lifelines with reference to class c are replaced with reference to class d.

v. The inclusion relationship between use cases inc and uc is deleted.

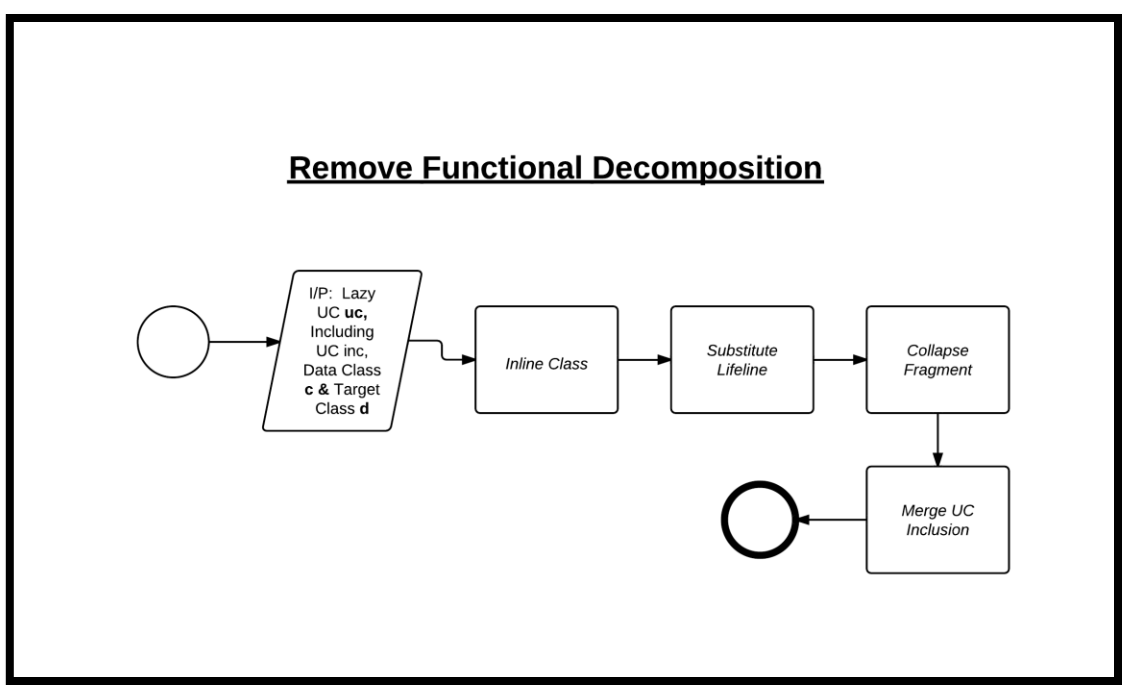


Figure 44 Remove Functional Decomposition Refactoring

5.4.1 (f) Example

Figure 45 shows a subset of the model views from the NBS system that depicts the creeping featurism model smell. The *Change Rate* use case is included only by the *Update* use case and is not associated with any actor. On further examination of the sequence diagram for the *Change Rate* use case and *Update* use case, we identified a behaviorally rich class *BankServer* using a data class *InterestRate* (based on information from the class diagram).

The *InlineClass (BankServer, InterestRate)* refactoring is first applied to inline and remove the class *InterestRate.* The *SubstituteLifeline (BankServer, InterestRate)* refactoring is then applied to substitute and redirect all messages that were initially communicated to/from *InterestRate* to *BankServer.* The *CollapseInteraction (Update,*

changeRate) refactoring is then applied to merge the *changeRate* interaction into the *Update* interaction at the point of fragment reference. The interaction for *changeRate* is hence deleted as part of the *CollapseInteraction* refactoring. Finally, the *MergeUCInteraction (Update, ChangeRate)* refactoring is applied to merge the functionally decomposed use case *ChangeRate* into its base use case *Update*. The refactored model views are shown in Figure 46.

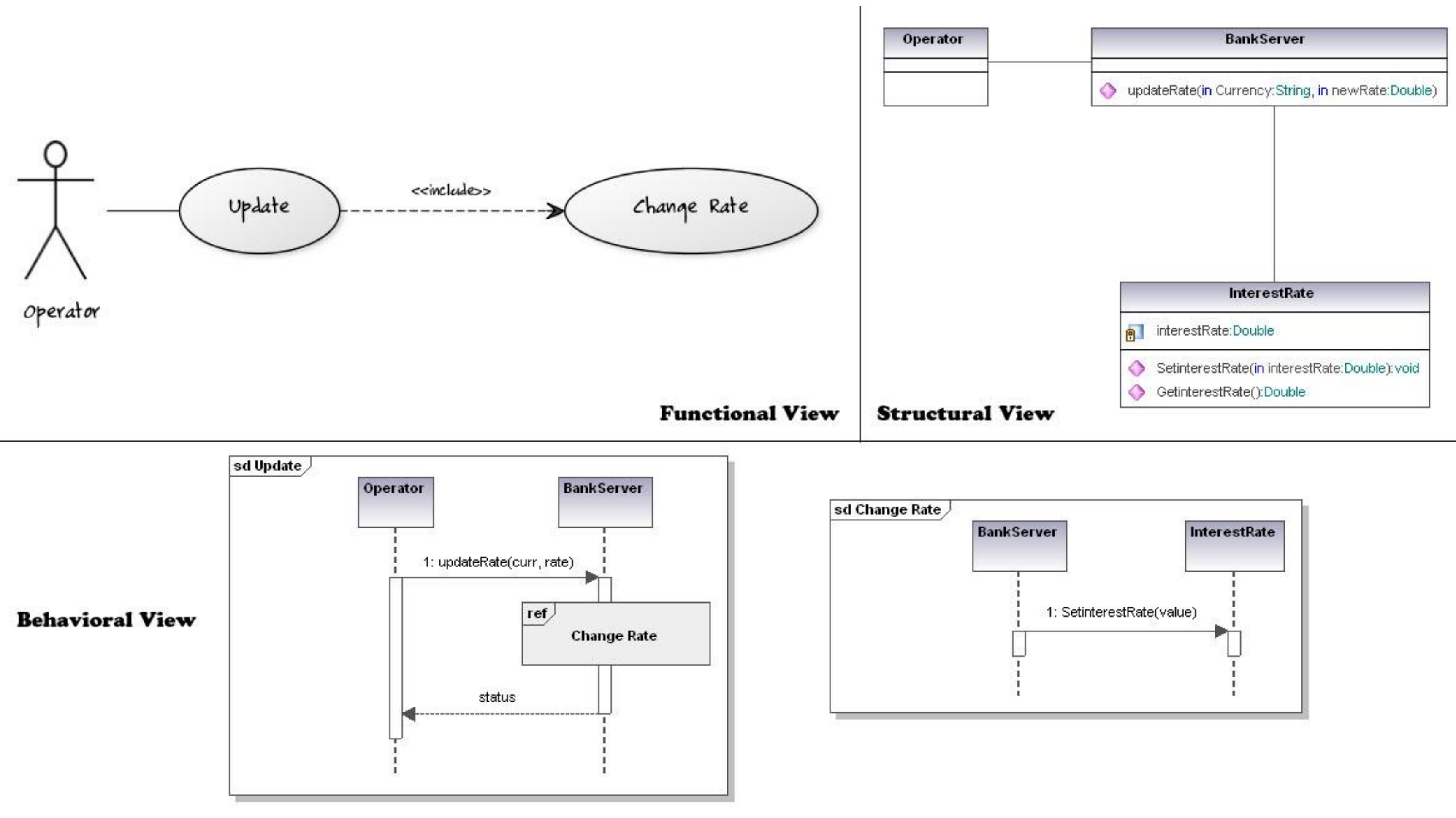
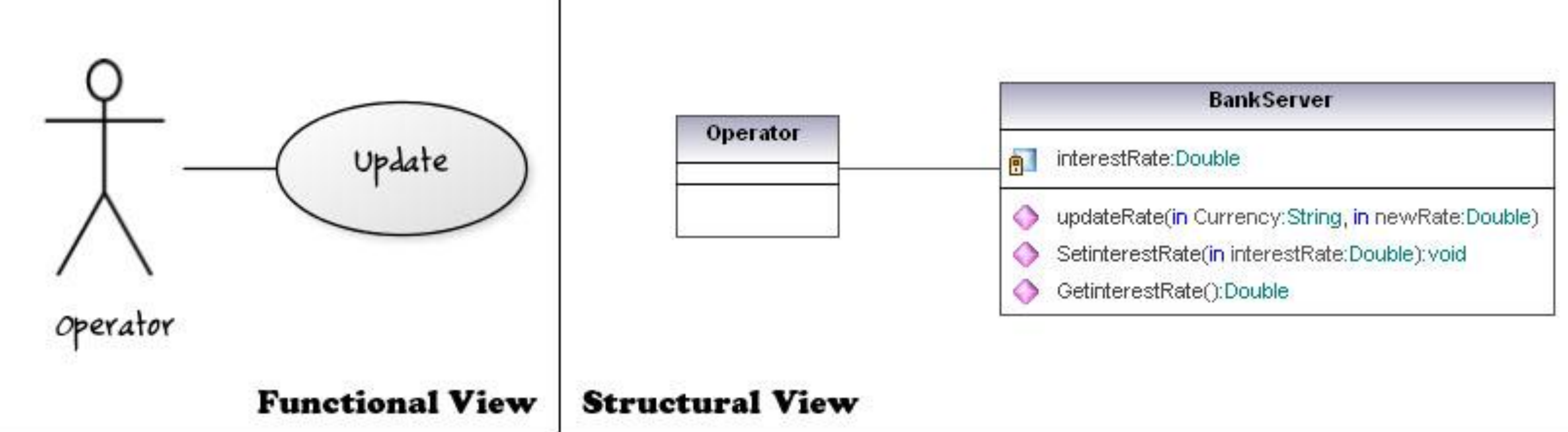


Figure 45 Excerpt of the NBS model views depicting Creeping Featurism Smell



Behavioral View

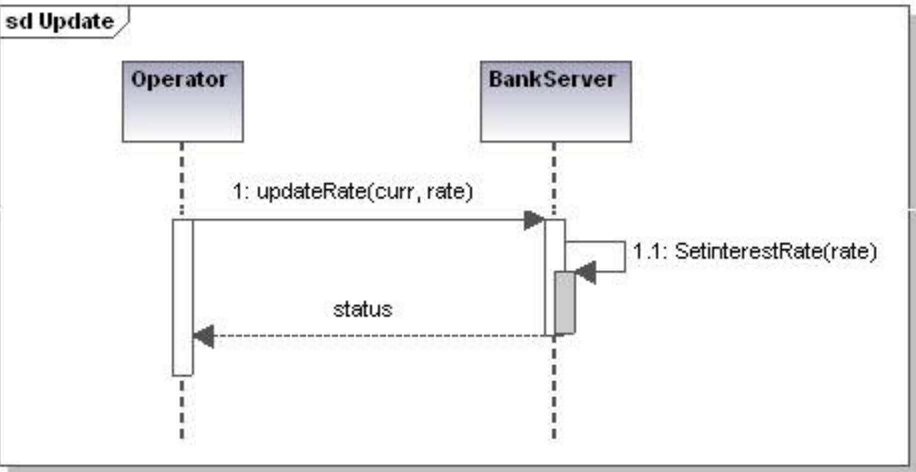


Figure 46 Excerpt of the NBS model views after refactoring

5.4.1 (g) Post Refactoring Model Improvement

The Functional View of the refactored model will not have unnecessary inclusion relationships and hence will reduce the complexity of the use case model view. The Behavioral View of the refactored model is improved a lot as a result of the refactoring operation. Some notable improvements are Reduction in the number of get and set messages exchanged between behaviorally rich classes and dumb data classes, removal of simple interaction fragments that result in referring to multiple sequence models for comprehension and enhanced behavior distribution by moving data to lifelines where it is used mostly. The Structural View of the refactored model will show improved modularity by the removal of data classes that increase coupling.

5.4.1 (h) Side Effects

Functional Decomposition when done due to lack of object-oriented knowledge is surely considered a smell and needs to be refactored. However, sometimes smaller use cases are extracted from a larger use case for future use by either associating an actor or making it reusable for other use cases. Using the Remove Functional Decomposition refactoring discussed in this section will result in deletion of this use case.

5.4.2 Multiple Personality

5.4.2 (a) Description

Multiple personality smell [430] is a result of inappropriate requirements allocation. It can be found in use cases that play multiple roles. Ideally, each use case is required to play a single role. Hence, it is required that a use case contains only one, coherent set of responsibilities. Multiple personality can lead to the detection of two different situations:

a secondary role superimposed on a single class or multiple classes cutting across a single use case. The former is a well-known anti-pattern known as God Class or Blob [431]. Following the same terminology, we refer to the later in our work as a *God Use Case*. A God use case is a result of improper partitioning of responsibility during system evolution, so that one module becomes predominant.

Based on the works done to estimate the effort required for use case implementation [385, 386, 432], use cases are classified into three categories. A use case is considered *simple* if it has three or fewer transactions and the implementation of which requires five or fewer classes. A use case is considered *average* if it has four to seven transactions and the implementation of which requires five to ten classes. Finally, a use case is considered *complex* if it has more than seven transactions and the implementation of which requires more than 10 classes. Redistribution of functionality from a God Use Case becomes easier when we take a closer look into the behavior of the use case. Some of the identified symptoms are:

- A God Use Case includes a number of lazy classes. This will result in increased count of classes participating in the use case. Removing these lazy classes will reduce the complexity of the use case.
- Existence of middle man lifelines in the interaction of the use case. A middle man is a lifeline that sits between two other lifelines and just forwards method calls. Removing middle man elements will reduce the transaction count and number of classes implemented by the use case.

5.4.2 (b) Rationale

When working with use case diagrams, it can sometimes be the case that although the overall model is small and compact but each use case may be highly complex. Although we agree that complexity is a subjective term but a use case, which covers multiple system goals, handles multiple requirements, whose behavior description cannot be covered in a single page should be termed complex. Although use of complex use cases within the use case model generates a neat and well-organized functional view of the system, its behavioral view is surely complex with wide array of messages exchanged between a number of incoherent classes and extensive concurrent set of operations. Identification and removal of this model smell is beneficial to the user in order to manage the complexity of the sequence models representing the complex use cases. This in turn will also affect the modularity of the class model.

5.4.2 (c) Target Quality Improvements

- Management of Use Case Complexity
- Management of Behavior Complexity
- Modular Design/Cohesion

5.4.2 (d) Model Smell Detection Strategy

To identify the availability of this model smell in an integrated model, a God use case needs to be identified. The interaction part of this use case is examined to look for number of classes implemented by the use case and the number of transactions.

Definition 5.2: A use case is termed as a God Use Case if

- Its implementation contains more than 10 classes
- Its behavior has more than 7 transactions

Based on the definition provided by Astels [251], we define a pattern for detecting whether a lifeline is a middle-man or not. Each lifeline in the integrated model has event ends associated with it. These event ends are ordered and depicts the type of the message such as send event, receive event and so on. If for a lifeline, these events are ordered as shown in Figure 47, then the lifeline is considered as a middle-man as its only job in the diagram is to delegate message from one lifeline to the other.

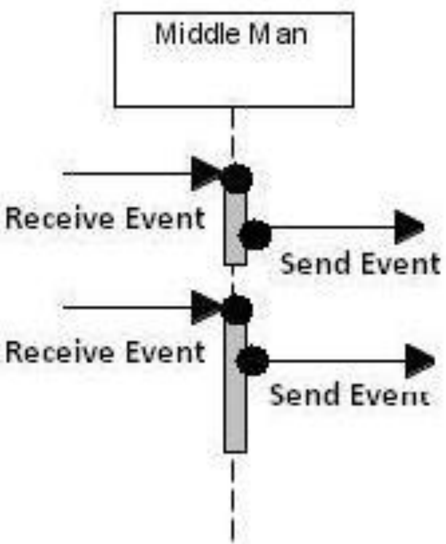


Figure 47 Middle Man Lifeline Pattern within a Sequence Model

The pseudo code given below describes the steps required for automated detection of the multiple personality model smell. The code returns a value of 0 if the smell does not exist, a value of 1 if the smell exists with inclusion of lazy classes in the God use case and a value of 2 if the smell exists with both inclusion of lazy classes and middle-man lifelines in the interaction.

```
: ALGORITHM: MULTIPLE PERSONALITY
: start
:   read Model
:   for (each use-case in the Model)
:     read UC
:     if (# of classes in UC is > 10) and (# of transactions in UC is > 7)
:                                              and (# of lazy classes in UC >= 2)
:       for (each lifeline in the UC)
:                               read Life
:                               end-List = (all ends on Life)
:                     for (each substring ss of end-List of size 2)
:                                         if (ss = {receiveEvent, sendEvent})
:                                              return 2
:                                         end if
:                     end for
:                               return 1
:            end for
:          else
:            return 0
:          end if
:   end for
: stop
```

5.4.2 (e) Model Refactoring Mechanics

Name: Decompose God Use Case

Parameters: Usecase *uc*, List *midman*, List *lazyClass*, List *base* where,

- *uc* is the God Use Case
- *midman* is the list of classes within the interaction of the God Use Case which are middle man lifelines
- *lazyClass* is the list of lazy classes
- *base* is the list of classes that that will inline the lazy classes

Preconditions:

i. Class *lazyClass* is not abstract.

ii. Class *lazyClass* and Class *base* has no common attributes.

iii. The direct base class of the Class *lazyClass* is also a base class of the Class *base*.

iv. The Class *lazyClass* is a sub class of the Class *base* or the two classes do not share any methods.

v. *Midman* is a lifeline model element in *uc*.

Mechanics:

1. In order to remove the lazy class, *Inline Class* refactoring is used to remove lazy classes that are not useful independent components. If the lazy class is a sub class, then use *Collapse Hierarchy* refactoring to merge the class into its super class.
2. *Substitute Lifeline* refactoring is then used to remove all references to the old lazy classes from all interaction diagrams and replace it with its merged class or super class.
3. Finally, *Remove Middle Man* refactoring is used to remove the lifelines from the use case interaction.

Figure 48 shows the ordering of the composite refactoring Decompose God Use Case.

Post Conditions:

i. All association ends with Class *lazyClass* in the previous model are replaced with Class *base* in the refactored model.

ii. Class *lazyClass* is removed from the model.

iii. Lifelines with reference to Class *lazyClass* are replaced with reference to Class *base*.

iv. *Midman* lifeline does not exist in the interaction for use case *uc*.

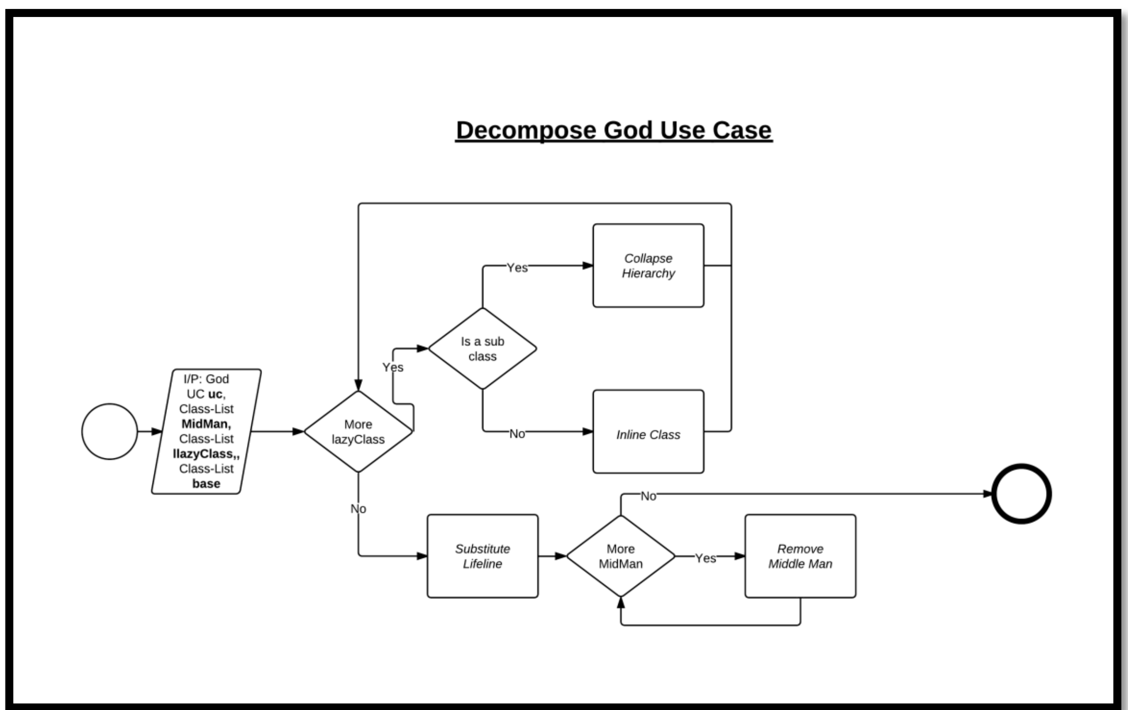


Figure 48 Decompose God Use Case Refactoring

5.4.2 (f) Example

Figure 49 and Figure 50 shows a subset of the model views from the NBS system that depicts the multiple personality model smell. The existence of a God use case *wireTransfer* (implements eleven classes) is identified on examination of the use case diagram and all the sequence diagrams associated with each use case. Closer examination of the sequence diagram for the *wireTransfer* use case yielded the existence of two middle man classes *TransferChannel* and *IBAN* and lazy classes *AccountInfo* and

InterBankTransfer. The existence of lazy classes was conformed from the class diagram of the system.

The *InlineClass* refactoring is initially applied to all the lazy classes and middle man classes identified by the model smell. These refactoring operations are listed below:

1. *InlineClass (BankServer, TransferChannel)*
2. *InlineClass (Accounts, AccountInfo)*
3. *InlineClass (Accounts, IBAN)*

Since the lazy class *InterBankTransfer* is a sub class of the *Transfer* Class, the *CollapseHierarchy (Transfer, InterBankTransfer)* is used to inline the class with its parent class. The *SubstituteLifeline* refactoring is then applied to substitute and redirect all messages that were initially communicated to/from the lazy classes. The refactoring operations are as follows:

1. *SubstituteLifeline (Transfer, InterBankTransfer)*
2. *SubstituteLifeline (Accounts, AccountInfo)*

Finally, the *RemoveMiddleMan (wireTransfer, IBAN)* & *RemoveMiddleMan (wireTransfer, TransferChannel)* refactoring is applied to remove the middle man lifelines and initiate direct communication. The refactored model views are shown in Figure 51 (structural and functional view) and Figure 52 (behavioral view).

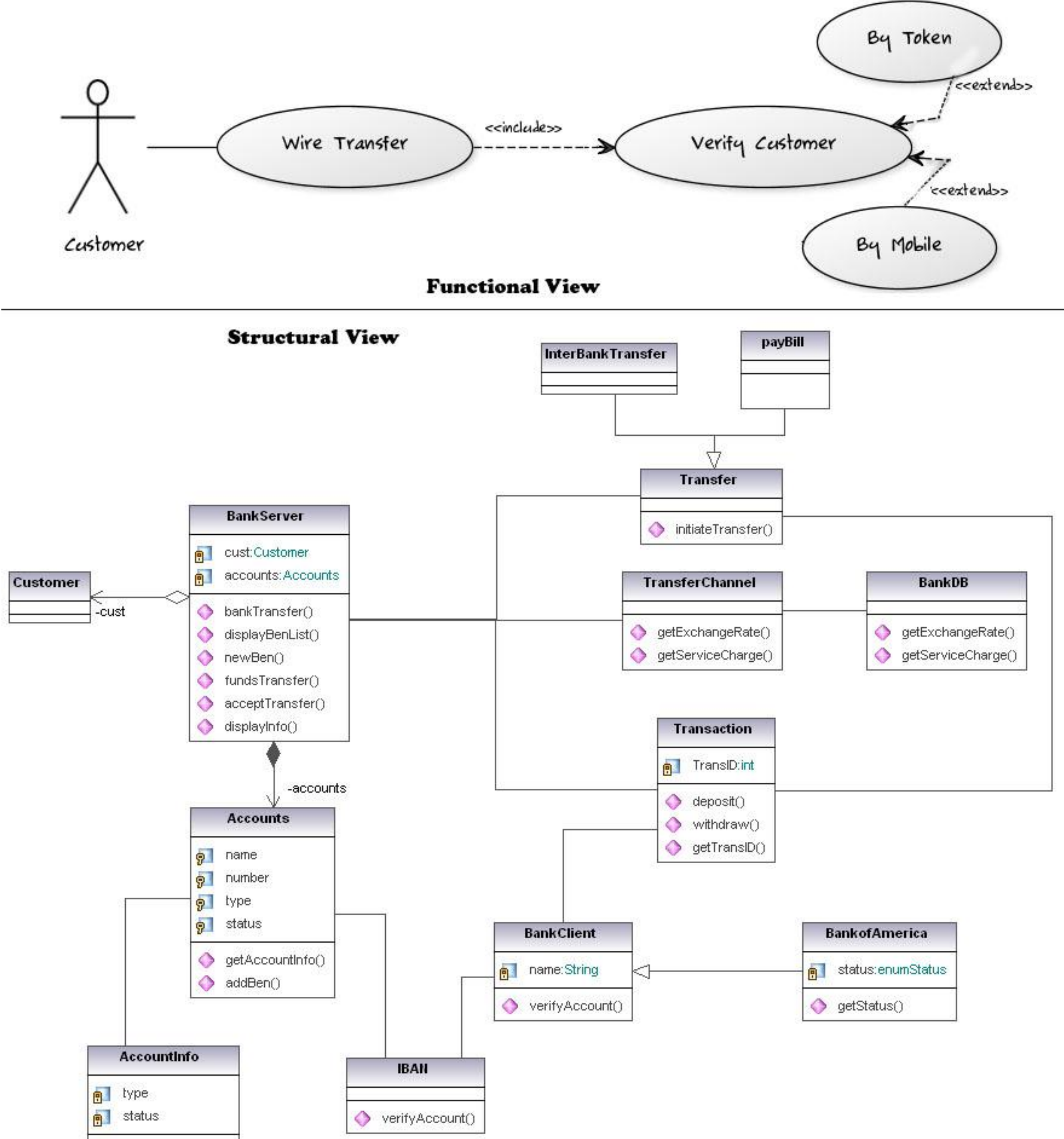


Figure 49 Excerpt of the NBS model views depicting Multiple Personality Smell

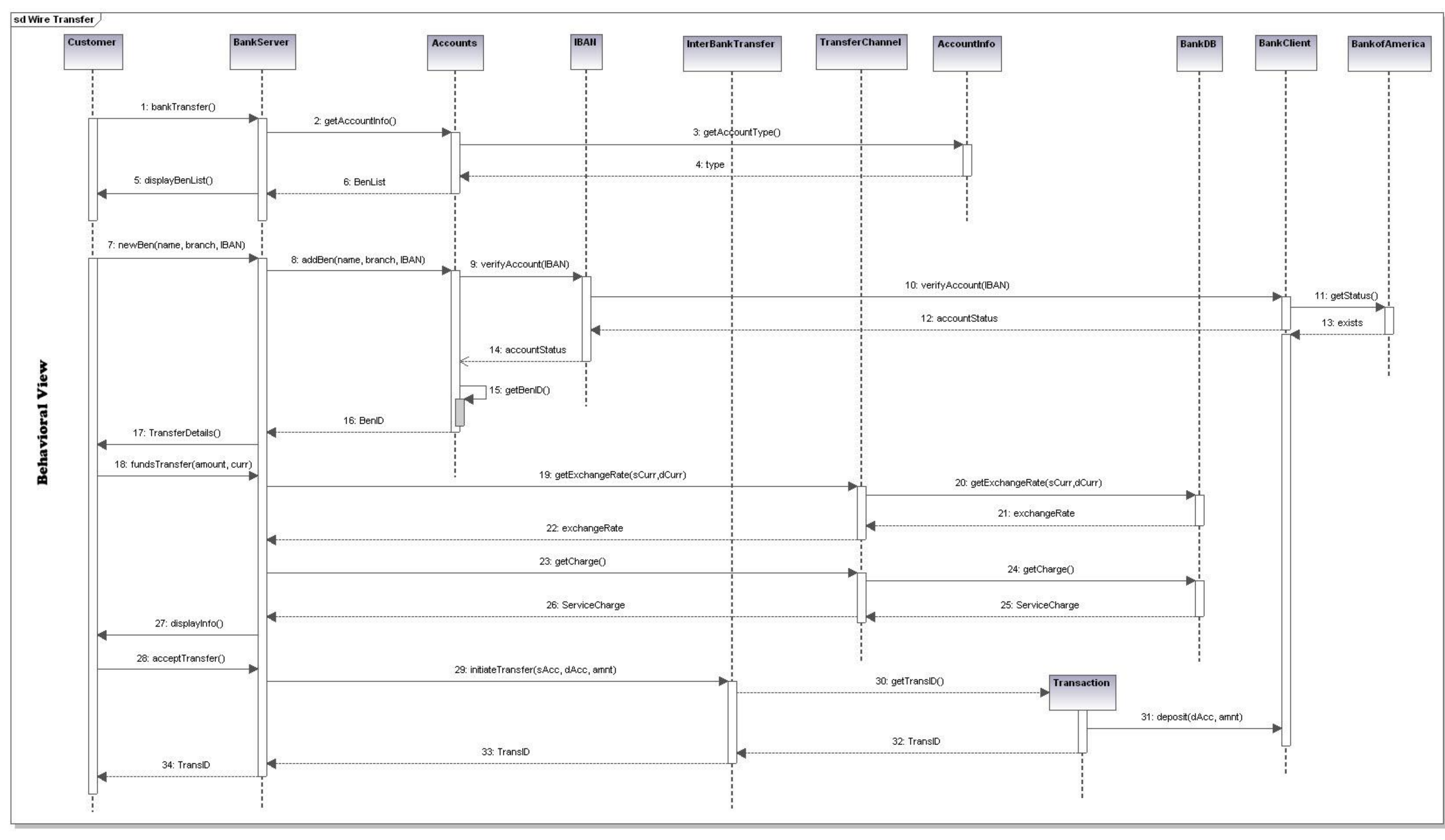


Figure 50 Excerpt of the NBS model view depicting Multiple Personality Smell

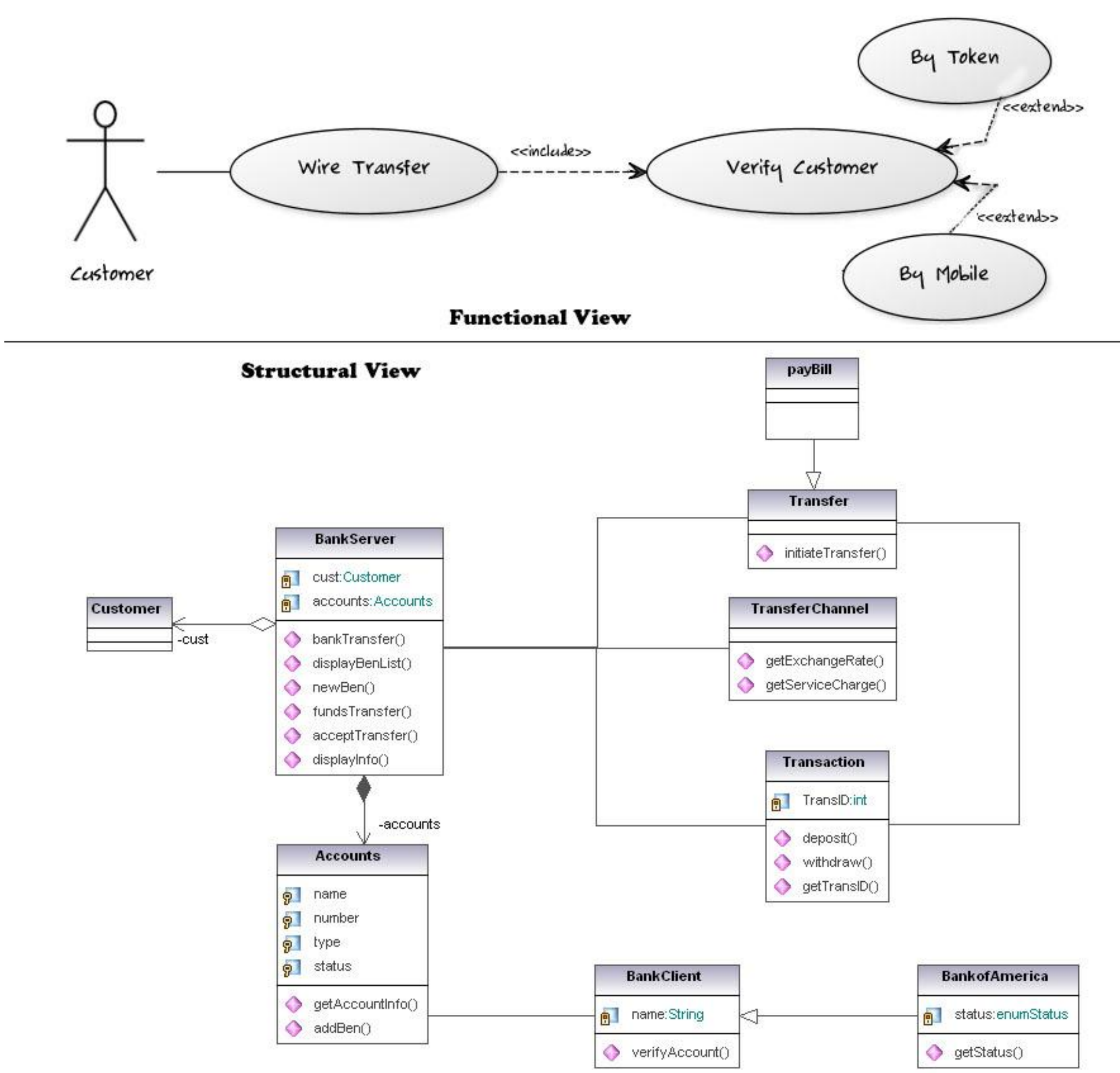


Figure 51 Excerpt of the NBS model views after refactoring

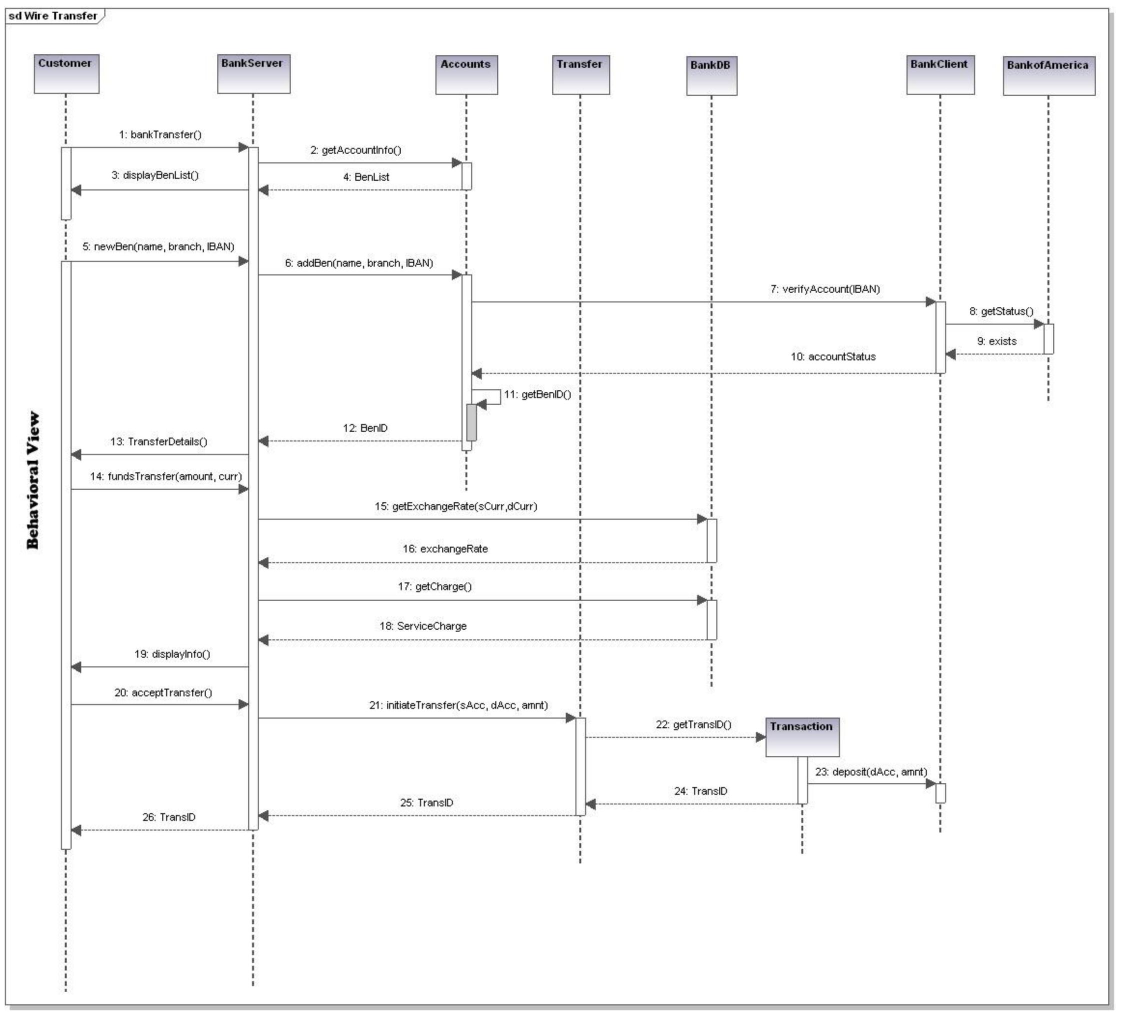


Figure 52 Excerpt of the NBS model view after refactoring

5.4.2 (g) Post Refactoring Model Improvement

The Behavioral View of each of the complex use cases from the Functional View is improved a lot in the refactored model because of the refactoring operation. The complexity of the use case and its interaction is reduced by removing additional classes such as lazy classes and middle man classes. Removal of these classes also reduces the number of transactions within the interaction model of the use case. Hence, it is safe to quote that the refactoring operation reduces the complexity and effort required to implement the use case and its behavior. The Structural View of the refactored model will show improved modularity by the removal of lazy classes that increase coupling and results in improved cohesion among the inlined classes.

5.4.2 (h) Side Effects

Since this refactoring targets lazy classes and delegating lifelines in order to reduce the complexity of the God Use case, it does not have any negative effect on the model. But some patterns make use of Delegating Classes to provide multiple views of information such as Model-View-Controller (MVC) pattern. It is difficult to detect and differentiate whether delegation in behavior is done to provide multiple views of model to a view or using lazy middle man classes to forward messages. Hence, one important side effect of the *Decompose God Use Case* model refactoring is its inability to differentiate between the above-mentioned functionalities provided by middle man classes in the integrated model.

5.4.3 Excessive Alternation

5.4.3 (a) Description

Excessive Alternation smell [430] occurs when the *extend* relationship between use cases is misused by the designers. The use case "extend" relationship allows additional behavior to be inserted into the base use case at a specific point known as *extension point.* One potential problem with use case modeling is to identify when to stop identifying alternative cases. Failure to identify this may lead to designers abusing the use case relationships like include and extend for functional decomposition. Building a non-trivial application, armed with the latest GUIs and event driven systems, there is a possibility to have a number of use cases that can produce essentially infinite number of usage scenarios. Too few use cases result in an inadequate specification, while too many use cases lead to functional decomposition. Limiting the analysis to the most obvious or important scenarios that generalizes to all use cases is a good approach. Fowler classified use cases into system use cases and user use cases [429]. System use cases are generic use cases that do not delve into many user-specifics. System use cases are more appropriate while modeling use cases, as they are useful in iteration planning and system testing. However, with every system use case, there are a number of user use cases hiding behind it waiting to be extended.

Another potential problem with use case modeling is the comprehension of the semantics of the extend relationship. In many cases the extend relationship is used in place of include or generalization relationship and even worse in place of pre and post conditions. This misuse can lead to a form of anti-pattern seen in Program Code known as the *Switch Pattern*. In this pattern, the base use case performs a few transactions in the beginning

and then keeps switching to other extension use cases conditionally. This scenario is similar to the switch construct used in some programming languages.

Although the existence of excessive alternation model smell can be identified by examining the functional view, in order to conform and to ensure automatic mitigation of this model smell requires the examination of other model views. Excessive alternation may lead to a complex use case model difficult to understand and maintain. In order to mitigate excessive alternation, common behavior from the base use case is extracted and inserted into all the extension use cases replacing the extension with an inclusion relationship.

5.4.3 (b) Rationale

El-Attar and Miller [264] included the abuse of the extend relationship for functional decomposition in their suite of use case anti-patterns. Although described, their approach did not provide an implementable detection and mitigation strategy. The use of multiple views for detection of excessive alternation not only provides means to identify misuse of extend relationship but also provide detail information to remove the identified smell in an automated manner.

Excessive alternation may lead to a complex use case model difficult to understand and maintain. A number of authors agree that the use of include and generalization relationship is much easier for most people to understand and use than the extend relationship [394, 433]. The misuse of extend relationship in place of utilizing the pre and post conditions of a use case could overwhelm and obscure other content in the diagram due to the presence of a number of extend arrows. *"Encapsulatable"* behavior at the

beginning of a use case can be separated and this can be replaced as a precondition of the use case. The availability of excessive alternation in a use case diagram not only complicates the functional view but also adds redundant behavior in the use case behavior and ignores a number of object-oriented advantages such as inclusion, polymorphism and inheritance in its structural view.

5.4.3 (c) Target Quality Improvements

- Use Case Maintainability
- Management of Behavior Complexity
- Reduction of Behavior Redundancy
- Modular Design
- Enhance Reusability

5.4.3 (d) Model Smell Detection Strategy

To ensure the applicability of this model smell in the integrated model, a use case with multiple extension points is selected. In order to quantify the number of extension points required in order to select the use case as a candidate for further examination, we use the "Number of Extension Point (NOEP) metric and its maximum acceptable value of 3 as provided by Gronback [288]. Based on this suggestion, any use case with three or more extension points is used for further examination for applicability of this model smell.

The behavior of the selected use case is then examined to identify whether a *"switch pattern"* exists. In order to explain this, we first divide the behavior of a use case model into three sections as shown in Figure 53. These sections are the preamble, body and post. Hence, a base use case with a preamble length of greater than two, a body with only an

"alt" fragment and post length equal to zero is considered to depict excessive alternation model smell.

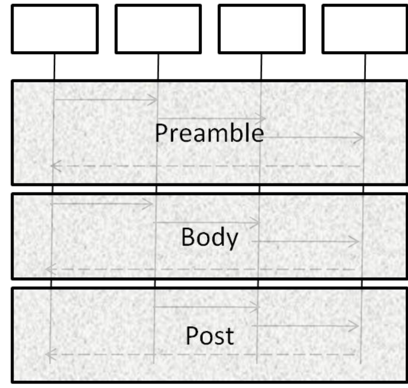


Figure 53 Use Case Behavior (Sequence Model) divided into three sections

The pseudo code given below describes the steps required for automated detection of the excessive alternation model smell.

```
: ALGORITHM: EXCESSIVE ALTERNATION
: start
:   read Model
:   for (each use-case in the Model)
:     read UC
:     if (# of extension-points in UC is >= 3)
:           if (# of preamble steps in UC > 2) and (switch-pattern(body) is true)
:                                                    and (# of post steps is = 0)
:             return UC
: stop
```

5.4.3 (e) Model Refactoring Mechanics

Name: Substitute Excessive Extensions

Parameters: Usecase *uc,* String *newUC* where,

- *uc* is the Base Use Case
- *newUC* is the temporary name for a new use case

Preconditions:

i. The name of the new use case (*newUC*) does not conflict with the name of an existing use case within the model.

Mechanics:

1. In order to use the same name as the base use case, we first need to rename the base use case. *Rename UseCase* refactoring is initially used to rename the use case to any other name.
2. *Create UseCase* refactoring is used to create a new use case with the same name as the base use case.
3. *Extract Fragment* refactoring is then used on the base use case sequence diagram to extract the preamble transactions into the newly created use case.
4. If the operand of "alt" fragment in the body of the use case behavior is not an Interaction Use Fragment, then first use *Extract Fragment* refactoring to extract the steps in the operand into a new use case.
5. *Insert Fragment* refactoring is then used to add the common behavior in the beginning of all the extension use case sequence diagrams and the one created in step 4 (if applicable).
6. *Add Inclusion* refactoring is used to add inclusion between the base use case and newly created use cases in step 4 and the extension use cases of the previous base use case.

7. *Move Actor Reference* refactoring is used to add uses relationship from the actor to all the previous extension use cases. The actor's relationship to the base use case still remains in the model.
8. *Isolate UseCase* refactoring is used to remove all relationships and actor references from the previous base use case.
9. Finally, *Delete UseCase* refactoring is used to the remove the old base use case renamed in step 1.

Figure 54 shows the ordering of the composite refactoring *Substitute Excessive Extensions*.

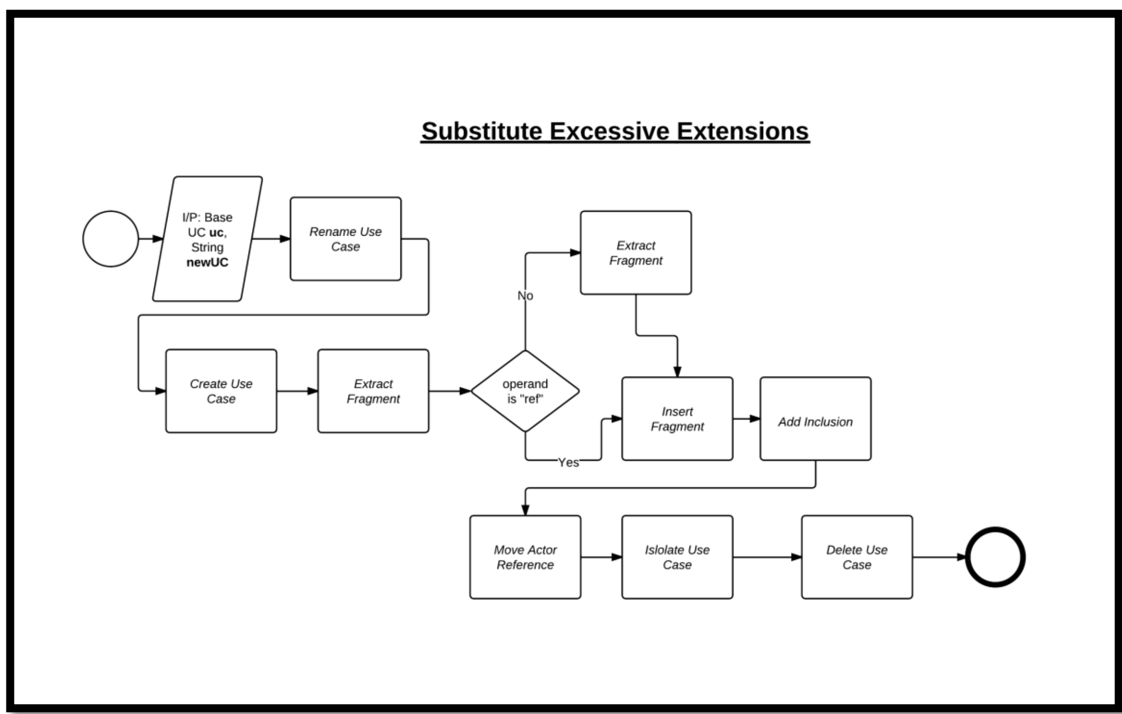


Figure 54 Substitute Excessive Extensions Refactoring

Post Conditions:

i. A use case with name newUC does not exist in the model.

ii. All extension Points within the use case uc are removed.

iii. There are no extend relationship between uc and other use cases in the model.

5.4.3 (f) Example

Figure 55 and Figure 56 shows a subset of the model views from the NBS system that depicts the excessive alternation model smell. On examination of the use case diagram, the existence of the use case *Login* was identified having more than two extension points. Closer examination of the sequence diagram for the *Login* use case revealed the existence of a switch pattern (more delegations than transactions). Since all the lifelines in the *Login* sequence diagram were subsets of the lifelines in the sequence diagram for the extension use case, the *login* sequence diagram was added using a *"ref"* combined fragment in all its extension sequence diagrams.

The *RenameUseCase (Login, newUC)* refactoring and *CreateUseCase (Login)* is initially applied to rename the *Login* use case with a temporary name *newUC* and create a new one with the same name to preserve its name. *ExtractFragment (newUC, startPoint, endpoint, Login)* refactoring is then used to extract the preamble part of the use case into the newly created *Login* use case.

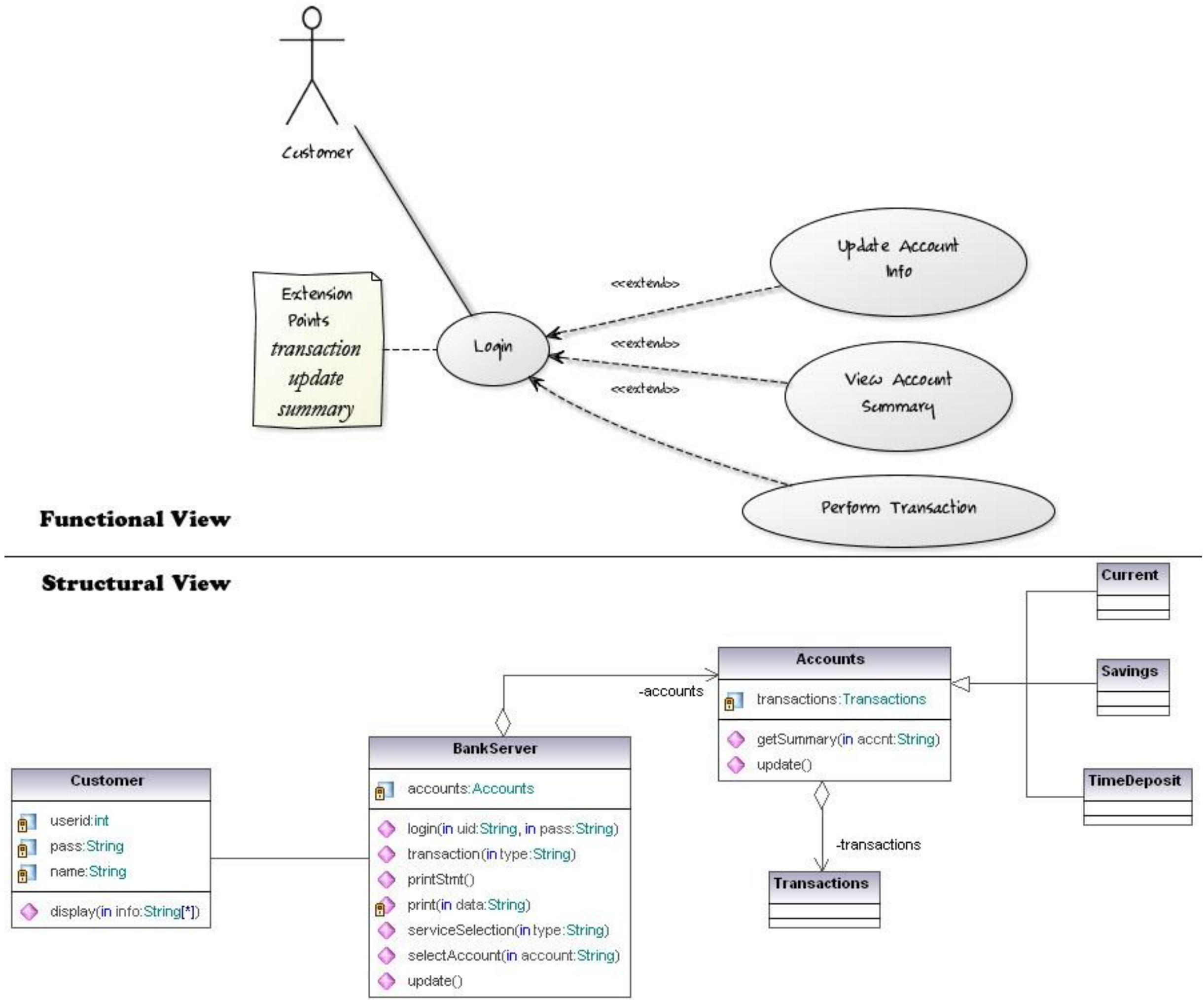


Figure 55 Excerpt of the NBS model views depicting Excessive Alternation Smell

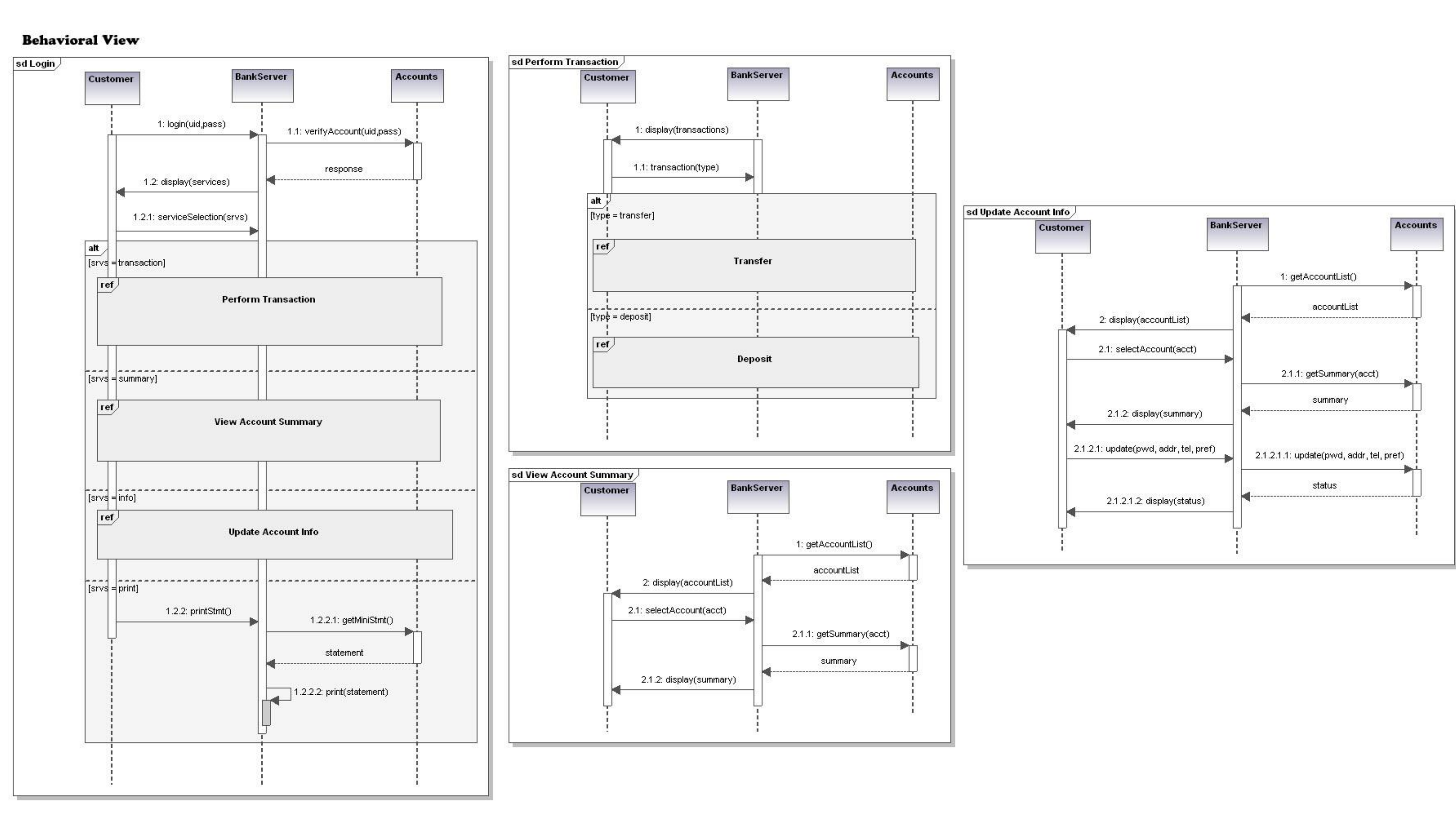
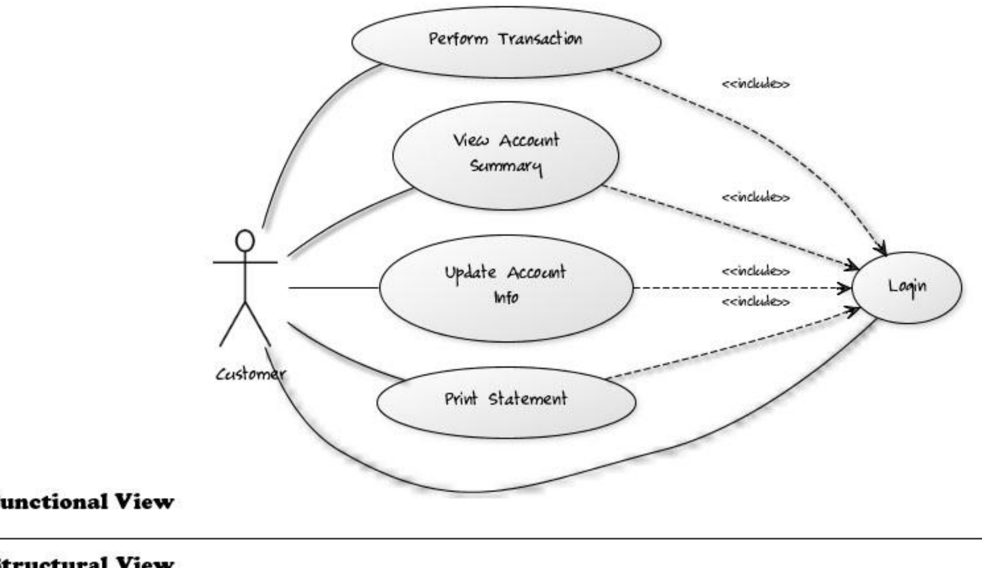


Figure 56 Excerpt of the NBS model views depicting Excessive Alternation Smell



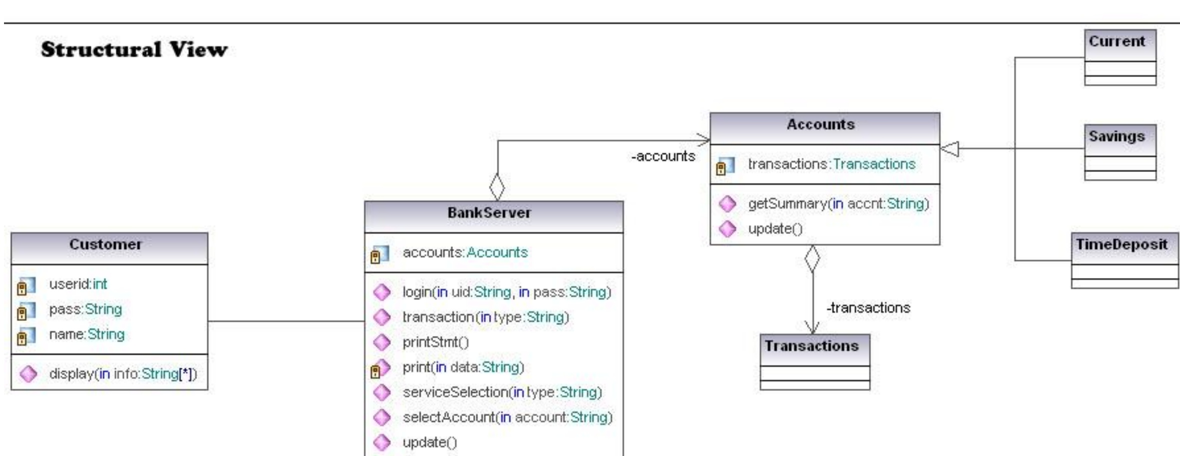


Figure 57 Excerpt of the NBS model views after refactoring

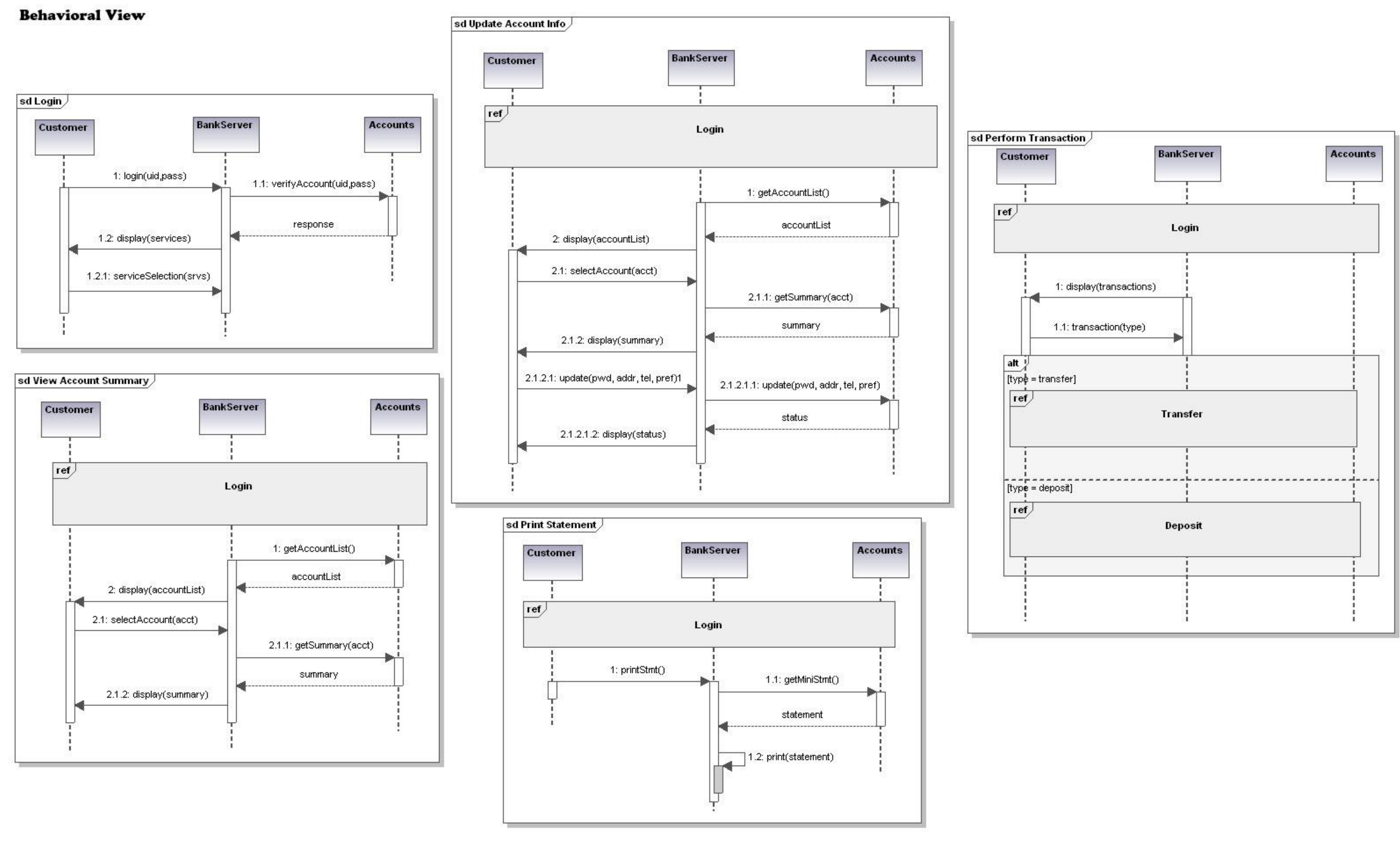


Figure 58 Excerpt of the NBS model views after refactoring

For the operand without the “alt” operand, *ExtractFragment (newUC, startPoint2, endPoint2, Print Statement)* refactoring is performed. The common behavior extracted earlier into the Login use case is then added to all the extension use cases using the *Insert Fragment* refactoring. The refactoring operations are as follows:

1. *InsertFragment (Perform Transaction, Login)*
2. *InsertFragment (View Account Summary, Login)*
3. *InsertFragment (Update Account Info, Login)*
4. *InsertFragment (Print Statement, Login)*

AddInclusion refactoring is then performed to add inclusion relationship between *Login* and the newly extracted use case and other “ref” fragment use cases. The refactoring operations are as follows:

1. *AddInclusion (Login, Print Statement)*
2. *AddInclusion (Login, Perform Transaction)*
3. *AddInclusion (Login, View Account Summary)*
4. *AddInclusion (Login, Update Account Info)*

Then the *MoveActorReference* refactoring is applied to move all the actor references from the *newUC* use case to the newly created base use cases. The refactoring operations are as follows:

1. *MoveActorReference (newUC, Perform Transaction)*
2. *MoveActorReference (newUC, View Account Summary)*

3. *MoveActorReference (newUC, Update Account Info)*
4. *MoveActorReference (newUC, Print Statement)*
5. *MoveActorReference (newUC, Login)*

Finally, the *IsolateUseCase (newUC)* refactoring is used to remove all relationships from the *newUC* and *DeleteUseCase (newUC)* refactoring is performed to remove the use case from the model. The refactored model views are shown in Figure 57 and Figure 58.

5.4.3 (g) Post Refactoring Model Improvement

A use case that spends less time performing its own tasks and switches from one use case to the other throughout its lifetime is considered a bad form of behavior distribution. Not only it complicates the functional view with a number of extension points and extends relationships, it also increases the complexity of the behavior by magnifying its Cyclomatic Complexity (result of increase in the number of branch points). Identifying and substituting these cases with simpler relationships like "include" enhances comprehension and maintenance of the functional view of the system and alleviates the complexity of the behavioral view of the model. These in turn opens commonality features to be considered for enhancing the modularity of the structural view of the model.

5.4.3 (h) Side Effects

Excessive Alternation done because of identifying as many alternate scenarios as possible for a system under design can be considered a good quality practice. However, overdoing can complicate the model and affect other aspects of the system. Although extensions are

problematic, they do provide the ability for a base use case to begin execution of the extension use case from a specified step within the extension use case as opposed to inclusion where execution must start at the first step. Removing this relationship and substituting with the include relationship will not allow designers to benefit from this attribute of the extend relationship. Another side effect of this refactoring is the increase in the number of use cases associated with an actor. But since the new associations fully describe what the actor can do with the system, it can be justified [434]. If the actor association with the use cases is due to improper depiction of actor role in the system, the Spider's Web model smell and its associated refactoring can be applied (see Section 5.4.5).

5.4.4 Undue Familiarity

5.4.4 (a) Description

One of the main principles of Object Oriented Design is Encapsulation. This means that the implementation details are hidden behind the definition of the object. When objects violate encapsulation, the model smells of *Undue Familiarity*. Undue Familiarity is a model smell that occurs when one object knows more about another object than it is required to. This model smell is mostly similar to the *Inappropriate Intimacy* Smell found in Source Code.

Classes in UML class diagram are related to each other by three major relationships: *Generalization, Aggregation* and *Association*. Out of these, association is the only relationship that can be bi-directional. Although a bi-directional association between classes in a class diagram does not indicate the existence of the Undue Familiarity model

smell, it can be considered as the point of origin for further investigation. Studying the mode of interaction between these classes will provide more information as to whether objects of one class know more about the objects of the other class. This in turn results in a complex use case with more than required messages and classes implemented by the use case and a use case model with inappropriate behavior distribution.

5.4.4 (b) Rationale

Undue Familiarity model smell results in a system design that is unstable and less reusable. Because of this model smell, the design is more likely to have changes in one part of the system impact another part of the system. For instance, if the user interface has the knowledge that its data access layer makes use of a particular form of data storage, then the data access layer cannot change without potentially making changes throughout the user interface. Hence, the user interface cannot run or be tested without a connection to the database to populate the used form of data storage. Therefore, this inappropriate knowledge makes the system more fragile. Simple changes create breaking changes. Reusability of objects is reduced as they assume that the intimate information in the other familiar objects remain the same.

The existence of inappropriately familiar classes within a class diagram not only obscures the structural view but also increases the message communication frequency in the behavioral view and ignores a number of model design primitives such as behavior distribution and use case complexity in its functional view.

5.4.4 (c) Target Quality Improvements

- Use Case Maintainability and Complexity

- Management of Behavior Complexity
- Modular Design/Coupling
- Model Maintainability, Stability & Reusability

5.4.4 (d) Model Smell Detection Strategy

To ensure the applicability of this model smell in the integrated model, pairs of bi-directionally associated classes are identified. An association with both its ends as owner-ends is referred to as a bidirectional association. For each of these pairs, examine the interaction parts of all the use cases they are part of and their mode of interaction within those interaction model elements. Message interactions between two classes can be termed inappropriate if they access data and methods from each other frequently. In order to identify if message passing between two sets of lifelines is inappropriate, we define two types of messages: *Access* and *Update*. An access message is a “getter” method requesting data from the other class. A return statement in the interaction diagram usually follows this message. An update message is a “setter” method updating data in the other class. Update messages are parameterized messages. Hence, message passing between two classes is termed inappropriate if both classes involved perform update and access message exchanges. If message-passing frequency between these two classes is inappropriate and these pairs occur in interaction parts of more than one use cases, then undue familiarity model smell exists in the integrated model.

The pseudo code given below describes the steps required for automated detection of the Undue Familiarity model smell.

```
: ALGORITHM: UNDUE FAMILIARITY
: start
:   read Model
:   for (each association in the Model)
:     read Assoc
:     if (ends of Assoc are both owned)
:           c1 = one end of the Assoc
:           c2 = other end of the Assoc
:       for (each use case in the model)
:         read UC
:             diff = (lifelines in UC) ∩ (set{c1,c2})
:             if (diff != empty)& (mesg freq between c1 & c2 is inappropriate)
:                   counter++;
:             end if
:           end for
:           if (counter > 1)
:                  return Assoc
:           end if
:         end if
:   end for
: stop
```

5.4.4 (e) Model Refactoring Mechanics

Name: *Break Intimate Elements*

Parameters: Association *assoc,* Class *src,* Class *tar,* String *newCase*
where,

- *assoc* is the intimate association relationship
- *src* is one end of the association relationship *assoc*
- *tar* is the other end of the association relationship *assoc*
- *newCase* is the name of a new use case if similar fragments are extracted

Preconditions:

i. The association relationship *assoc* is bi-directional.

ii. The name *newCase* does not conflict with the name of an existing use case within the model.

Mechanics:

The mechanics of this refactoring is based on the nature of the intimate elements. Hence, the solution is divided into two parts:

1. If the nature of the association is breakable i.e. if the messages and data items involved between the associated classes is exclusive to these classes and not invoked by other associations to the *tar* class.
    a. For each message access and update message from the *src* class to the *tar* class, *Move Attribute* and *Move Operation* refactoring is applied. This is repeated across all interactions involving communication between the *src* and *tar* classes. If Move Operation is successful, *Remove Message* refactoring removes the message call between the classes involving the moved operations
    b. If the *tar* class is empty after the previous refactoring application and has no relationship with other classes in the class model, *Remove Empty Class* refactoring is applied.
    c. Since all message incident to the removed class are included in the *src* class, *Remove Lifeline* refactoring is applied to the *tar* lifeline across all interactions.
2. If the nature of the association is unbreakable, i.e. if the messages and data items involved between the associated classes is not exclusive and are invoked by other associations.

a. *Extract Fragment* refactoring is then used on the frequent message exchange fragment of the interaction if the same message exchange pattern appears in other interactions of the system. This extracted fragment is added into a new use case *newCase.*

b. *Add Inclusion* refactoring is used to add inclusion between the base use cases and the newly created use case *newCase* in step 2a.

Figure 59 shows the ordering of the composite refactoring *Break Intimate Elements*.

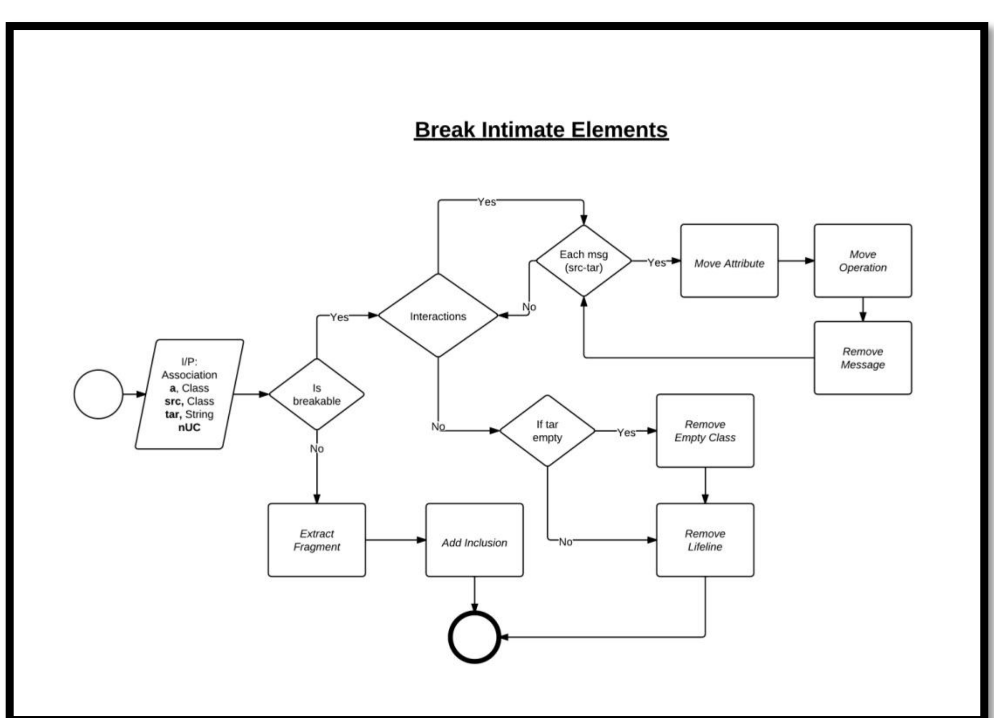


Figure 59 Break Intitmate Elements Refactoring

Post Conditions:

Due to the alternative nature of the refactoring operation, no post-conditions are specified. In case the first path is traversed, Class *tar* may not be part of the

refactored model. In case the second path is traversed, Use Case *newCase* exists within the refactored model.

5.4.4 (f) Example

Figure 60 shows a subset of the model views from the NBS system that depicts the undue familiarity model smell. The association pair between the *Accounts* and *Credit* class was found to be bi-directional and further investigated for inappropriate interactions within the sequence model. These pairs appeared within two interactions *POS Payment* and *Increase Limit.* Closer examination of the identified interactions revealed that message passing between these two classes was inappropriate as both classes performed update and access message exchanges between each other.

The following MoveAttribute and MoveOperation refactorings were applied to move the familiar attributes and operations to the source class.

1. *MoveAttribute (Accounts, Credit, limit)*
2. *MoveAttribute (Accounts, Credit, outstanding)*
3. *MoveOperation (Accounts, Credit, increaseLimit)*
4. *MoveOperation (Accounts, Credit, reimburseLimit)*

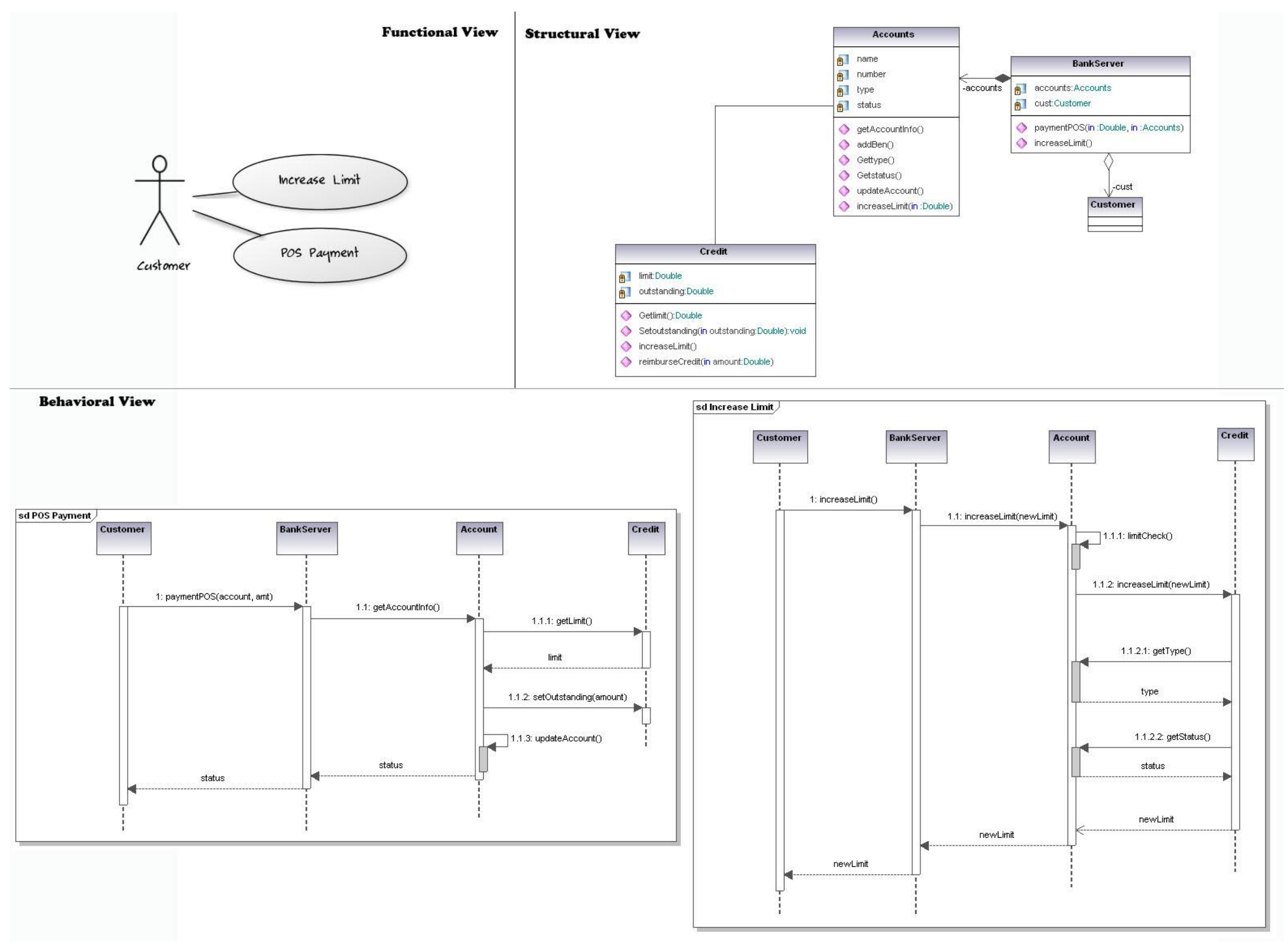


Figure 60 Excerpt of the NBS model views depicting Undue Familiarity Smell

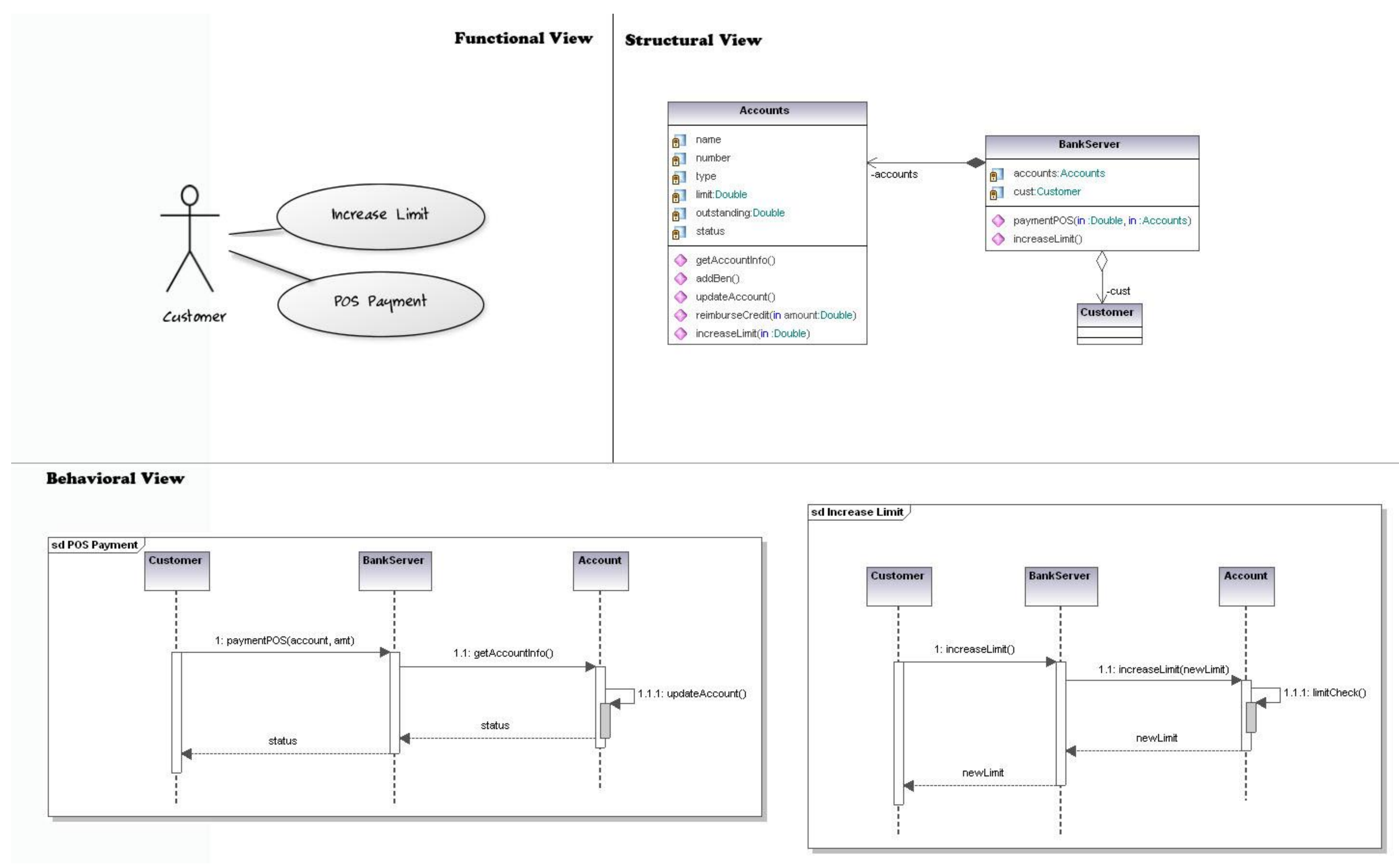


Figure 61 Excerpt of the NBS model views after refactoring

For each operation moved into the *Accounts* class, the *Remove Message* refactoring was applied to remove the message interaction between the two classes. The following set of refactorings was applied to the interactions of *POS Payment* and *Increase Limit.*

1. *RemoveMessage (Accounts, Credit, getLimit)*
2. *RemoveMessage (Credit, Accounts, limit)*
3. *RemoveMessage (Accounts, Credit, setOutstandingAmount)*
4. *RemoveMessage (Accounts, Credit, IncreaseLimit)*
5. *RemoveMessage (Credit, Accounts, getType)*
6. *RemoveMessage (Accounts, Credit, type)*
7. *RemoveMessage (Credit, Accounts, getStatus)*
8. *RemoveMessage (Accounts, Credit, status)*
9. *RemoveMessage (Accounts, Credit, newLimit)*

The *Remove Lifeline (Credit)* is then applied to the isolated *Credit* lifeline in both the *POS Payment* and *Increase Limit* interaction. Since the class *Credit* became empty as a result of the move operations, the *Remove Empty Class (Credit)* is applied to remove it from the structural view. The refactored model views are shown in Figure 61.

The example presented here for Undue Familiarity is one instance of the model smell. Hence, the functional view was not modified.

5.4.4 (g) Post Refactoring Model Improvement

When objects are properly encapsulated, the model as a whole is more pliant to change. But when objects go against encapsulation, the model becomes more difficult to change. Problems in one object propagate to other objects throughout the system and changes in one object require changes in other objects.

Application of this refactoring reduces intimacy between overly intimate classes by either combining them or moving features where they are used most often. This ensures encapsulation principle of Object Oriented Programming and hence reduces coupling between classes and makes the model more reusable, maintainable and easier to update. The complexity of the use case and its interaction is also reduced by removing additional transactions within the interaction model of the use case. Behavior and functionality is properly distributed in the functional view of the model. Hence, it is safe to quote that the refactoring operation reduces the complexity and organization of the use cases within the model.

5.4.4 (h) Side Effects

Inappropriate Intimacy is a result of improper behavior distribution within the software model beginning from its functional view in high-level design phase and propagating to its structural view in low-level design phase. Reduction of this intimacy will not cause any side effects within the design model, as it was a result of improper behavior distribution.

5.4.5 Spider's Web

5.4.5 (a) Description

Lilly [434] provided a list of the top ten pitfalls that occur when using use cases for modeling real time projects. The same author when discussing the actor-to-use-case relationship suggested the name *Spider's Web*. This model smell is derived from the same concept. This model smell occurs when an actor in the use case model has multiple responsibilities (i.e. associated with a number of use cases) so that the view looks like a spider's web. A pictorial representation of the spider's web model smell in the form of a sample use case model is illustrated in Figure 62.

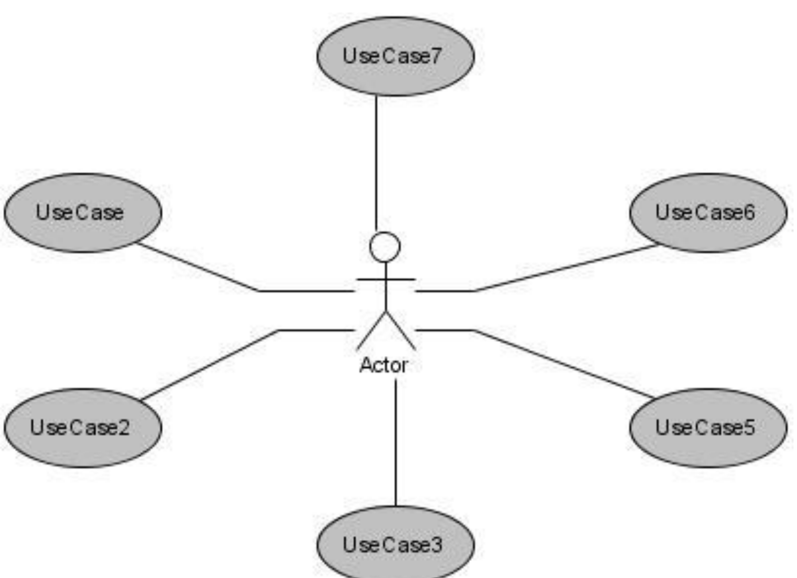


Figure 62 Sample use case model depicting Spider's Web Model Smell

An actor initiating multiple use cases is either an indication that the actor is defined too broadly [391] or inappropriate granularity of use cases. In case of improper actor identification, the behavior of actor participation in the sequence diagram and its association with other entity classes (since actors are realized as entity classes in the

detailed design phase) must be examined. For instance, a *User* actor is very general and is usually associated with a large number of use cases. In case of inappropriate granularity of use case composition, use case behavioral view must be examined to ensure the fragmented use case is non-trivial, does not describe an internal system process and provides a usable output value to the system's user. For instance, highly fragmented use cases usually describe interactions between the system and the actor rather than the actual goal.

Although the existence of the spider's web smell can be visually identified through the use case diagram, it cannot be classified as a model smell unless all views are examined to detect the existence of improper actor classification and use case decomposition.

5.4.5 (b) Rationale

Spider's Web model smell may lead to a complex use case model that is difficult to understand and maintain. The existence of spider's web model smell in the use case model is also an indication of God Class existence in the structural view. Since one of the effects of spider's web model smell is the improper fragmentation of use cases, the total number of sequence diagrams described by the system increase causing duplication and unnecessary implementation. Hence, the availability of spider's web in a use case diagram not only complicates the functional view but also adds unnecessary redundant behavior in sequence diagrams and may result in behaviorally rich entity classes that realize the actors involved in the model smell.

5.4.5 (c) Target Quality Improvements

- Use Case Maintainability

- Reduction of Behavioral Redundancy
- Modular Design

5.4.5 (d) Model Smell Detection Strategy

To ensure the applicability of this model smell in the integrated model, an actor associated with multiple use cases is selected. In order to quantify the number of use cases required in order to select an actor as a candidate for further examination, we use the *"Number of Use Cases per Actor (NUCA)"* metric and its maximum threshold UP_{NUCA}. Since this upper limit threshold value is not available in the literature, we consider actors that are associated with more than 30% of the total use cases implemented by the system.

The behavior of the selected actor is then examined to identify whether the actor represent a user type or a role. Using actors to represent types rather than roles results in compromising usability and stability of the use case model [391]. In order to identify whether an actor is representing multiple roles within the system, a behavior signature is associated with each use case associated with an actor. A behavior signature is a set of lifelines interacting with the actor to realize the use case functionality in the sequence diagram. Use cases associated with the actor are then classified based on behavior signature similarity. Two signatures are also considered similar if the exclusion lifelines are child classes of the same parent class. If an actor is associated with multiple signatures, the existence of the Spider's web model smell is confirmed and is need of refactoring.

The pseudo code given below describes the steps required for automated detection of the spider's web model smell.

```
: ALGORITHM: SPIDER'S WEB
: start
:   read Model
:   for (each actor in the Model)
:     read A
:     if (# of use-cases for A is >= UP_NUCA)
:        for (each use-case associated with A)
:           read UC
:           for (each lifeline associated with UC)
:              read Life
:                     if (Life is a child class)
:                                   sig = sig U {parent(Life)}
:                       else
:                                   sig = sig U {Life}
:                       end if
:           end for
:           if (first use-case)
:                       base-sig = sig
:           end if
:           if (sig != base-sig)
:                       diff = diff +1
:           end if
:        end for
:      end if
:      if (diff >= 2)
:           return A
:      end if
:   end for
: stop
```

5.4.5 (e) Model Refactoring Mechanics

Name: Redistribute Responsibility

Parameters: Actor *a,* List *actorNames,* List *ucNames* where,

- *a* is the Actor with multiple roles
- *actorNames* is the list of new actors to distribute the use cases

- *ucNames* is the list of the use cases to be associated with each new actor in the *actorNames* list.

Preconditions:

i. The name of the new actors (*actorNames*) does not conflict with the name of existing actors within the model.

ii. The list *ucNames* includes all use cases assigned to Actor *a*.

Mechanics:

1. *Split Actor* refactoring is used to split actor *a* into the number of actors mentioned in the *actorNames* list.
2. Each new actor is associated with a subset of use cases assigned to the main actor *a*. Since Split Actor refactoring in the previous steps associates all use cases associated with the main actor to the newly created actor, unwanted associations are removed using the *Remove Actor Reference* refactoring based on the list provided by *ucNames*.
3. *Isolate Actor* refactoring is applied to main actor *a* to isolate it from the use case model.
4. *Delete Actor* refactoring is used remove the actor from the system.
5. If the lifeline for actor *a* has an incoming call event in the interaction, *Create Sub Class* refactoring is performed to create a new class based on the new actor to which the use case is assigned.
6. *Push Down Operation* refactoring is performed to move the incoming message to the newly created specialized class for the actor *a*.

7. Finally, *Substitute Lifeline* refactoring is then used to remove all references to the old actor from respective interaction diagrams and replace it with the new actor based on the new actor-use case relationship.

Figure 63 shows the ordering of the composite refactoring Redistribute Responsibility.

Post Conditions:

i. Actor with name *a* does not exist in the model.

ii. Lifelines with reference to Actor *a* are replaced with reference to actors in *actorNames*.

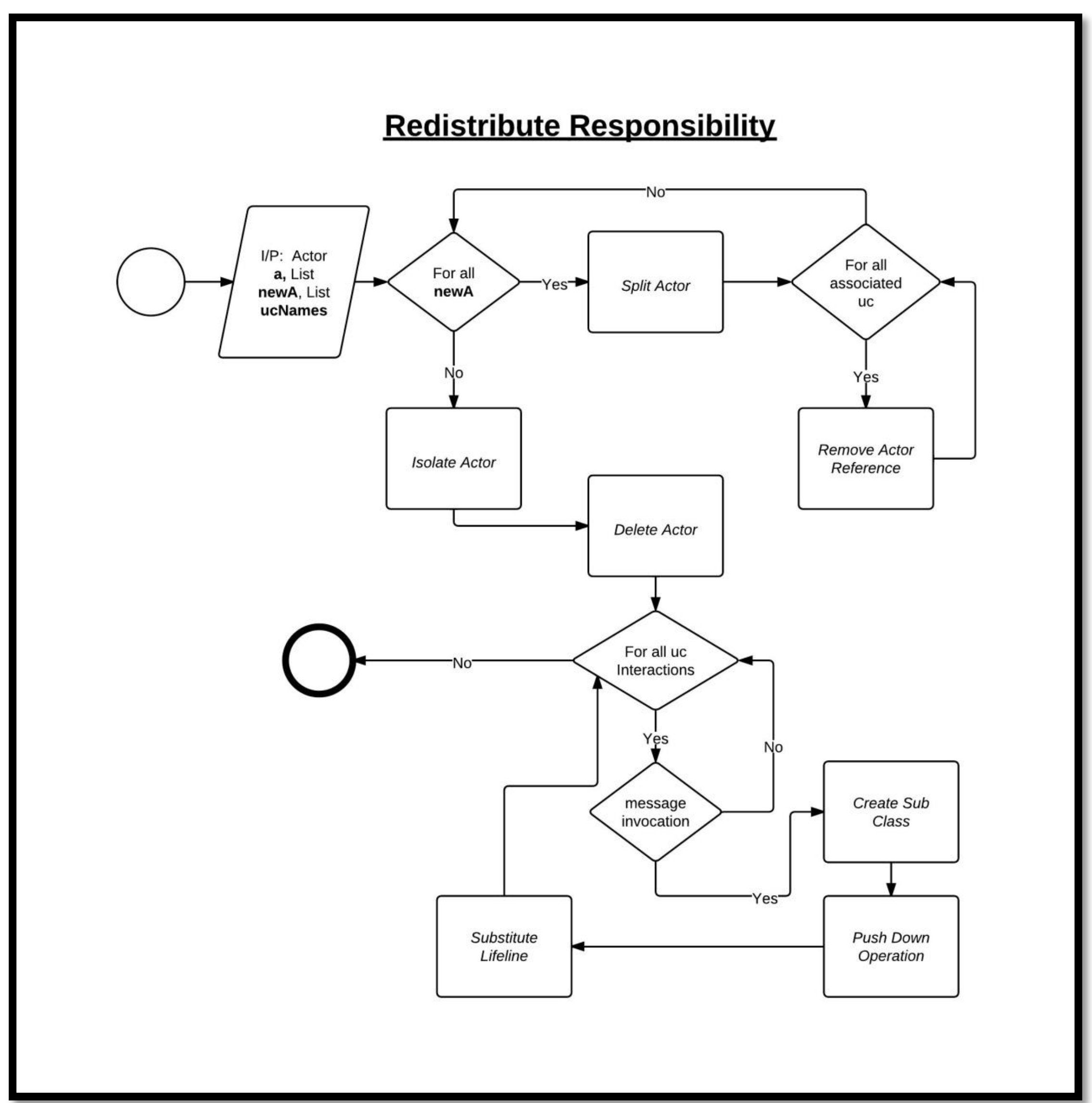


Figure 63 Redistribute Responsibility Refactoring

5.4.5 (f) Example

Figure 64 shows a subset of the model views from the NBS system that depicts the spider's web model smell. On examination of the use case diagram, the existence of the actor *Operator* was identified having a number of use case associations.

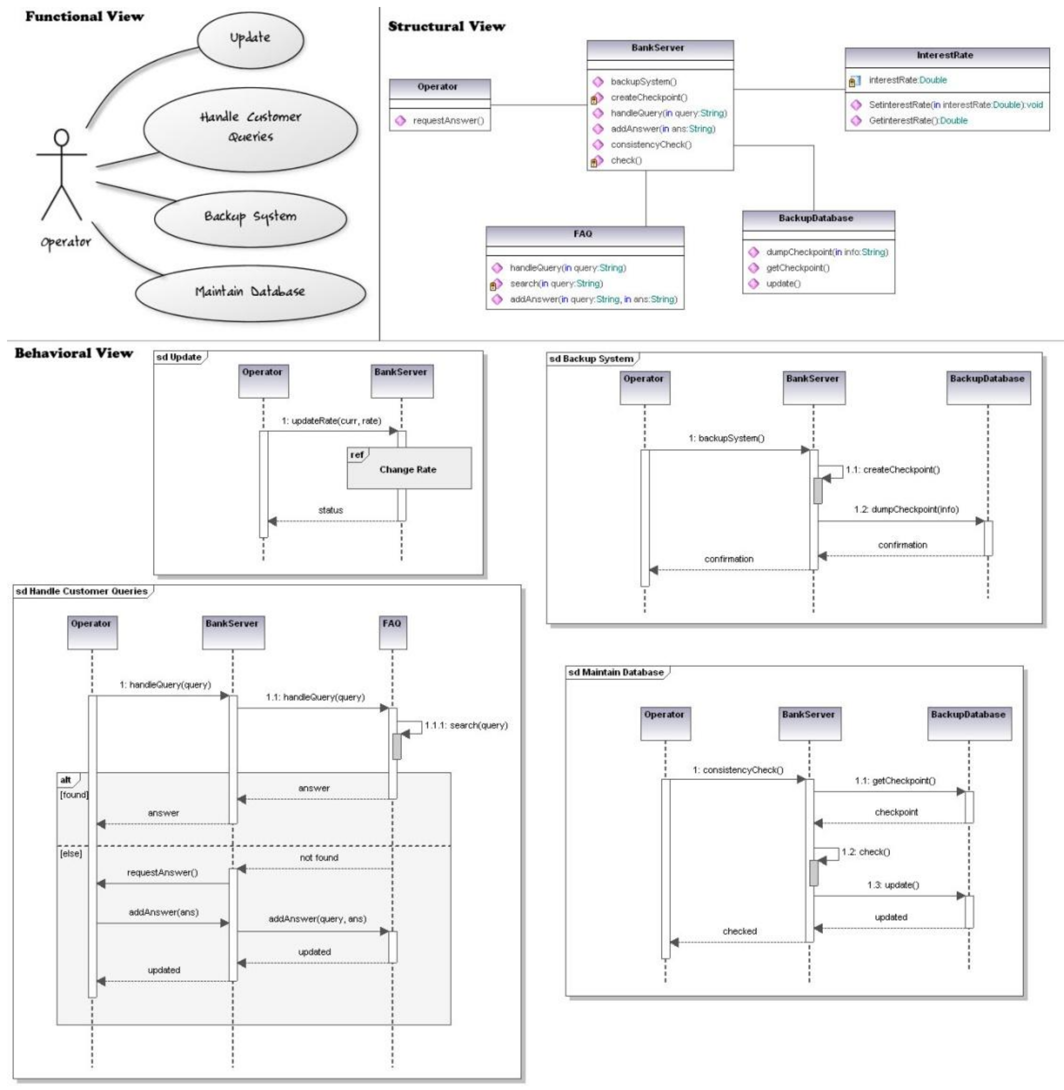


Figure 64 Excerpt of the NBS model views depicting Spider's Web Smell

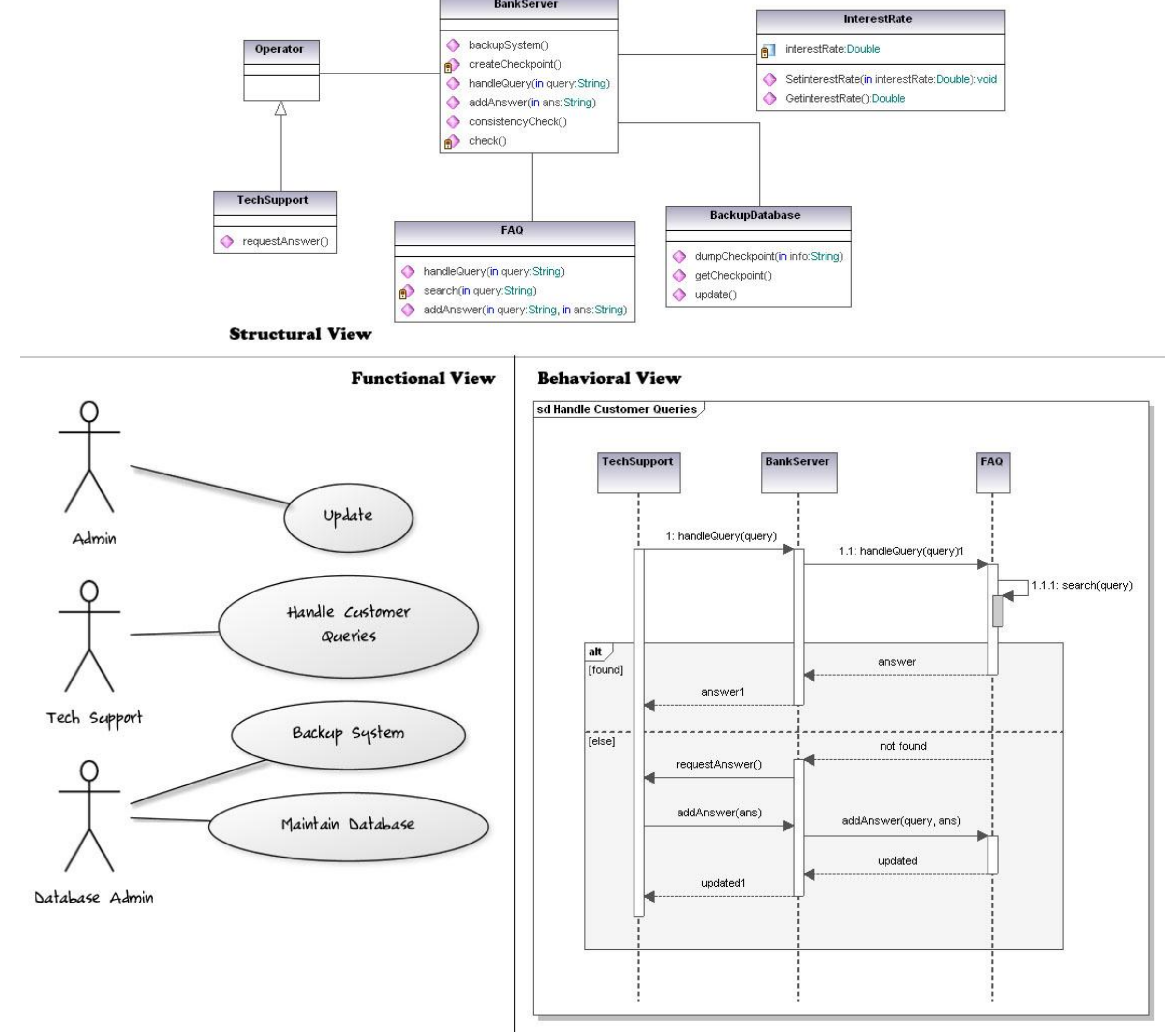


Figure 65 Excerpt of the NBS model views after refactoring

In order to identify whether the actor is representing multiple roles within the system, its behavior signature is created for each use case it is associated to based on the information from the behavioral view. Following are the four signatures in line with the four use cases *Operator* is associated to in the use case diagram.

1. Update: {BankServer}
2. Backup System: {BankServer, BackupDatabase}
3. Customer Queries: {BankServer, FAQ}
4. Maintain Database: {BankServer, BackupDatabase}

Based on the signatures, it was identified that Operator was involved with more than one role in the system. Hence, the availability of the Spider's Web model smell is confirmed. In order to remove this smell, initially the *Split Actor* refactoring is applied based on the number of different signatures found. The following refactoring operations are hence applied to the model.

1. *SplitActor (Operator, Admin)*
2. *SplitActor (Operator, Tech Support)*
3. *SplitActor (Operator, Database Admin)*

Since the Split Actor refactoring associated all new actors with the associations of the base actor, the *Remove Actor Reference* refactoring is applied to assign the new actors to their specific use cases. This is based on the information available from the behavioral view. The following refactoring operations are hence applied to the model.

1. *RemoveActorReference (Admin, Handle Customer Queries)*
2. *RemoveActorReference (Admin, Backup System)*
3. *RemoveActorReference (Admin, Maintain Database)*
4. *RemoveActorReference (Tech Support, Update)*
5. *RemoveActorReference (Tech Support, Backup System)*
6. *RemoveActorReference (Tech Support, Maintain Database)*
7. *RemoveActorReference (Database Admin, Update)*
8. *RemoveActorReference (Database Admin, Handle Customer Queries)*

Since the use cases are appropriately and completely partitioned among the new actors, the *IsolateActor (Operator)* and eventually *DeleteActor (Operator)* is applied to remove the actor *Operator* from the model.

All interactions of the use cases involved in the refactoring process are examined to identify if the lifeline for the actor *Operator* has an incoming call event in the interaction. Since the *BankServer* lifeline invokes the message *requestAnswer()* from the *Operator* lifeline in the *Handle Customer Queries* interaction, *CreateSubClass (Operator, Tech Support)* refactoring is performed to create a new class based on the new actor *Tech Support* to which the use case is assigned. The *PushDownOperation (Operator, requestAnswer)* is performed to move the operation to the *Tech Support* Class and finally *SubstituteLifeline (Operator, Tech Support)* is applied to redirect messages to the newly created sub class. The refactored model views are shown in Figure 65.

5.4.5 (g) Post Refactoring Model Improvement

System actors trigger use cases and an actor can start more than one use case within the system. This is depicted by an association relationship between the actor and the use case in the use case diagram. The more use cases associated with an actor, the more complex is the relationship between actors and the system.

Application of this refactoring reduces the number of use cases associated with an actor by splitting them among actors. This ensures that actors within the system are not user types but roles. From the viewpoint of an actor, the complexity of the system is reduced, as it has to deal with fewer use cases. Apart from improving the complexity of the actors and their interaction, behavior is properly distributed and associated to appropriate triggers. This restructuring also affects the structural view by introducing the concept of modularity through generalization and functionality distribution.

5.4.5 (h) Side Effects

The Spider's web model smell exists within a system due to improper actor identification and functionality association. Although reducing the number of use cases associated per actor comes at the cost of having more actors in the system. This increase in the number of actors affects the size of the system and hence increasing its overall use case point value used popularly for use case effort estimation.

5.4.6 Specters'

5.4.6 (a) Description

Specters' model smell occurs in cases where designers new to object-oriented design define system architectures. In this model smell, one or more ghostlike apparition classes exist in the system that appear only briefly to initiate some action in another more permanent class. We refer to these classes as Specter classes as they have a very brief lifecycle and are classes with limited responsibilities and roles to play in the system.

Although the name of this smell suggests a smell related to the class diagram, the existence of this smell requires information from all UML views for the following key reasons:

1. A specter class is a stateless class or in other terms, a class with no attributes. This can be identified from the system's structural view. This class is also referred to as an Irrelevant Class [431].
2. All associations of the specter class are transient. A temporary, short-duration class pops into existence only to invoke other classes through temporary associations. This can be confirmed by taking into consideration all the sequence diagrams (behavioral view) associated with the system. Specter classes within the sequence diagram usually send messages to other classes but never receive any messages back.
3. It is part of a single-operation use case that exist only to invoke other use cases through an *include* relationship. Single-operation use cases are usually in the center of a nested "include" path for delegating control to an essential use case.

5.4.6 (b) Rationale

The specters' model smell is a variation of a well-known anti-pattern known as Poltergeist [431]. The specters' model smell is usually intentional on the part of some architects who do not really understand the object-oriented concept. Availability of these classes results in a chaotic software designs, inclusion of unnecessary abstractions; and hence make the system design excessively complex, hard to understand, and hard to maintain.

5.4.6 (c) Target Quality Improvements

- Use Case Comprehension and Maintainability
- Management of Behavior Complexity
- Modular Design/Cohesion

5.4.6 (d) Model Smell Detection Strategy

To ensure the applicability of this model smell in the integrated model, classes with no attributes and associated with a number of other classes are selected. The behavior of these classes within the sequence diagram is then studied. If these classes are invoked by other classes only to act as a delegate or simply invoke other classes without receiving any reply, the existence of the specters' smell is confirmed. In order to reduce the search space, information from the functional view plays a vital role. Specter' classes are usually part of inclusion use cases or highly complex use cases (such as the God Use Case). Since the Multiple Personality smell discussed in Section 5.4.2 handles existence of transient classes that act as agent classes or middle-men classes, the specter's smell identifies transient classes that simply invoke other classes.

The pseudo code given below describes the steps required for automated detection of the specters' model smell.

```
: ALGORITHM: SPECTERS'
: start
:   read Model
:   for (each class in the Model)
:     read C
:     if (# of attributes for C is = 0)
:         for (each inclusion use-case in the model)
:             read UC
:             for (each lifeline associated with UC)
:                 read Life
:                         if (Life = C) and (# of receive Events for Life = 0)
:                                         false = 0
:                           else
:                                         false = 1
:                           end if
:             end for
:         end for
:       end if
:       if (false = 0)
:             {specters} = {specters} U (C)
:       end if
:   end for
:   return specters
: stop
```

5.4.6 (e) Model Refactoring Mechanics

Name: Remove Specters Class

Parameters: List *classNames* where,

- *classNames* is a list of classes suspected of being specter's

Preconditions:

i. The list of classes in (*classNames*) does not have any attributes (objects excluded).

Mechanics:

1. Search all classes that invoke the specters class and use the *Move Operation* refactoring to move the method to the classes that use it.
2. Since the specter classes invoke other permanent classes based on its initial invocation of the start method, these corresponding invocations are required to be moved to the invoking lifeline in all interactions that include the specters class. This is simply done by applying the *Remove Middle Man* refactoring.
3. Since all operations are moved to the classes that invoke the specters class, the *Remove Empty Class* refactoring is applied to remove the class from the structural view of the system.
5. If interaction belongs to an inclusion use case and removal of the specters class result in a no message occurrences except for other inclusions and extensions through the "ref" fragment, the *Collapse Fragment* refactoring is then used. This refactoring inserts the interaction fragment of the inclusion use case into the interaction diagram of the including/base use case at the point of inclusion (ref fragment).
6. Finally, *Merge UC Inclusion* refactoring is used to merge the inclusion use case into the including use case.

Figure 66 shows the ordering of the composite refactoring Remove Specters'.

Post Conditions:

i. Classes with names in the class-list *classNames* do not exist in the system.

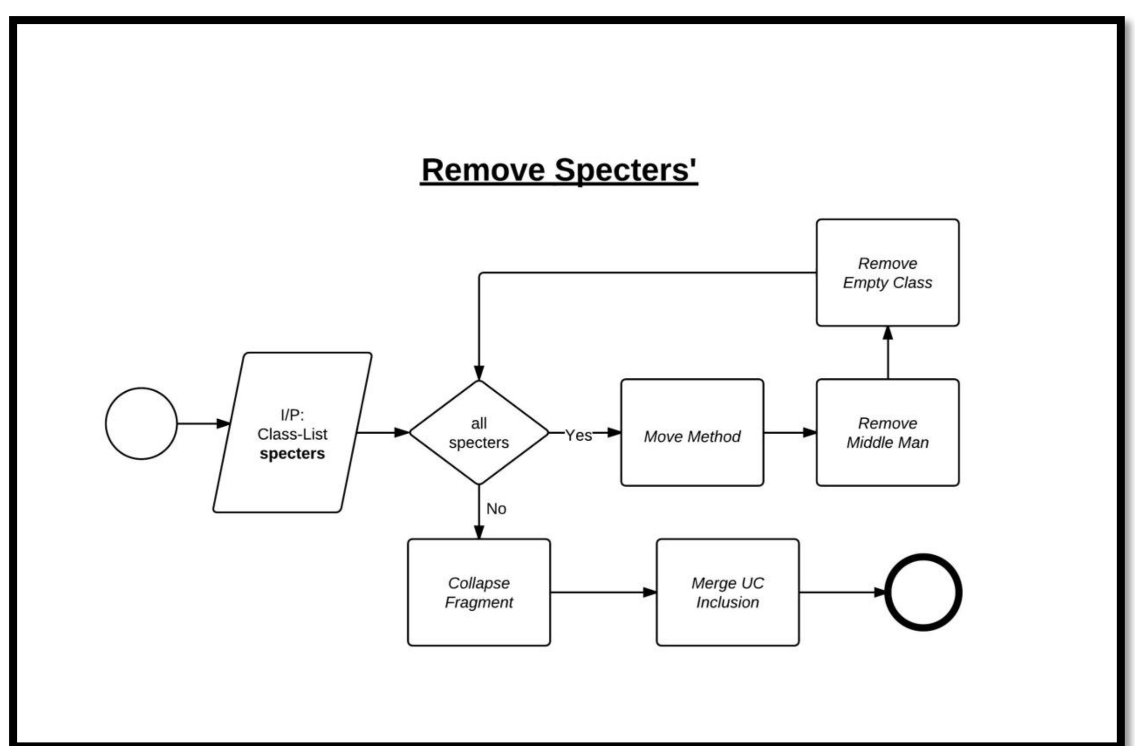


Figure 66 Remove Specters' Refactoring

5.4.6 (f) Example

Figure 67 shows a subset of the model views from the NBS system that depicts the specters' model smell.

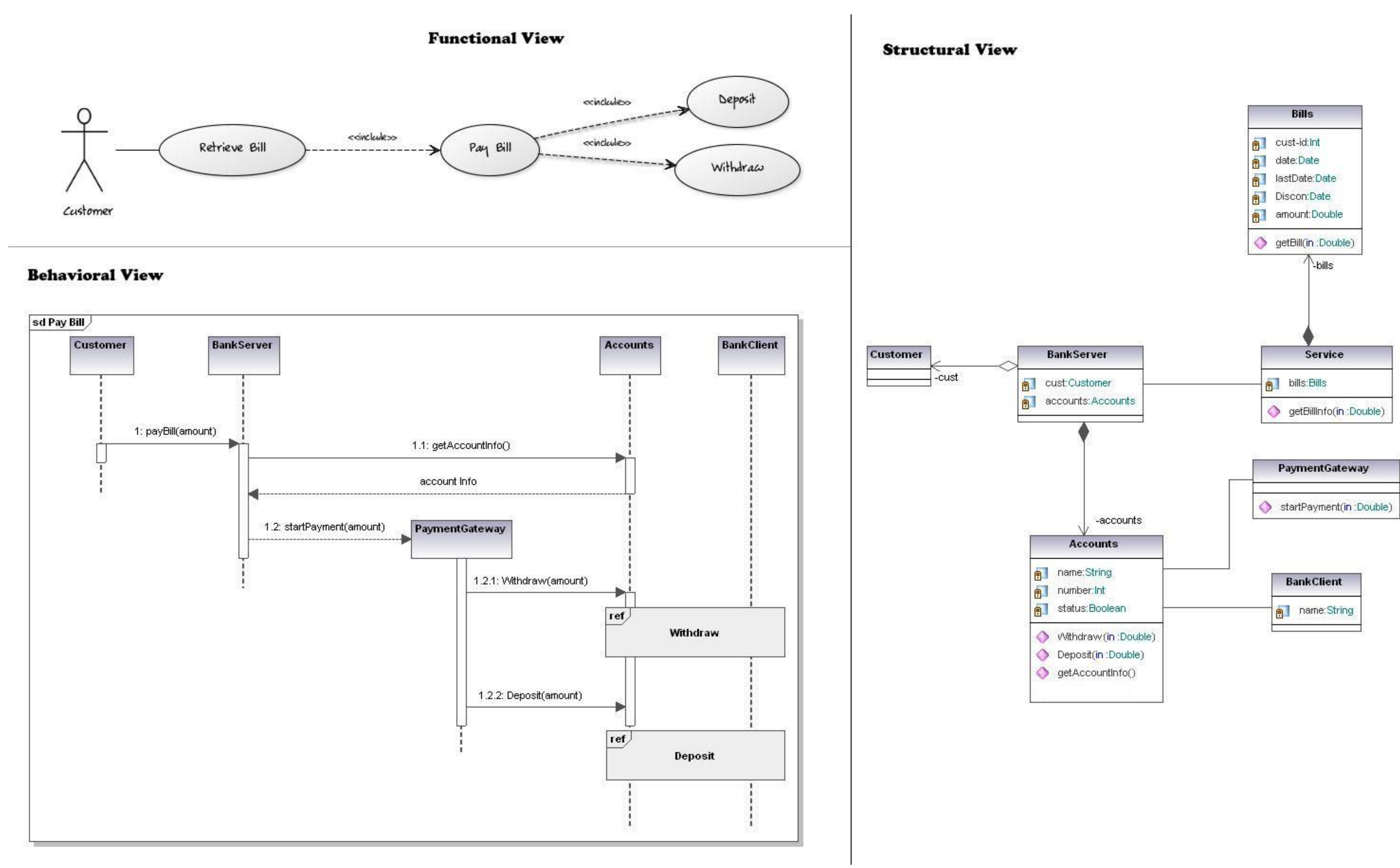


Figure 67 Excerpt of the NBS model views depicting Specters' Smell

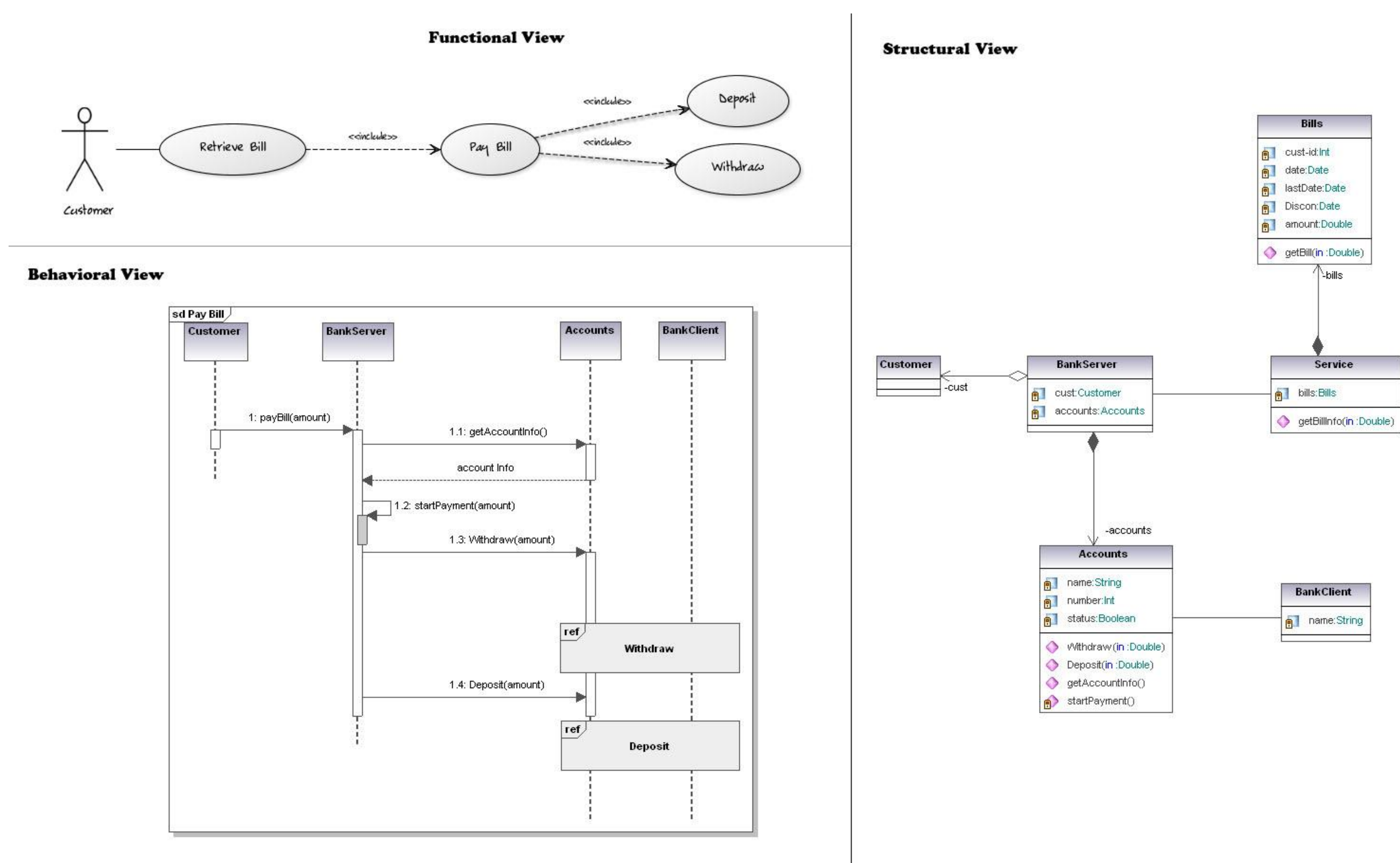


Figure 68 Excerpt of the NBS model views after refactoring

On examination of the class diagram, the existence of an *Irrelevant* class instance *PaymentGateway* was identified. Based on a list of all inclusion use cases (possibly those in the middle of a *Include* or *Extend* chain) obtained from the functional view, the interactions of all these were examined. The *Pay Bill* interaction made use of the *PaymentGateway* and the interaction had no receive events (except the invoking operation, which is ignored). Based on this information, the existence of specters' model smell is confirmed in the model.

Initially, the invoking operation is moved into all the associated classes. Hence, the *MoveOperation (PaymentGateway, Accounts, startPayment)* refactoring is performed for the given example. The *RemoveMiddleMan (BankServer, PaymentGateway)* refactoring is applied to remove the middle man lifeline and initiate direct communication. Since the invoking operation *startPayment* is moved to the invoking class, the empty class *PaymentGateway* is removed by applying the *RemoveEmptyClass (PaymentGateway)* refactoring. Since the *PayBill* interaction had other message occurances even after the removal of the specter class, the *CollapseFragment* refactoring and *MergeUCInclusion* refactoring are not invoked resulting in no change made to the functional view of the system. The refactored model views are shown in Figure 68.

5.4.6 (g) Post Refactoring Model Improvement

Specter classes have limited responsibility in the system. They are stateless classes with a short lifecycle. Removal of these classes from the system reduces behavioral complexity by removing unnecessary interactions and lifelines from the interactions and as a result improves modularity between classes in the structural view by reducing coupling and

increasing cohesion. As a result, an overall improvement is seen in the functional view wherein the seeding use-case behavior realized by includes and extends is reduced to reusability rather than adding to use case sequencing and scheduling. The depth of includes and extends relationship in the functional view is also reduced to enhance maintainability.

5.4.6 (h) Side Effects

When correcting anti-patterns such as specters' (or poltergeists), the local and structural refactorings applied to the design can produce side effects that may introduce other anti-patterns. The most common side-effect anti-pattern that may result because of removing specters' from the model is the God Class. This is because the removal of an irrelevant class merges its functionality into the associated class that earlier held methods whose data may have been located in a rich God class.

This side effect can be easily circumvented by allowing the application of refactorings that handle God class before this refactoring such as Multiple Personality, Creeping Featurism and Undue Familiarity. Hence, this could move attributes from the invoking God class and the specters' class would no longer be considered as an irrelevant class.

5.4.7 Model Duplication

5.4.7 (a) Description

Duplication is one of the most common bad smells when it comes to code based refactoring. Although usually not defined over models, the use of an integrated model allows for the identification of common model fragments throughout the system description. Therefore, Model duplication considering multiple views can be defined as

information objects described separately within the system specification even when processed in the same manner. Duplicated model fragments are more difficult to identify than duplicate code fragments mainly because they are not exact replicas of each other.

In order to detect duplication, an initial point has to be established from one of the views. In this smell description, the Actor-Use Case relationship is selected as the point of origin for duplication detection and analysis. This selection is based on use case duplication observed by Ciemniewska et al. [435]. The detection strategy described in this work starts from this point; that is identifying near similar patterns and confirming them through information from the behavioral and the structural view as it traverses the functional view.

5.4.7 (b) Rationale

Duplication, be it code or model, is considered one of the most abhorrent smell evident from the literature. Not only does it reduce reusability, changes made to one portion of the duplicated fragment will remain unchanged in other similar fragments. Detection of duplication was not handled in previous studies on model refactoring mainly due to the lack of complete information in one single view of the system specification. The integration of model views allows exploitation of inter-view relationships and aids in the detection of duplication across models view.

5.4.7 (c) Target Quality Improvements

- Reduction of Redundant Use Case Associations
- Reduction of Behavioral Redundancy
- Design Cohesiveness and Modularity

5.4.7 (d) Model Smell Detection Strategy

To ensure the applicability of this model smell in the integrated model, all actor-use case relationships are considered. To demonstrate this, we use the concept of trees. For each actor in the system, a tree is constructed (hypothetically) with the actor as the root node. Each of these trees is composed of multiple paths from the root node to the leaf node. An illustration of this concept is shown in Figure 69.

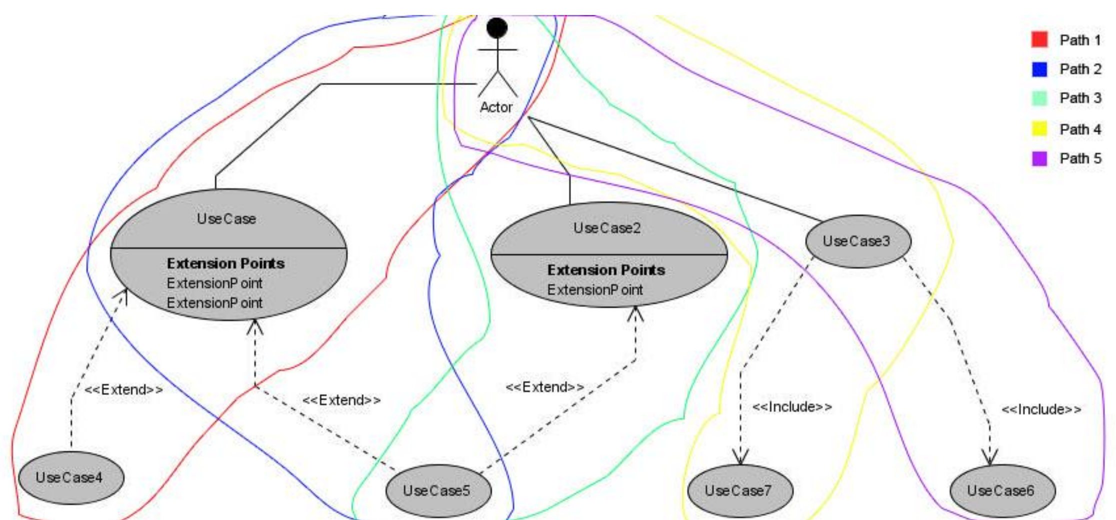


Figure 69 Concepts of Paths in the detection strategy for Duplication Model Smell

The maximum depth of paths traversed is equivalent the maximum value among the maximum Depth of Inclusion Relationship (DOIR) and the maximum Depth of Extension Relationship (DOER). Two paths are investigated for similarity if the root node (i.e. the Actor) and the leaf node (an extension or inclusion use case) are same. For instance, Paths 2 and 3 in Figure are similar and are investigated to identify the availability of Model Duplication Smell. For the sake of simplicity, the use cases between the root and the leaf node are referred to as Middle Use Cases. Behavior of all middle use cases are examined and compared to establish similarity. Two behaviors are structurally similar if:

1. The lifelines involved in both the interactions are same. If not, at least the different ones are sub classes of the same super class.
2. The sequence of message interactions among lifelines is the same. Each message interaction is represented by a tuple {source lifeline, message-type, destination lifeline}.
3. Message names may or may not be similar but the size of the arguments are same for messages between the same sequences.
4. Extension and inclusion use cases (through "ref" fragments in the behavior) are invoked at the same sequence.

If structural similarity between two similar use cases is established, the existence of Model Duplication is confirmed. The pseudo code given below describes the steps required for automated detection of the model duplication smell.

```
: ALGORITHM: MODEL DUPLICATION
: start
:   read Model
:   for (each actor in the Model)
:     read A
:     for (each use-case associated with A)
:           read UC
:           sig = A + UC
:           if (# of extends for UC > 0) or (# of includes for UC >0)
:               for (each extension or inclusion of UC)
:                       sig = sig + CLOSURE (UC)
:               end for
:           end if
:           {sig-set} = {sig-set} U (sig)
:      end for
:      for (each pair from {sig-set})
:           read sig1, sig2
:           if (size of sig1 = size of sig2) and (last two elements of sig1 and sig 2 are same)
:                       status = SIMILARITY (sig1, sig2)
:           end if
:           if (status = 1)
:                       dup = dup U {sig1, sig2}
:           end if
:      end for
:   end for
:   return dup
: stop
```

The pseudo-code for model duplication uses two sub-functions: CLOSURE and SIMILARITY. Since the functionality of CLOSURE is trivial, we do not provide the algorithm for it here. The pseudo code for SIMILARITY that checks for structural similarity of two interactions is given below.

```
: ALGORITHM: SIMILARITY
: start
:   read sig1 and sig2
:   for (each i from 2 to size-2 of sig1)
:     read UC1 = sig1(i) and UC2 = sig2(i)
:     diff = (lifelines in UC1) ∩ (lifelines in UC2)
:     if (diff is a super-sub relation)
:           for (each message occurance in UC1 and UC2)
:                   read msg1 in UC1 and msg2 in UC2
:                   msg1-set = (source, type, destination of msg1)
:                   msg2-set = (source, type, destination of msg2)
:                   if (msg1-set = msg2-set)
:                           similar = 1
:                   else
:                           similar = 0
:                           break
:                   end if
:           end for
:       end if
:     end for
:     return similar
: stop
```

5.4.7 (e) Model Refactoring Mechanics

Name: Remove Duplication

Parameters: Actor *a,* Use Case *uc1*, Use Case *uc2* String *newName*
where,

- *a* is the Actor
- *uc1* is one of the duplicate use cases
- *uc2* is the other duplicate use case
- *newName* is the name of a new use case that results from merging the two duplicate use cases.

Preconditions:

i. The name of the new use case (*newName*) does not conflict with the name of an existing use case within the model.

ii. The use cases *uc1* and *uc2* are assigned to Actor *a*.

Mechanics:

1. *Create UseCase* refactoring is used to create a new use case.
2. *Extract Fragment* refactoring is then used on either use case sequence diagram (*uc1* or *uc2*) to extract the complete interaction into the newly created use case.
3. Since the structurally similar behavior of the two use cases may have different messages, the *Replace Message* refactoring is used to rename the message. An argument "*type"* is also added to the message that determines the type of action performed by the structurally similar use cases. *Merge Operation* refactoring is also applied to merge the lexically different operations in the class and renamed it to the new message name used in the interaction.
4. If different lifelines exists in the two interactions (they are sub-classes based on the constraint included in the smell description), *Substitute Lifeline* refactoring is performed to add the super class to the interaction.
5. *Add Actor Reference* refactoring is performed to add an association between the actor triggering the use cases *uc1* and *uc2* and the new use case.

6. *Isolate UseCase* refactoring is applied to the use cases *uc1* and *uc2* to isolate them from the use case model.
7. *Delete UseCase* refactoring is used to remove the use cases *uc1* and *uc2* from the system.

Figure 70 shows the ordering of the composite refactoring *Remove Duplication*.

Post Conditions:

i. Use cases with names *uc1* and *uc2* does not exist in the model.

ii. Use case with name *newCase* is added to the model.

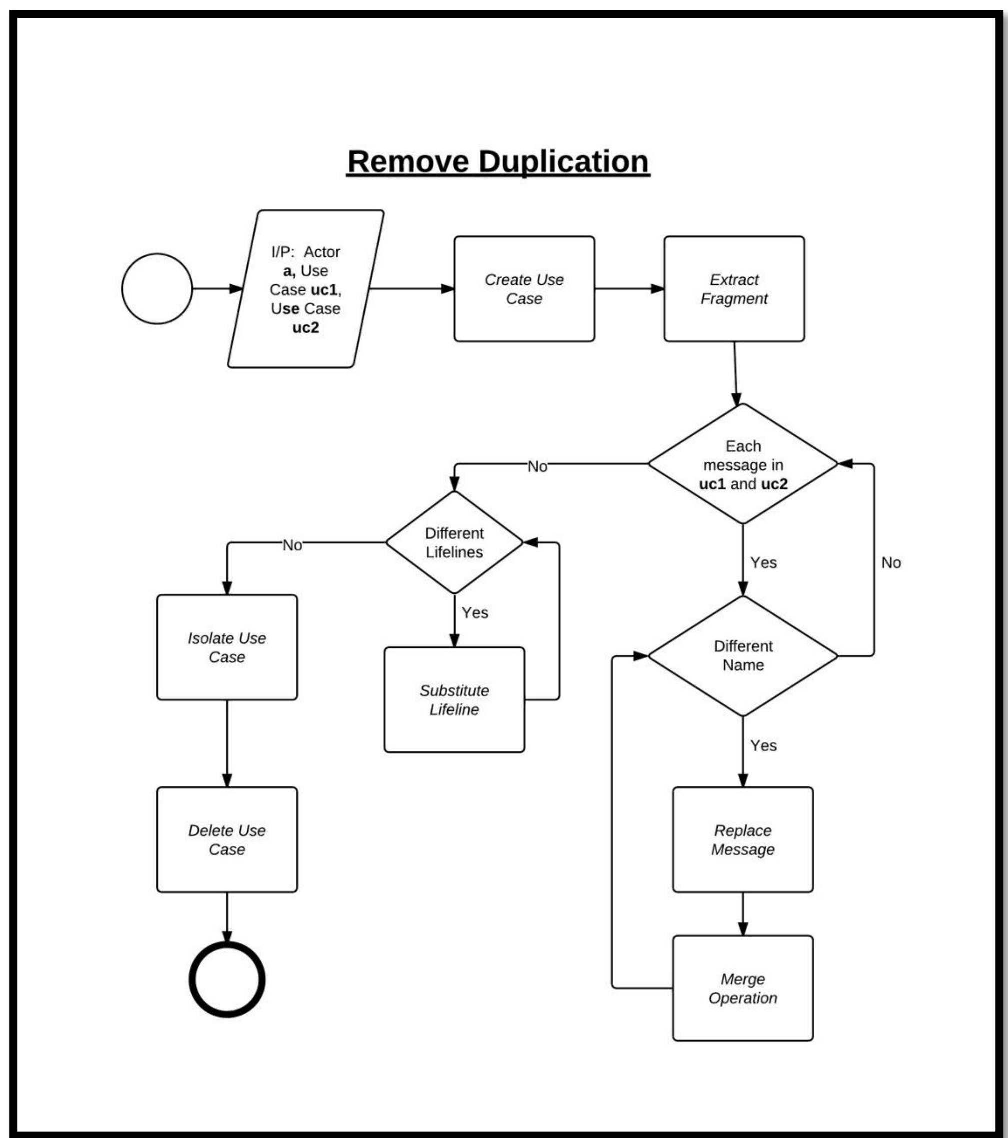


Figure 70 Remove Duplication Refactoring

5.4.7 (f) Example

Figure 71 shows a subset of the model views from the NBS system that depicts the duplication model smell. On examination of the use case diagram, two paths associated with the actor *Operator* were identified. In order to ensure the existence of the model smell, the behavior of the middle use cases involved *Add Application Form* and *Edit Application Form* were observed. The sequence of message occurrence between the two

interactions was found to be structurally similar. Hence, the existence of the duplication model smell was confirmed.

Initially, the *CreateUseCase (Manage Application Form)* refactoring is applied to create an empty isolated use case. Then the *ExtractFragment (Add Application Form, Manage Application Form)* refactoring is performed to copy the complete interaction fragment from one of the similar use cases (either can be used) into the new use case. In order to identify lexically different message interaction between the use cases, each message in the interaction of *Add Application Form* and *Edit Application Form* is compared. A message with a different name is replaced in the interaction of the new use case *Manage Application Form* with a new message. The following refactoring operation is hence applied *ReplaceMessage (createForm, manageForm(type))*. If both the messages are not used in any other interactions, they are replaced in the class diagram. The *MergeOperation (createForm, EditForm, manageForm)* refactoring is applied to the structural view to apply the change. Since the use of super-sub class relationship was not utilized (as lifelines in both the use cases were same), the *AddActorReference (Operator, Manage Application Form)* is applied. The duplicate use cases are initially isolated by applying the *IsolateUseCase (Add Application Form)* and *IsolateUseCase (Edit Application Form)* and finally deleted by applying the *DeleteUseCase (Add Application Form)* and *DeleteUseCase (Add Application Form).*

The refactored model views are shown in Figure 72.

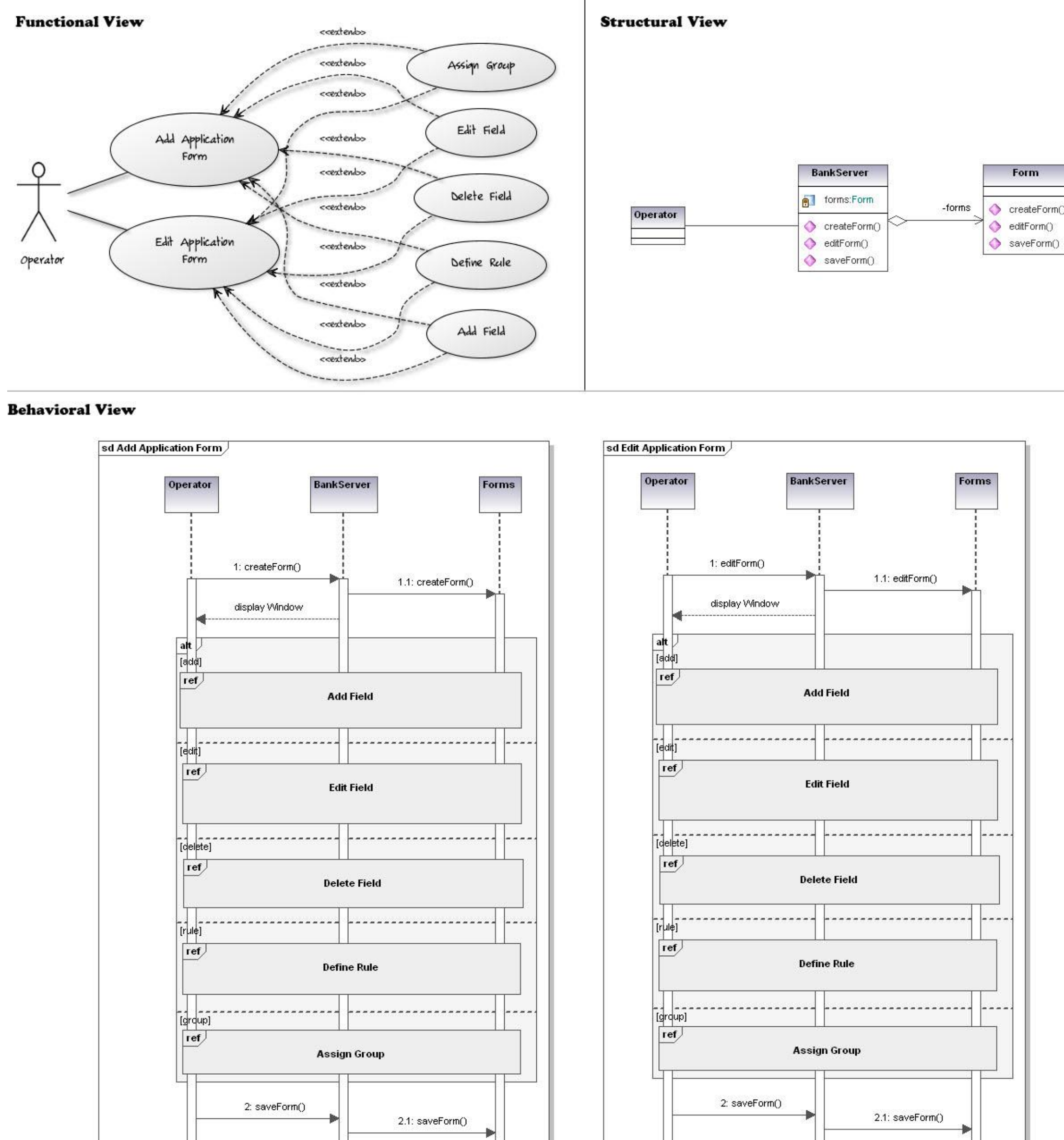


Figure 71 Excerpt of the NBS model views depicting Duplication Smell

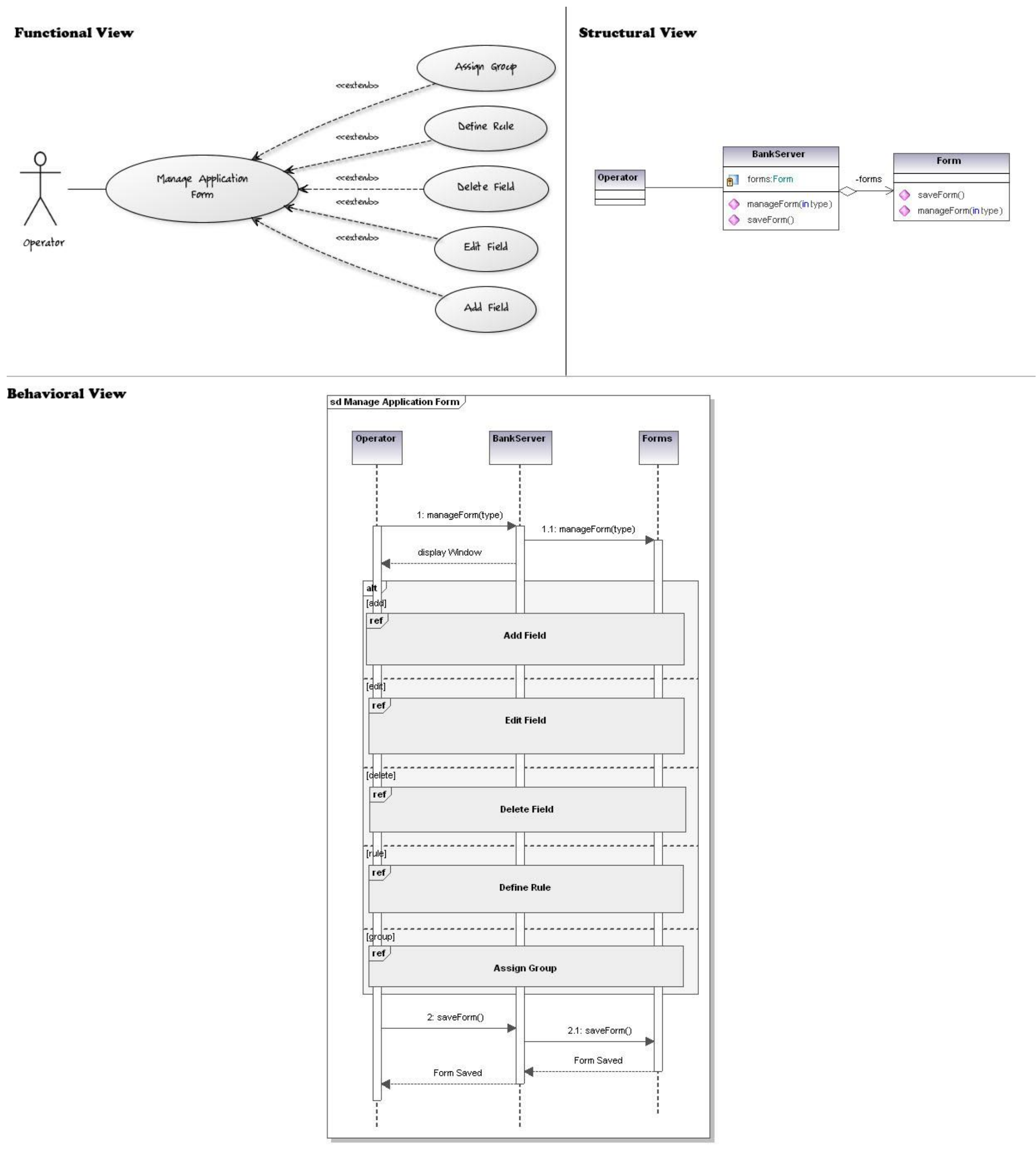


Figure 72 Excerpt of the NBS model views after refactoring

5.4.7 (g) Post Refactoring Model Improvement

Duplication is one of the most common defects that can be observed in models. The most common form of this duplication is through the use of similar or different information objects and describing the processes that manipulate them as separate use cases.

Merging use cases that handle similar information objects through a structurally similar sequence of message interactions reduces the redundancy in describing their behavior in the sequence diagram. It also reduces the number of use cases in the use case model and the number of use cases associated with an actor, which in turns reduces complexity of the use case model. Merging use cases that manipulate different information objects through a similar process helps in identifying and applying object-oriented principles such as reusability through inheritance and polymorphism to the structural view of the model.

5.4.7 (h) Side Effects

Although the removal of duplication from the integrated model does not introduce side effects into the model, it does require a change in the operation arguments in the class and sequence diagrams. Ensuring behavior preservation can get complicated with the model size and hence the complexity of this refactoring is directly proportional to the size of the integrated model considered for refactoring. A more stable algorithm for duplication resolution for large model system is hence sought as a future work of this model smell.

5.4.8 Ripple Effect

5.4.8 (a) Description

A change in one design artifact can cause cascading changes to all related artifacts. This propagation is based on the degree of dependency that exists between the related artifacts. In case of a multi-view modeling environment such as UML, artifacts usually belong to different views. Functional requirements specify the intended behavior of the system and use cases have become a widely accepted modeling notation for capturing them. Software requirements are volatile and their change can occur at multiple points during the development process and is inevitable [436]. The ripple effect model smell identifies the strength of dependency between use cases and classes which are connected through an intermediate artifact; the sequence diagram. The strength of dependency is an indicator that a change in the use case specification will eventually effect the structural organization of objects within the system. A high degree of change can therefore question the stability of the system and severely affects its efficiency and maintainability.

The ripple effect model smell is a variation of the shotgun surgery and divergent change bad smells proposed by Fowler et al. [15]. However, unlike them, the ripple effect makes use of the additional information from functional view and tries to identify the change impact caused to the structural and behavior view because of changes to the functional requirements of the system under design.

5.4.8 (b) Rationale

Dependency between different artifacts is mainly due to the use of multi-phase development by most of the software development paradigms. Use of information from

one artifact for the development of others ensures consistency. Although dependency is certain, the degree of dependency depends on the design of the system. If modeled incorrectly, severely affects the design maintainability and reusability. When the number of classes implementing a use case is high, this indicates that changes in a use case can have impact on a large number of classes. This change propagates to all other related classes and since classes are shared between use cases results in a cycle of change propagation. More specifically, an indicator that related functionality is spread over the system design. Hence, this adversely affects design stability and maintainability.

5.4.8 (c) Target Quality Improvements

- Use Case Maintainability and Reusability
- Behavioral Dependency
- Structural Stability

5.4.8 (d) Model Smell Detection Strategy

Based on the inter-view relationship, the number of classes per use case can be identified by information from use case diagrams, sequence diagrams and class diagram. Use cases describe the functional requirements of a system. Classes implement these requirements and their participation within use cases is depicted in the sequence diagrams. In order to detect the existence of the Ripple Effect smell, we developed a basic metric called *Impact Factor (IF),* which is calculated for each use case.

Each class in the integrated model is associated with a number of other classes through association, aggregation and composition relationship. The metric *Number of Associations Linked to a Class (NASC)* provides this value for each class (see Appendix

9). The behavior of each use case is represented through a sequence diagram, which is composed of a number of classes. For a given class (lifeline) in a particular use case, we calculate the number of classes it is interacting within the interaction of the use case. We refer to this as the *Number of Internal Connections (NOIC)*. Based on this information, we calculate the *Number of External Connections (NOEC)* for each class in a use case behavior as follows:

$$NOEC_{class} = NASC_{class} - NOIC_{class}$$

Hence, NOEC is the measure of the number of classes that might be affected because of any change occurring to the description of the class. Hence, the Impact Factor metric is a summation of all classes external to the use case that may be affected because of a change made to the requirement specification modeled by the respective use case. The *Impact Factor* is thus calculated as follows:

$$IF_{use\ case} = \sum_{class} NOEC_{class}$$

The Ripple Effect model smell identifies classes most affected by a change in the functional requirement of the system and tries to solve this by localizing changes through model refactoring operations over all participating views.

In order to quantify an acceptable *Impact Factor* metric for a use case we use a maximum threshold value UP_{IF}. Since this upper limit threshold value is not available in the literature, we consider the 70/30 principle. Hence, the upper limit is equal to

$$UP_{IF} = 0.3 * NCM$$

Where NCM is the number of classes in the system (see Appendix 9). This ensures that 30% of change is allowed (i.e. 70% should be stable and not affected). The pseudo code given below describes the steps required for automated detection of the ripple effect model smell.

```
: ALGORITHM: RIPPLE EFFECT
: start
:    read Model
:    for (each use-case in the Model)
:      read UC
:      if (IF(UC) >= UP_IF)
:          {uc-list} = {uc-list} U UC
:       end if
:    end for
:    return uc-list
: stop
```

5.4.8 (e) Model Refactoring Mechanics

Name: Class Responsibility Assignment

Parameters: List *ucNames* where,

- *ucNames* is the list of use cases that are not stable

Preconditions:

i. The name of the new class (*newClass*) does not conflict with the name of an existing class within the model.

Mechanics:

For each use case in the ucNames list, the class (lifeline) contributing most the value of *IF* is selected. The resolution of this smell requires identifying

applicability of Single Responsibility principle, part of the design principles proposed by Martin [437] better known by their mnemonic acronym S.O.L.I.D. The Single Responsibility principle targets cohesion. There should never be more than one reason for a class to change. If a class has more than one responsibility, then they become coupled. Changes to one responsibility may impair or inhibit the class' ability to meet the others. The following refactoring operations are applied to divide responsibility,

1. *Create Sub Class* refactoring is used to create two sub classes from the names provided in the *newClass* list.
2. *Push Down Method* refactoring is then used to push the related alternatives behavior to the sub classes. This assigns responsibility of the behavior (method) using polymorphic operations to the classes for which the behavior varies.
3. Finally, *Substitute Lifeline* refactoring is applied to replace the lifelines with their appropriate child classes from the structural view.

Figure 73 shows the ordering of the composite refactoring *Class Responsibility Assignment*.

Post Conditions:

None

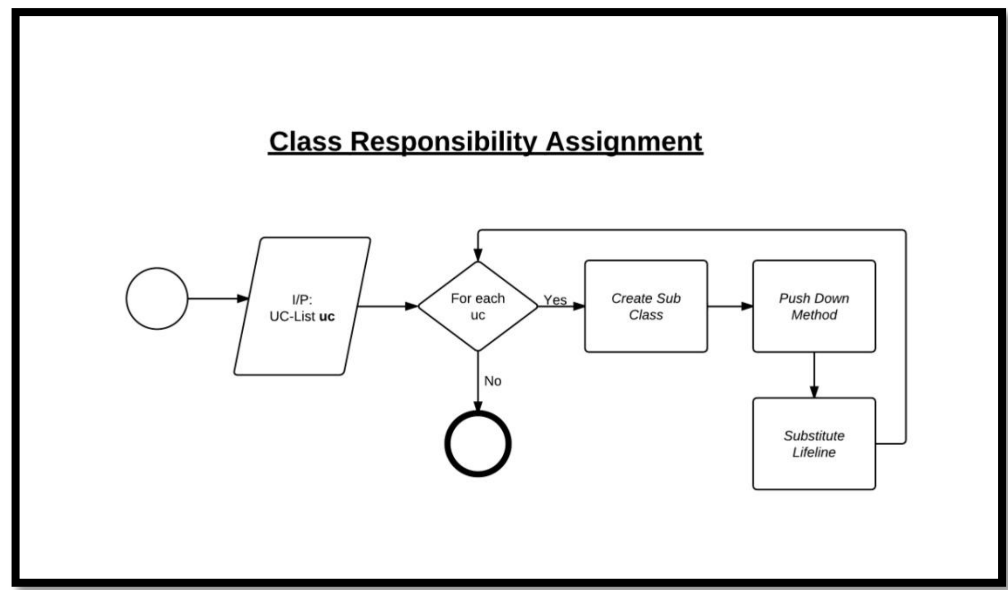
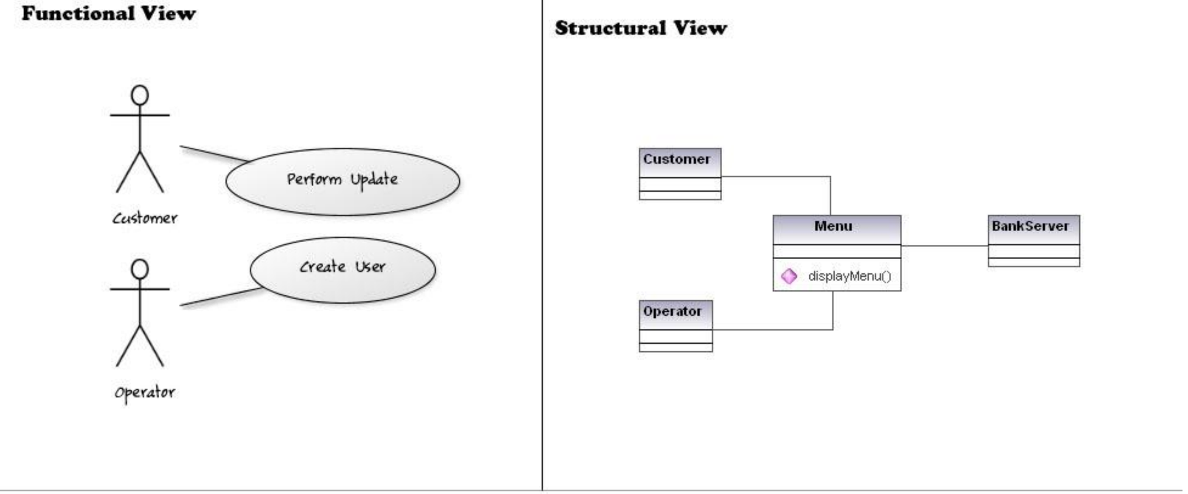


Figure 73 Class Responsibility Assignment Refactoring

5.4.8 (f) Example

Since a complete example of this model smell detection and resolution is difficult to portray, we illustrate an abstract example using the same NBS system. The two actors Customer and Operator of the NBS system access their functionality through a menu that is handled by the Menu Class. Two use cases considered for this illustration are Update Information and Create User. Figure 74 shows a subset of the model views from the NBS system that depicts the ripple effect model smell.



Behavioral View

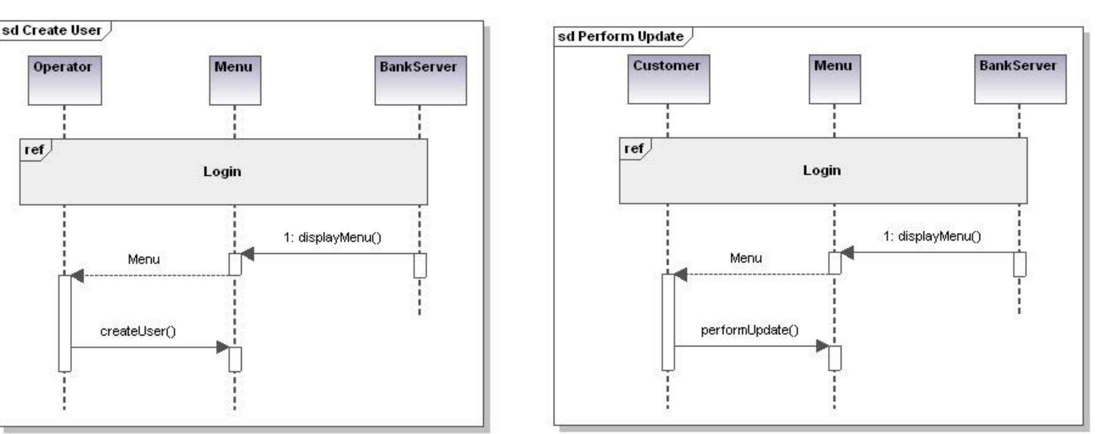


Figure 74 Excerpt of the NBS model views depicting Ripple Effect Smell

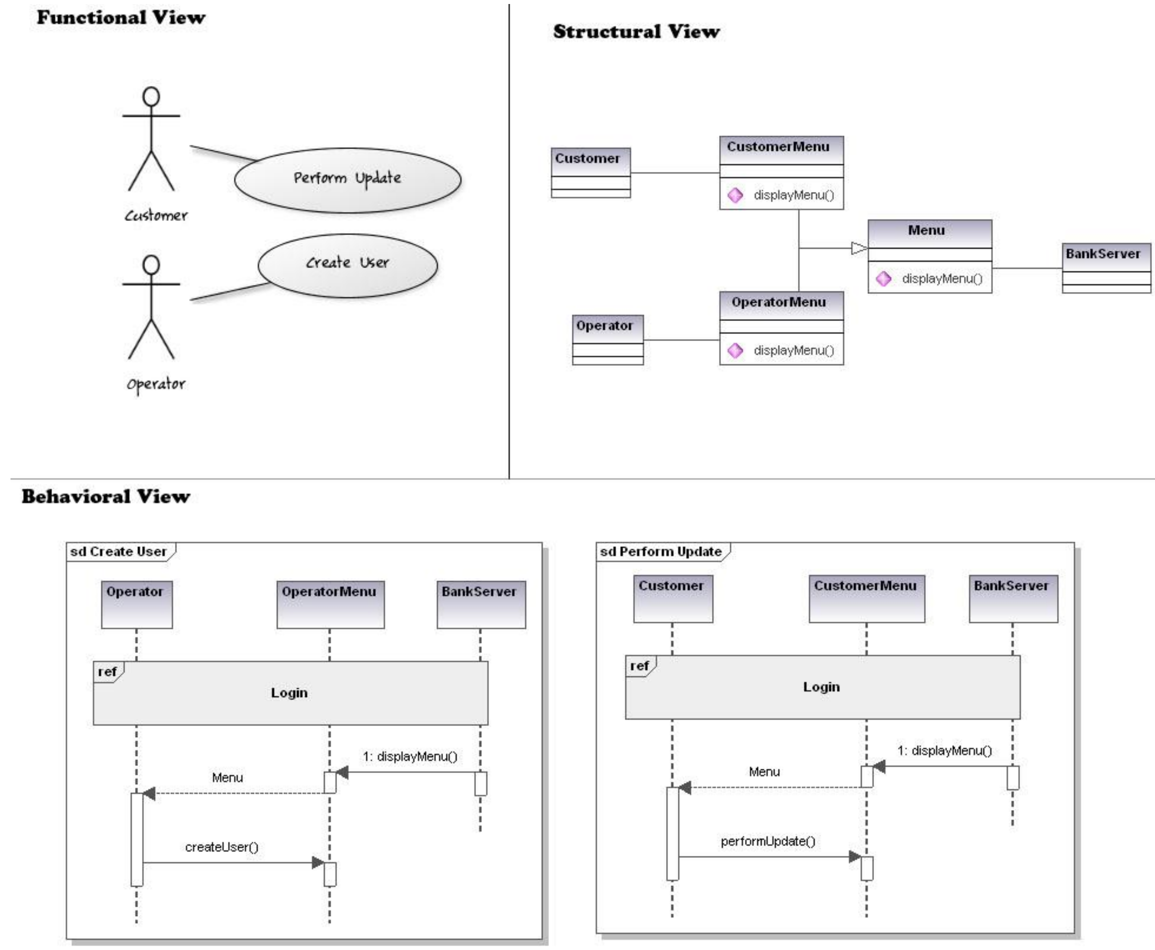


Figure 75 Excerpt of the NBS model views after refactoring

Figure 75 depicts a refactored version to solve this problem. In the refactored version, two subclasses are employed *OperatorMenu* and *CustomerMenu* of the class *Menu*. Each of the responsibilities is assigned to the subclass individually. As a result, *displayMenu* is implemented with these subclasses separately. Thus, the function of displaying operator menu is implemented in the class *OperatorMenu* without any alternatives of customers and the operator; these conditional branches are realized by means of polymorphism of *displayMenu* in the subclasses. The following refactoring operations are used to obtain the refactored version

1. *CreateSubClass (Menu, OperatorMenu)*
2. *CreateSubClass (Menu, CustomerMenu)*
3. *PushDownOperation (Menu, displayMenu)*
4. *SubstituteLifeline (Menu, OperatorMenu)*
5. *SubstituteLifeline (Menu, CustomerMenu)*

5.4.8 (g) Post Refactoring Model Improvement

A class that is coupled to a large number of other classes, and would produce a large number of changes throughout the system in the event of an internal change (due to a change in the use case specification the class is part of), contributes to the Ripple Effect smell. By the definition, a class that presents this smell tends to be coupled to a large number of other classes. Hence, removing this smell reduces the coupling between the classes in its structural view. This in turn localizes the effect on any change made to the behavior of the use case to classes included within the use case only and reduces their impact on other classes.

5.4.8 (h) Side Effects

Ripple Factor is a result of improper responsibility distribution within the software model beginning from its functional view in high-level design phase and propagating to its structural view in low-level design phase. Proper assignment of responsibility of classes based on the information from the functional view will not cause any side effects within the design model. It will in turn make the design more resilient to change by localizing changes and demonstrate effective use of object-oriented design principles.

CHAPTER 6

TOOL SUPPORT

This chapter offers discussion about the two tools developed as part of this work: *UCDesc* and *IntegraUML*. UCDesc is a complementary tool to provide use case modeling support for IntegraUML. IntegraUML is the main tool that provides the capability of integrating UML models; specifically class diagrams, sequence diagrams and use case diagrams, and allowing designers to refactor this integrated model. The schema for the Integrated Model is created using Altova XMLSpy 2010 [438] and presented in Appendix 7. In this chapter, we describe in detail the motivation, architecture and implementation of both the UCDesc tool and IntegraUML tool.

6.1 UCDesc: A Use Case Description Tool

UCDesc is designed for documenting and analyzing the behavior of use cases in the system. The most important element of Use Case analysis is the authoring of Use Case "flow" or "narrative". Traditional UML tools provide limited support for this vital artifact. As a result, designers end up using word processors and a myriad of informal templates to document use cases. The main motivation for designing the UCDesc tool is to allow designers to properly design and analyze use cases and to provide capability of exporting them as a means of model interchange. In this section, we provide details of the implementation and usage of this use case description tool.

6.1.1 Analysis of Existing Use Case Modeling Tools

Use case modeling tools can be classified into two categories: tools that provide support in the construction of the structural view of use cases and tools that provide support in documentation of the behavioral aspect of use cases along with the structural view. Since the structural view is the one that is part of the UML standard, numerous commercial software tools are available [439-443]. Experts agree that the most important aspect of use case analysis is the authoring of use case descriptions. However, traditional UML tools provide limited support for this important activity. Some provide basic description features such as composing use case behavior as prose text or documentation [444, 445].

Three noteworthy tools that are available for modeling the behavior aspect of use cases are CaseComplete [446], Visual Use Case [447] and Visual Paradigm for UML [448]. These tools provide powerful features when it comes to composing use case flow of events. They all provide a glossary feature that allows the reuse of similar terms in the flow of events. The flow of events in all tools has two representations, the traditional flow of events representation with a sequence of numbered steps and an activity diagram representation. Hyperlinks are provided in steps to allow access to use cases referenced through the include and extend relationships in the main flow. These tools also provide additional functionality like requirements tracing, collaboration and versioning.

Although powerful use case editors, the above-mentioned tools have some disadvantages. Visual Use Case lacks the functionality to export its diagrams to XMI. Due to this limitation, output models cannot be reused for analysis in other tools. One of the main advantages of UCDesc is that it provides the ability to export to XMI format so that it can be reused by other UML modeling tools for analysis and integration. Both CaseComplete

and Visual Paradigm for UML do provide an XMI export capability to the user but lacks a fine-grained description of the flow of events in the resultant XMI file. An excerpt taken from the XMI generated by CaseComplete is shown in Figure 76. Some tags are modified for clarity of representation.

As shown in Figure 76, all the steps in the flow of events of a use case are available as an attribute value. UCDesc provides syntactic processing of steps in order to allow a fine-grained representation of steps within flow of events. UCDesc relies on a carefully developed use case narrative metamodel. This metamodel provides sentence level analysis of use cases steps making them more readable and understandable.

```
<?xml version="1.0" encoding="utf-8" ?>
<XMI xmi.version="1.1" xmlns:UML="http://www.omg.org/UML">
 <XMI.header>
  <XMI.documentation>
        <XMI.exporter>Serlio Case Complete</XMI.exporter>
        <XMI.exporterVersion>5.2.3972</XMI.exporterVersion>
  </XMI.documentation>
  ……………..
  <UML:TaggedValuexmi.id="TV6" tag="Steps" value="1. Admin Logs into
     the System 2. System displays username and password to the Admin
     3. Admin enters the username and password into the System 4.
     Incorrect Username" modelElement="UC-34fb0ce5-fc39-46f4-b0f6-
     84eca0786a50" />
</XMI>
```

Figure 76 Sample XMI excerpt exported by CaseComplete UML Tool

Based on recent works done on extending the UML metamodel to supplement behavioral information, a number of prototype tools have been proposed [361, 363]. Since these tools depend heavily on the metamodel proposed in their respective works, it could not be used in our work.

6.1.2 UCDesc Architecture

UCDesc consists of sub-systems that provide different end user functionality. Figure 77 shows the architecture of UCDesc tool. The Use Case Editor documents the interactions between actors and use cases. Users can use the editor to write the narratives for the use cases, from the invocation of use case until the user accomplish the use case. Users can also document sub flows and alternative flows that extend from basic flow by defining sub and alternative scenarios respectively.

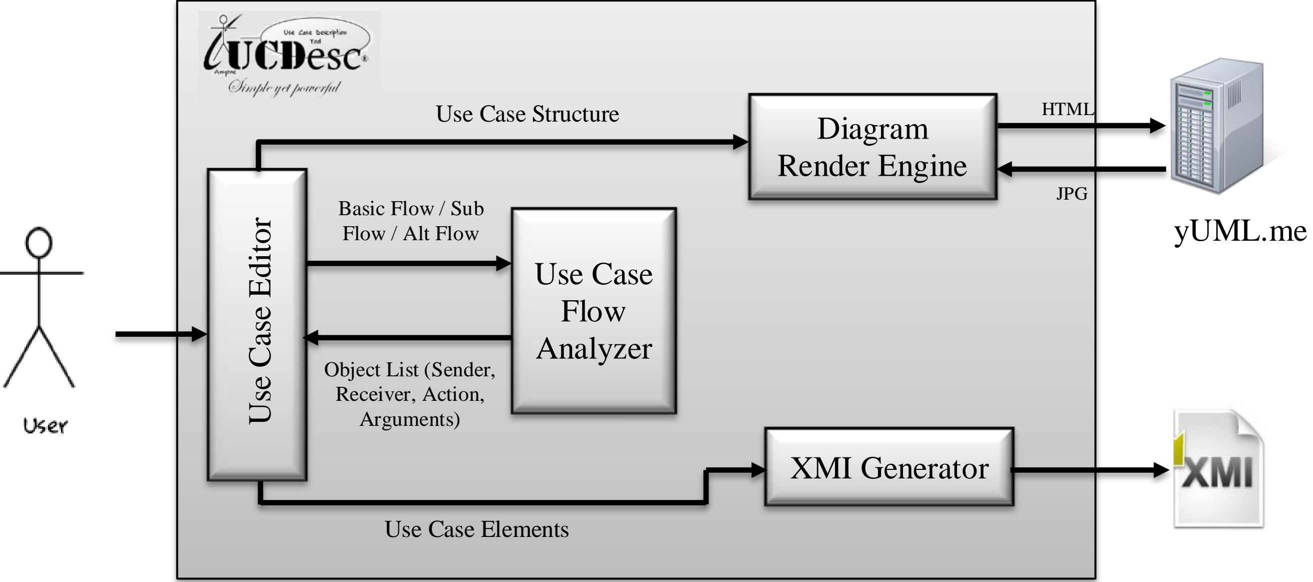


Figure 77 High-Level Architecture of UCDesc Tool

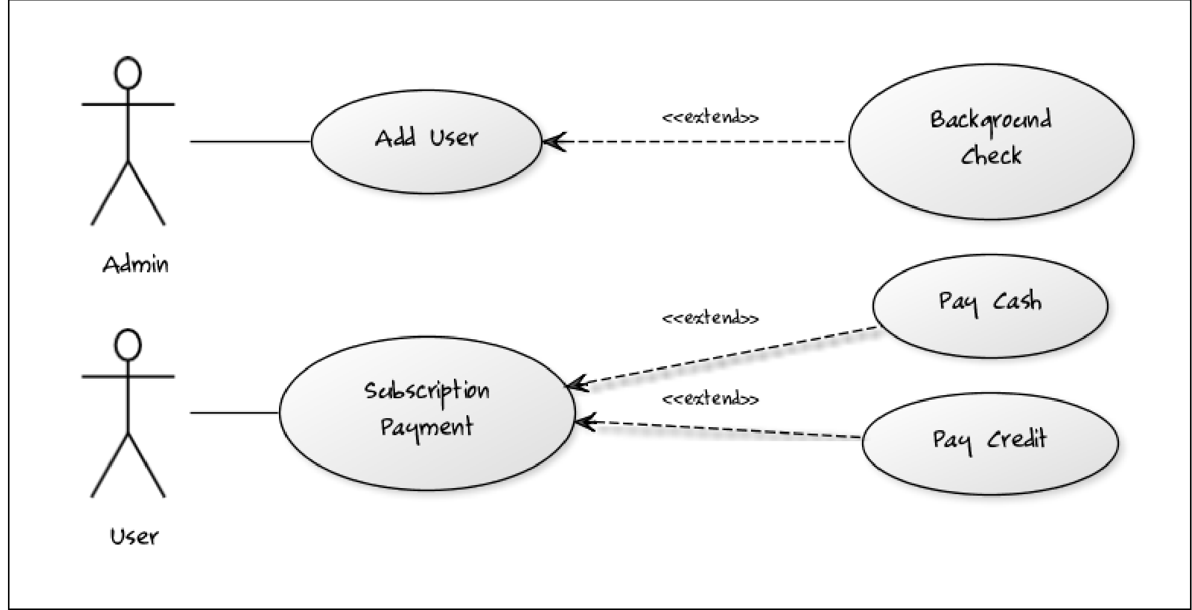
One of the most important features offered by UCDesc is the inclusion of use case flow analyzer. This module accepts the steps from the flow of events and identifies for each step, the sender, receiver, action and arguments. The analyzer works in two steps as follows:

1. **Tagging:** Each Step from the flow of events is tagged using part-of-speech (POS) tagging to distinguish nouns, verbs and adjectives in the sentences as candidate features that indicate syntactic structure. The Stanford POS tagger [449] is used in by UCDesc to accomplish this task.
2. **Mapping:** Based on the syntactic structure derived from the POS tagger, the mapping table shown in Table 7 is used to identify the objects of the flow step. Part of this mapping is based on the works of Li [370].

Table 7 Mapping of Syntactic Structure of Sentences into Use Case Objects

No.	Syntactic Structure	Sender	Receiver	Action	Arguments
1	subject verb object	subject	object	verb	-
2	subject verb1 object1 verb2(object2)	subject	object1	verb2	object2
3	subject verb object adjective	subject	object	be+ adjective	adjective
4	subject verb object1 participle (object2)	subject	object1	verb participle	(object2)
5	subject verb object1 object2	subject	object1	set+ object2	-
6	subject verb1 object conjunctive to verb1 (object1)		subject	verb1	object, verb1 (+object1)
7	subject verb gerund object		subject	verb	gerund verb (+object)
8	subject verb object1 preposition object2	subject	object2	verb	object1
9	subject verb (for) complement		subject	verb	complement
10	subject verb		subject	verb	
11	subject be predicative		subject	be + predicative	
12	subject verb preposition object		subject	verb + preposition	object

The diagram render engine renders the use case diagrams. UCDesc does not provide a built-in diagramming utility and hence uses a web-based use case diagramming tool known as yUML [450]. An appropriately constructed link (or URL) is accepted by the yUML server which then produces an image file with the use case diagram. This diagram is displayed in the systems web browser. It is the responsibility of the rendering engine to accept use case structural information from the editor and generate an HTML file to be passed on to the yUML server. The structural information includes actor-use case and use case-use case relationships. An example of the hyperlink generated and a sample rendered diagram is shown in Figure 78.



http://www.yuml.me//diagram/scruffy;/usecase/// Bill Payment Service, [Admin]-(Add User), [User]-(Subscription Payment), (Subscription Payment)<(Pay Credit), (Subscription Payment)<(Pay Cash), (Add User)<(Background Check)

Figure 78 Example yUML Link and corresponding Use Case Diagram

The XMI generator module generates XMI output of the use case diagram. In order to specify the structure of the flow of events in use cases, an extended version of the use

case metamodel was proposed in this work. This extended use case metamodel is shown in Figure 33. The XMI is based on an XML Schema presented in Appendix 5.

6.1.3 Features of UCDesc Tool

UCDesc is a simple use case description tool built on Java programming platform. The primary objective of UCDesc is to allow users to compose use case descriptions and provide the capability of exporting it to XMI. The main layout of UCDesc is shown in Figure 79.

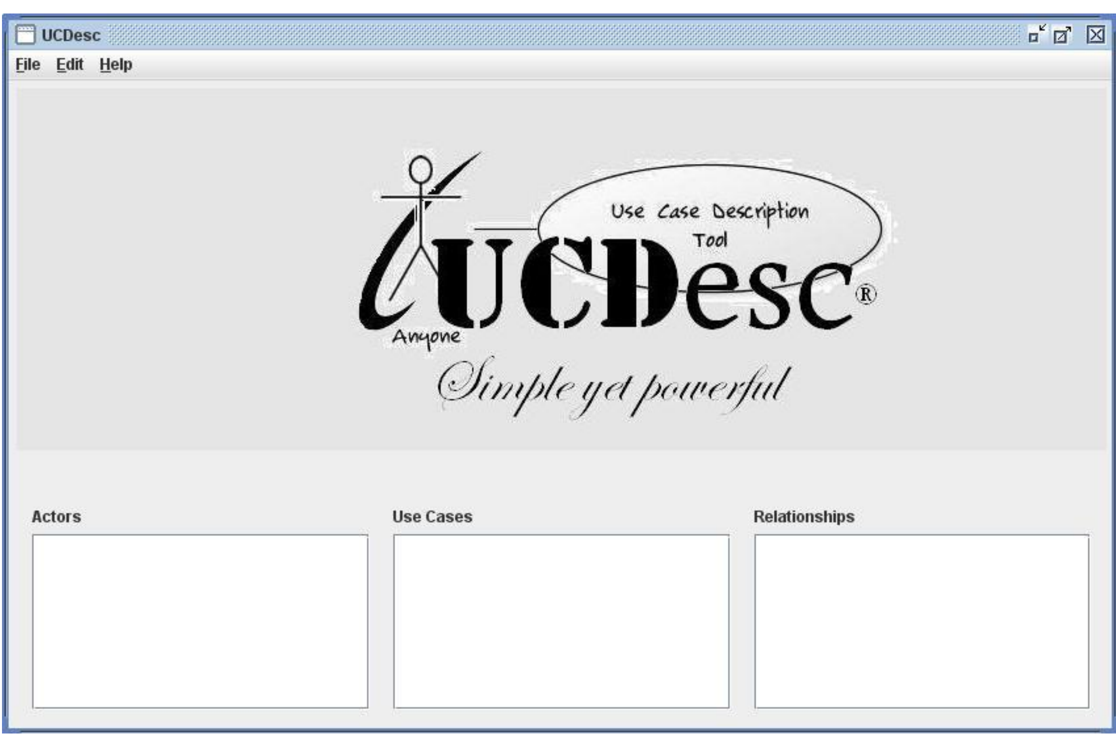


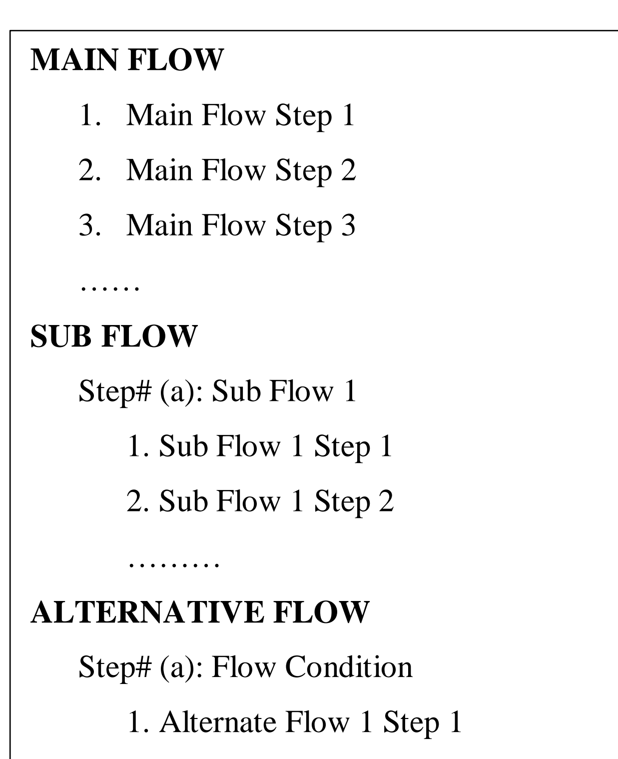
Figure 79 UCDesc Main Layout

The main layout consists of a top menu bar and three panels at the bottom: Actor, Use Cases and Relationships. The Actor panels lists all the actors available in a project, the use cases panel lists all the use cases available in the project and the relationships panel lists all the use case relationships (include and extends) available in the project. Users can

add actors and use cases making use of the options available in the Edit Menu. Relationships are added automatically when the user adds them to a particular use case description.

The format of use case description template followed by UCDesc needs to be defined here for comprehension. Composing use case flows requires the understanding of the following important guidelines:

1. **Use case step numbering:** In order to follow the different paths through a use case, the use case numbering scheme for the flow of events plays an important factor. Since there is no specific UML specification regarding the numbering scheme, the format adopted by UCDesc is shown below in Figure 80. The steps numbered 1, 2, 3 and so on makes up the main flow. The sub flows and alternative flows are specified after the main flow. The numbering of the sub flows and the alterative flows includes the step # where they can be invoked followed by a character (a-z) in case if more than one sub or alternative flow can be invoked at the same step of the main flow. Sub flows and alternative flows can themselves have sub and alternatives flows. Where to continue the execution after the end of a sub or an alternative flow is specified by a "Return" statement which indicates the return step. If there is no "Return" statement, the use case ends.



MAIN FLOW

1. Main Flow Step 1
2. Main Flow Step 2
3. Main Flow Step 3

……

SUB FLOW

Step# (a): Sub Flow 1

1. Sub Flow 1 Step 1
2. Sub Flow 1 Step 2

………

ALTERNATIVE FLOW

Step# (a): Flow Condition

1. Alternate Flow 1 Step 1
2. Alternate Flow 1 Step 2

………

Figure 80 UCDesc Use Case Description Format

2. **Including a Use case:** A use case can be included into another use case by using the anchor *Include* in the flow step followed by a use case name. For instance

   1. Include Login

   At step 1 in the main flow, the use case *Login* is invoked.

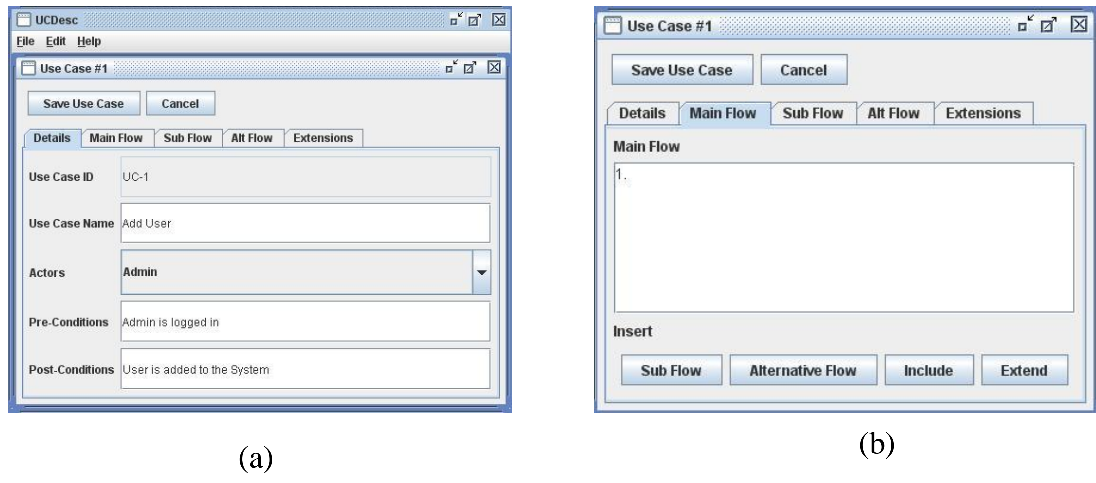
3. **Extending a Use case:** Extending a use case is a more complicated than the inclusion case. Before extending a use case, extension points must be defined in the base use case. A use case can be extended by another use case by including the extension point name within curly braces in the flow step of the base use case. Once defined, an extension can be added as follow;

   2. {Transfer}

At step 2 in the main flow, the use case mentioned in the Transfer extension point is invoked upon successful evaluation of the extension constraint included in the referred extension point. The extension point is typically defined as follows:

Transfer [transaction = transfer] : Transfer , return: 6

Snapshots of the UCDesc windows that allow users to insert use case description details and author various flows is shown in Figure 81. A detailed user manual for the UCDesc is provided in Appendix 6



(a) (b)

. **Figure 81 UCDesc (a) Use Case Description and (b) Flow Authoring Windows**

An example use case flow description and its corresponding XMI Specification conforming to the extended use case metamodel are shown in Figure 82.

Use Case: Perform Transaction **UC-ID:** 005	<?xml version="1.0" encoding="UTF-8"?> <UseCaseModel> <Actor id="actor_0" name="Customer" type="Human" num_roles="1"/> <Actor id="actor_1" name="Bank System" type="NetworkSystem" num_roles="1"/> <UseCase actor-ref="actor_0" id="UC-005" name="Perform Transaction"
SCOPE: System **LEVEL:** Primary Task **PRIORITY:** High	

ASYNCHRONOUS

Lost Connectivity

{System is not connected}

1. Display Error Message
2. Terminate User Session
3. Use Case Ends

PRECONDITIONS

System is Connected

ACTOR

PRIMARY

Customer

SUPPORTING

Bank System

MAIN FLOW

1. *Include* **Login**
2. System Displays a list of Transactions
3. Customer Selects Transaction
4. {Transfer}
5. {Pay Bill}
6. System displays Transaction Summary
7. Use Case Ends

ALTERNATIVE FLOW

6 (a) Customer Selects Print

1. The system sends the summary to the Printer
2. Return: 6

SUCCESS POST-CONDITION

Transaction is Successful

FAILURE POST-CONDITION

Transaction failed

EXTENSION POINTS

Transfer [transaction = transfer] : Transfer , return: 6
Pay Bill [transaction = pay] : PayBill , return: 6


```xml
isAbstract="false">
        <Supporting actor-ref="actor_1"/>
        <Description scope="System" level="PrimaryTask"
Priority="high"/>
        <Precondition>
            <Constraint>
                <Entity name="System"/>
                <Relation name="equals"/>
                <Value name="Connected"/>
            </Constraint>
        </Precondition>
        <Postcondition>
            <Success>
                <Constraint>
                    <Entity name="Transaction"/>
                    <Relation name="equals"/>
                    <Value name="Successful"/>

                </Constraint>
            </Success>
            <Failure>
                <Constraint>
                    <Entity name="Transaction"/>
                    <Relation name="equals"/>
                    <Value name="Failed"/>

                </Constraint>
            </Failure>
        </Postcondition>
        <AsyncExtend name="Lost Connectivity" uc-ref="UC-
013">
                <Constraint>
                    <Entity name="System"/>
                    <Relation name="not-equals"/>
                    <Value name="connected"/>
                </Constraint>
        </AsyncExtend>
        <Include uc-ref="UC-001"/>
        <Extend uc-ref="UC-003" extPoint="Transfer"/>
        <Extend uc-ref="UC-004" extPoint="Pay Bill"/>
        <ExtensionPoint name="Transfer" lower="0"
upper="1">
                <Constraint>
                    <Entity name="transaction"/>
                    <Relation name="equals"/>
                    <Value name="transfer"/>

                </Constraint>
                <RejoinLocation step="6"/>
        </ExtensionPoint>
        <ExtensionPoint name="Pay Bill" lower="0"
upper="1">
                <Constraint>
                    <Entity name="transaction"/>
                    <Relation name="equals"/>
                    <Value name="pay"/>
                </Constraint>
                <RejoinLocation step="6"/>
        </ExtensionPoint>
    <MainFlow>
        <Transaction order="1">
            <Step step-no="1">
                <ExternalInclusion  uc-ref="UC-001"/>
            </Step>
        </Transaction>
        <Transaction order="2">
            <Step step-no="2">
                <Sender name="System"/>
                <Receiver name="Customer"/>
                <Action type="action" name="display"/>
                <Argument name="transaction"/>
            </Step>
            <Step step-no="3">
                <Sender name="Customer"/>
                <Receiver name="System"/>
                <Action type="action" name="select"/>
                <Argument name="transaction"/>
            </Step>
            <Step step-no="4">
                <ExternalVariation extPoint="Transfer"/>
```

```
            </Step>
            <Step step-no="5">
                <ExternalVariation extPoint="Pay Bill"/>
            </Step>
          <Step step-no="6">
                <Sender name="System"/>
                <Receiver name="Customer"/>
                <Action type="action" name="display"/>
                <Argument name="tranbsaction
summary"/>
          </Step>
        </Transaction>
    </MainFlow>
    <AlternativeFlow variationStep="6" sequence="a">
        <Constraint>
            <Entity name="selection"/>
            <Relation name="equals"/>
            <Value name="print"/>
        </Constraint>
        <Transaction order="1">
            <Step step-no="1">
                <Sender name="System"/>
                <Receiver name="Printer"/>
                <Action type="action" name="send"/>
                <Argument name="summary"/>
            </Step>
        </Transaction>
        <RejoinLocation step="6"/>
      </AlternativeFlow>
  </UseCase>
</UseCaseModel>
```

Figure 82 An example use case flow description and its equivalent XMI

6.1.4 Current Limitations of UCDesc Tool

Although the UCDesc tool fulfills its basic responsibility of allowing users to create and edit use case descriptions and export them to as an XMI file, the tool has some limitations for it to be used as a complete use case description tool. The tool lacks a built-in diagram rendering engine and hence require users to have an active internet connection to view a diagrammatic representation of the structural view of a use case diagram. Another limitation is the lack of a glossary function as provided by other commercial tools in the market. Inclusion of this feature will enhance the use case analysis functionality provided by UCDesc.

6.2 IntegraUML: A multi-view UML Integration and Refactoring Tool

Based on the proposed integrated metamodel, we have implemented a prototype tool called *IntegraUML* (UML Model Integration and Refactoring Tool). IntegraUML is a tool to support model integration and transformation on UML models imported in the form of an XMI file. The UML models accepted by IntegraUML are Class diagrams, Sequence diagrams and Use Case diagrams. XMI models are imported by the tool and integrated into an intermediate format, which then is used for refactoring. IntegraUML is implemented on Java programming platform and makes use of the standard XML Parser to analyze the UML models. In this section, we provide details of the implementation and usage of the IntegraUML tool.

6.2.1 IntegraUML Architecture

Figure 83 illustrates a high-level architecture of the IntegraUML tool. The inputs to the tool are XMI files representing the UML models. The format of the XMI file accepted by IntegraUML is described in the next subsection. The main engine is composed of several modules that collectively operate to integrate and refactor the input models. These modules are explained below:

1. **Integration Module:** The Integration module makes use of the standard Java XML API to parse the input models and write them to a single integrated XMI file. Particularly, the Document Object Model (DOM) API is used. DOM represents XMI as trees of nodes. A detailed description of the DOM API can be found in Appendix 7.

2. **Smell Detection Module:** Model smells in the IntegraUML tool are defined in XQuery and stored in the Model Smell Repository. XQuery is conceived as a language for querying XML files, in the same way as SQL is used for querying relational databases. The smell detection module is build using the Saxon Query processor. Each smell, in the form of a query, from the repository is executed over the integrated model. If a smell exists in the integrated model, the model along with the smell is passed on to the Refactoring module. Model smells in the repository are organized in an order to minimize any side-effects and maximize refactoring opportunity detection over the integrated model.
3. **Refactoring Module:** The refactoring module, based on the detected smell, applies a composite refactoring to remove the model smell. A composite refactoring is composed of several primitive refactorings which are applied in an error-free manner. IntegraUML is a semi-automatic refactoring tool. Hence all refactorings before application are confirmed from the user.

After executing all the smells present in the repository, the final refactored model is stored and outputted in the form of an XMI file.

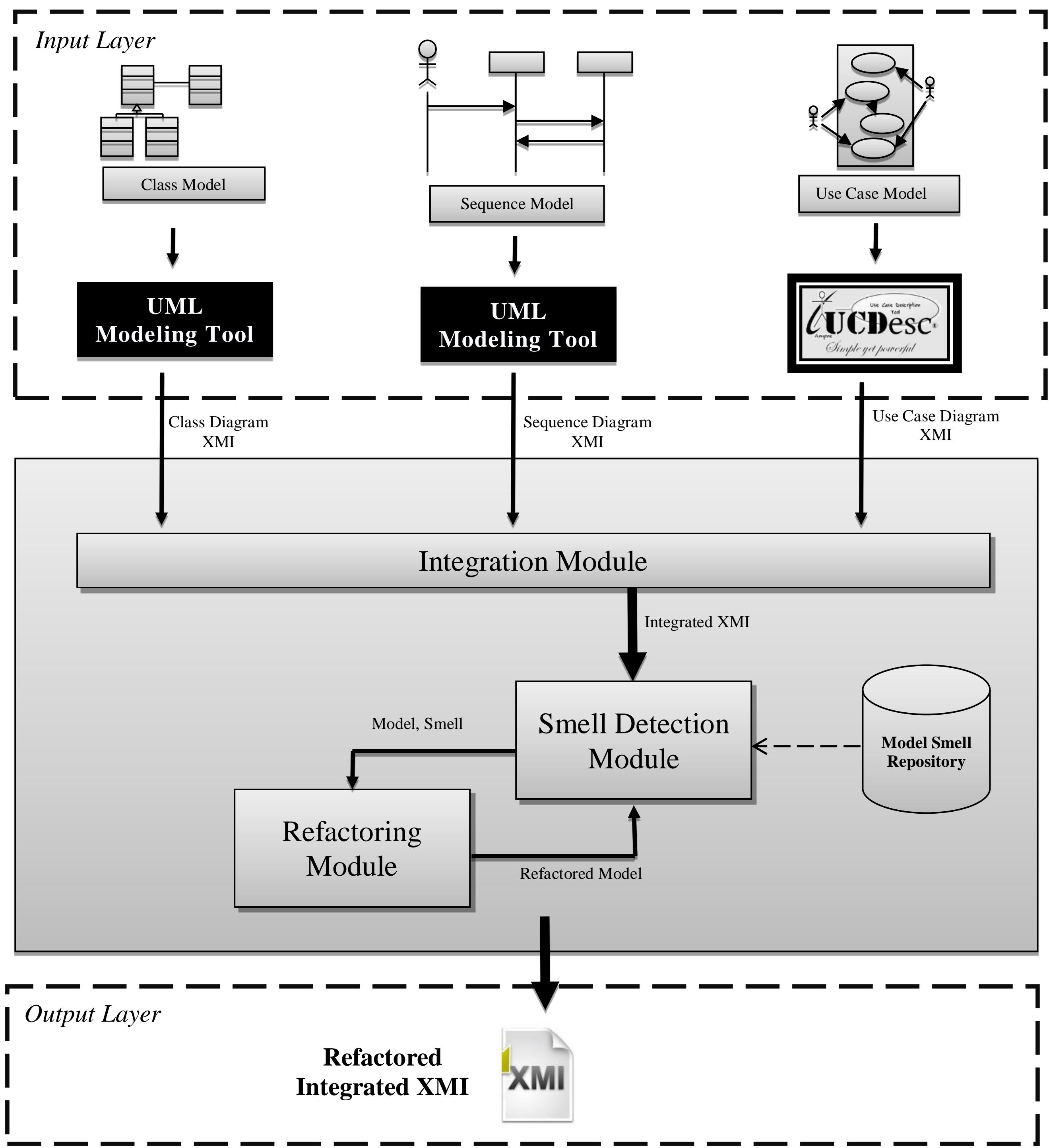


Figure 83 High-Level Architecture of the IntegraUML tool

A platform-specific mapping of the IntegraUML architecture is given in Figure 84. Different components are represented by a platform-specific view of their realization. The <<*java*>> stereotype reflects a java implementation; the <<*xmi*>> stereotype

reflects an XMI file; the <<*java-saxon*>> stereotype reflects a java implementation using the saxon xquery processor; the <<*xquery*>> stereotype reflects an xquery file; and the <<*java-class*>> stereotype reflects a compiled java class.

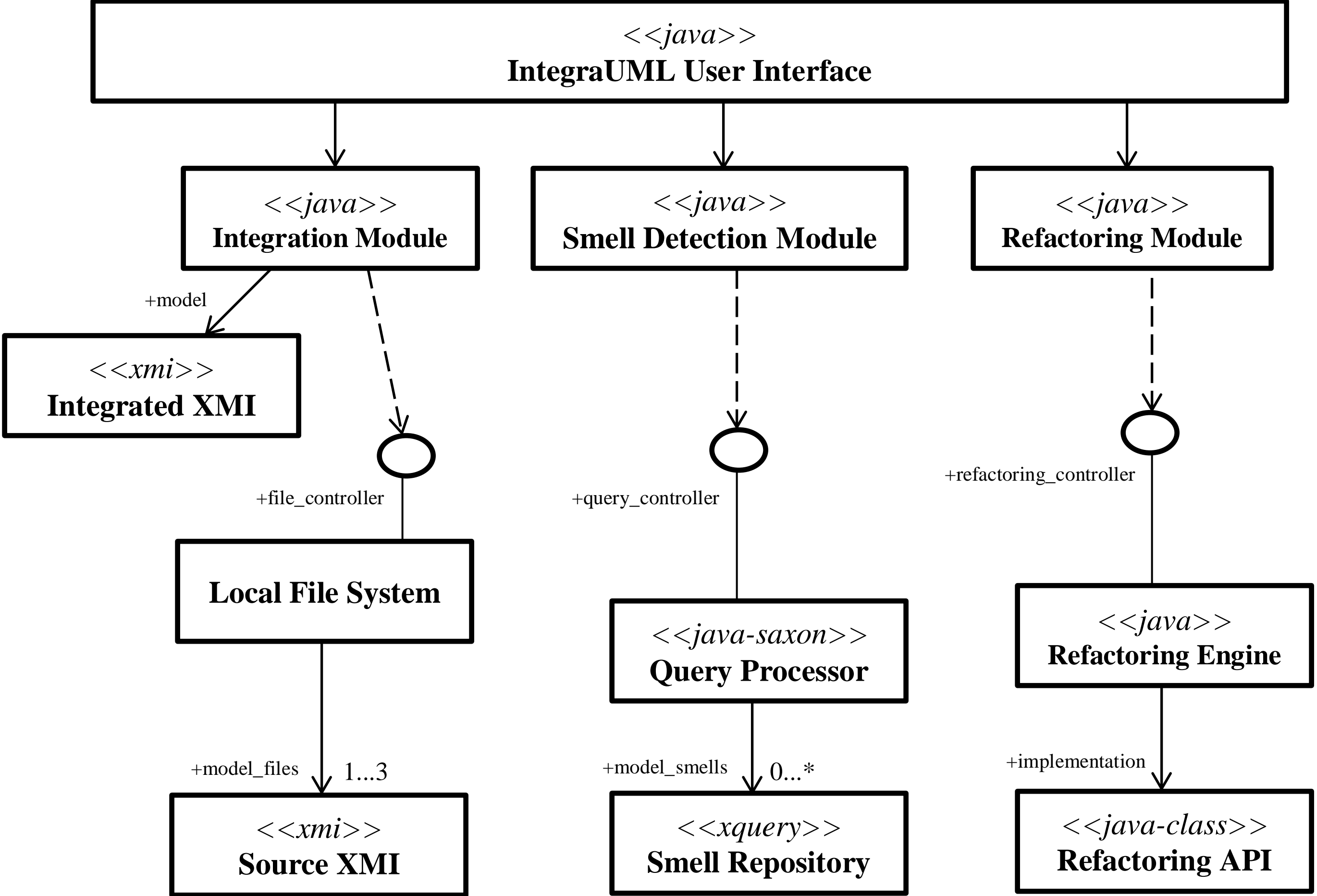


Figure 84 Platform-specific Architecture of IntegraUML

6.2.2 IntegraUML Input Format

XMI is a standard format for exchanging UML models between tools. Nonetheless, XMI-based model exchange currently has one major shortcoming: an XMI file exported from one tool is different from an XMI file exported from another tool for the same UML

model. There are many different reasons for these dissimilarities. Some prominent ones are:

- There are a number of versions of the underlying standards. For appropriate usability, the same version of MOF, XMI and UML must be used in both the exporting and importing tools.
- There are a number of ways in which a model can be serialized for export.
- The exporting tool may use a proprietary metamodel that is not based on MOF, the effect of which compromises interoperability.
- Finally, the most important one is the difference of tag names adopted by different tools.

In order to be consistent in our approach, we decided to follow the current XMI Schema Version 2.1 and UML version 2.4. An XML Schema diagram for the accepted UML models of Class and Sequence Diagrams is shown in Figure 85 and Figure 86. The XML Schema diagram for the Use Case model is given in Appendix 5 as the standard UML CASE tool exported XMI does not include its behavioral information. To the best of our knowledge, the UML CASE tool that supports these schemas (provided by OMG) is Altova's UModel [439].

The type attribute of the *packagedElement* element identifies the context of the element whether it is a package (*type="uml:Package"*), class (*type="uml:Class"*), association (*type="uml:Association"*), association class (*type="uml:AssociationClass"*) or a data type (*type="uml:DataType"*).

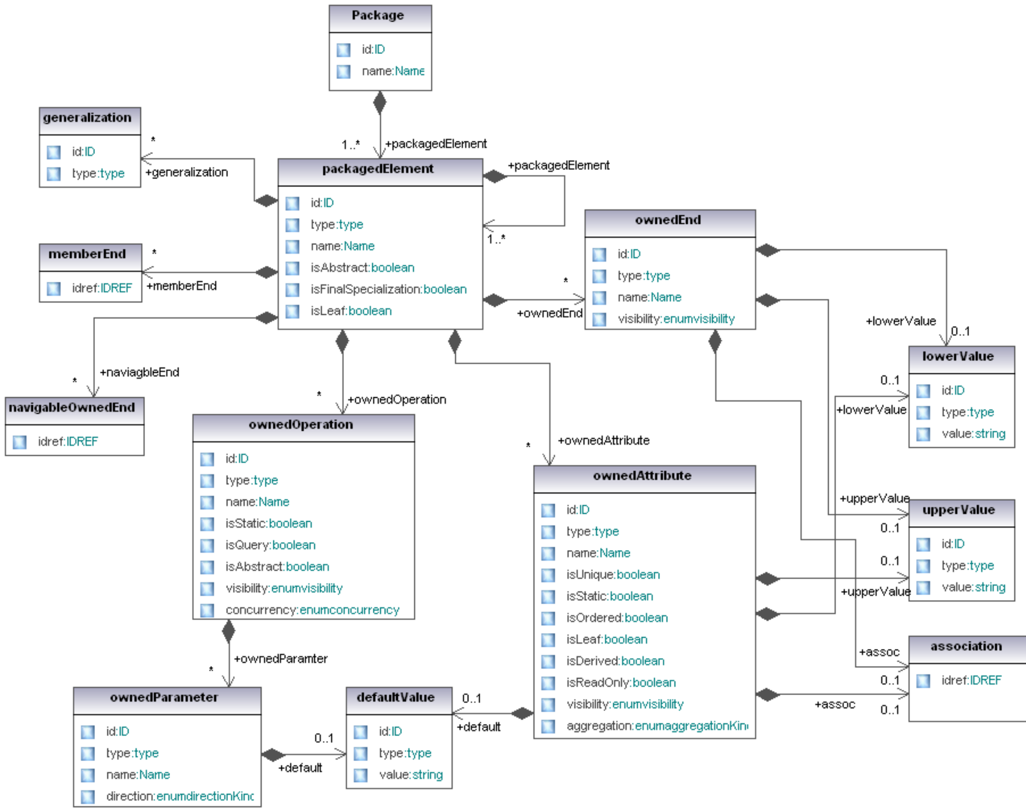


Figure 85 XML Schema Diagram of the UML Class Diagram

Similar to the class diagram schema, the type attribute of the *packagedElement* element identifies the context of the element. It can either be a package (*type="uml:Package"*), interaction (*type="uml:Interaction"*) or an event (*type="uml:CallEvent"*).

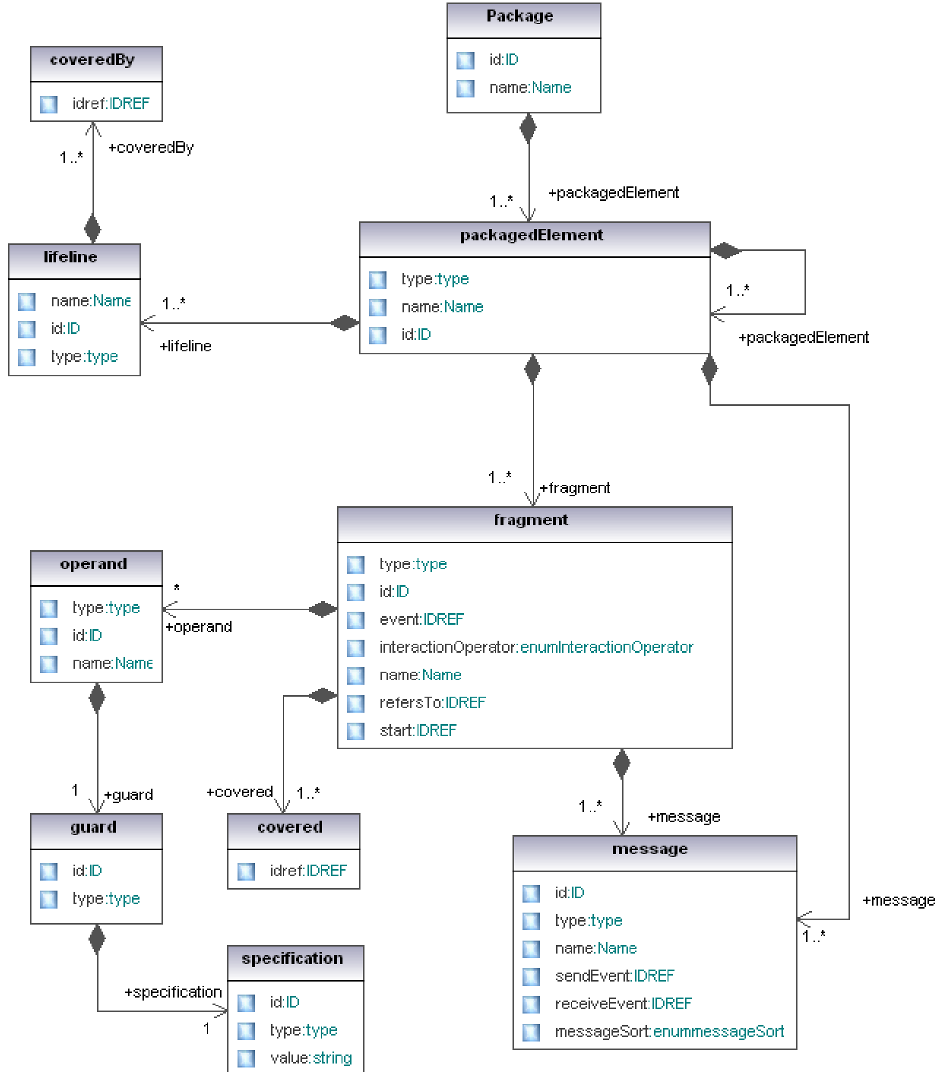


Figure 86 XML Schema Diagram of the UML Sequence Diagram

6.2.3 IntegraUML Features

IntegraUML is prototype UML model integration and refactoring tool built in java. Its main usage scenario is to import UML XMI models and generate an integrated model for the purpose of refactoring application. Figure 87 shows the high-level use cases that are most pertinent to a developer using IntegraUML.

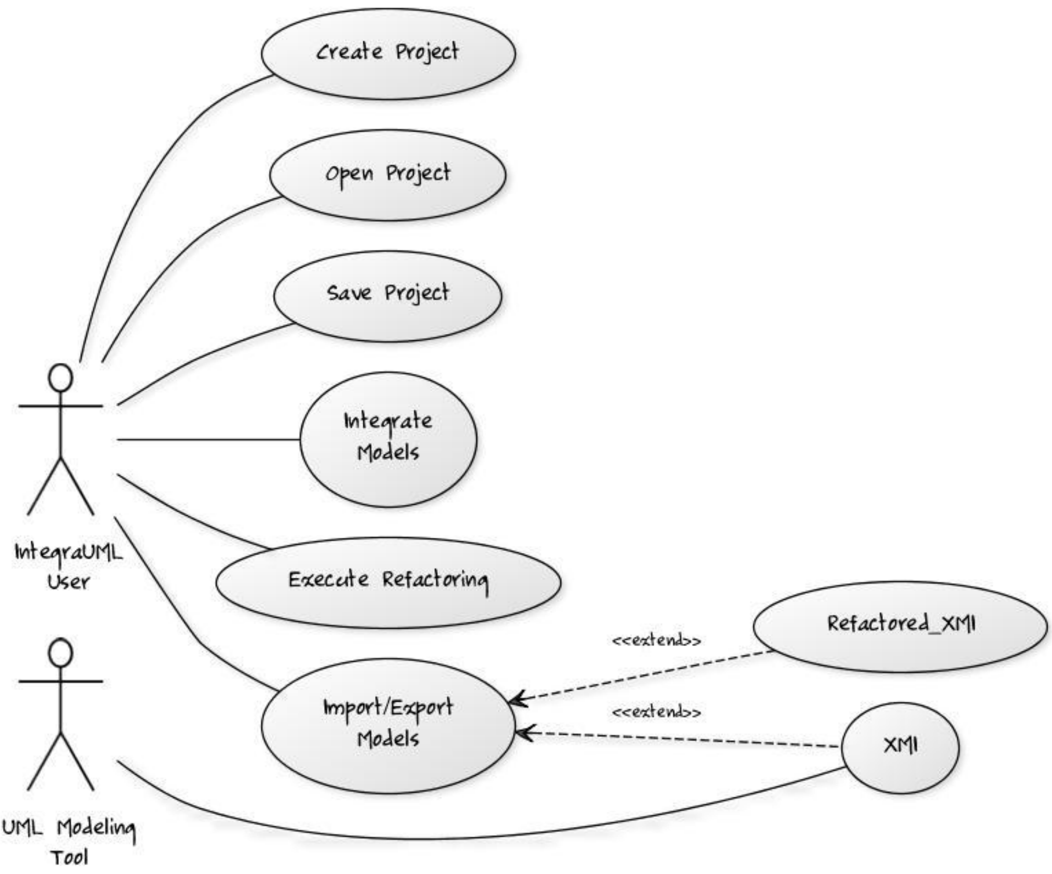


Figure 87 Use Case Diagram for IntegraUML

The main layout of IntegraUML is shown in Figure 88. It consists of a top menu bar and three panels. The Source UML Files panel is the main input panel. IntegraUML allows users to upload XMI files for the class diagram; sequence diagrams and use case diagram as individual files or combined as one. The browsing options are enabled based on the selection of an appropriate radio button at the top of the panel. XMI files can be browsed and parsed from this panel. The results of the parsing process are displayed in the Diagram Parse Log panel. Typical parse log information includes diagram version, tool exported from and statistical information like number of classes, number of interactions and so on.

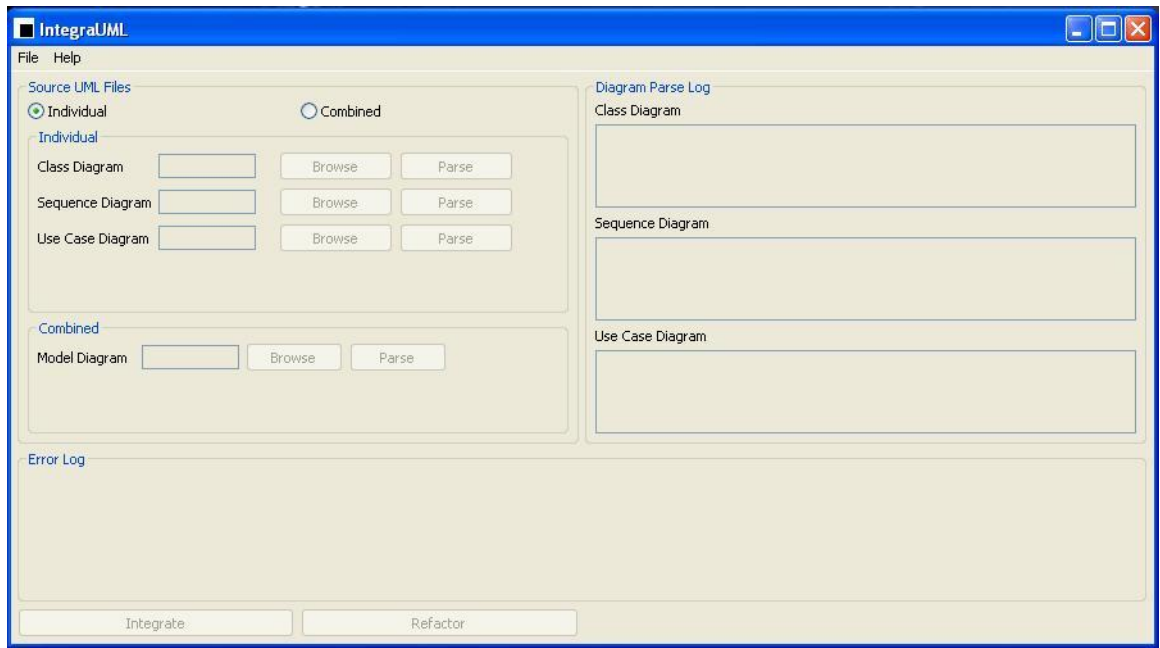


Figure 88 IntegraUML Main Layout

The Error Log panel displays any errors that occur during the model integration process. The Integrate and Refactor buttons are enabled upon successful model parsing and integration respectively. The refactoring process is an interactive one. Upon detection of a model smell, IntegraUML displays and confirms the refactoring operation from the user before its application. A detailed user manual for the IntegraUML tool is provided in Appendix 8.

6.2.4 Current Limitations of IntegraUML Tool

IntegraUML is a semi-automatic model refactoring tool. It requires the user to confirm refactoring actions before their application. A fully automated refactoring tool requires an additional module that could remember user actions and only confirm those not already applied. Another limitation of the IntegraUML tool is interoperability. As the output of IntegraUML is based on a proprietary metamodel, developed as part of this work, using it with other UML modeling tools is not suitable. Although this could be circumvented by

using a model disintegration module, which disintegrates the refactored model into class, sequence and use case diagrams, and then using XSLT transformation to map the resultant XMIs to a particular tools requirement. This is put forward as a future work to the IntegraUML tool. Finally, IntegraUML accepts a particular format of XMI as input to the tool. As there are myriad formats of XMI available for UML models, providing support for each is difficult to achieve.

CHAPTER 7

VALIDATION

In this chapter, we establish a framework to evaluate the effect of refactoring the integrated model proposed in this work. Initially, we construct a validation framework to evaluate our approach against existing approaches in Section 7.1. Then we describe and summarize the case studies used for validation in Section 7.2. Finally, baseline is established by evaluating existing approaches from the literature over the case studies in Section 7.3. Baseline results are compared, analyzed and discussed thoroughly with our proposed approach in the next chapter.

7.1 Validation Framework

An important objective of model-driven refactoring is to show the effect of refactoring on quality of the software model. Although one of the most important activity in the refactoring process, it is addressed only by a few published studies. Lack of an evaluation approach severely affects the usability of model refactoring approaches in the industrial software development. It is evident that despite being one of the most important activities, it is still in its infancy.

The only available approach used by the proposed studies [21, 32, 451] use model metrics and compare these metrics before and after the application of refactoring to validate their approach. Lange and Chaudron [69] developed a quality model for UML.

Jalbani et al. [294] proposed an integrated quality engineering approach for UML models. They divided their approach into two parts: Quality Assessment and Quality Improvement. Quality assessment includes the Quality Model for UML based on the Lange and Chaudron model and metrics for UML. Quality Improvement includes model smell detection and model refactoring. The framework developed by Jalbani et al. is still in development phase.

In this work, we also used model metrics to validate our proposed approach. We initially established an acceptable suite of UML model metrics for class, sequence and use case diagrams proposed in the literature. These metrics are catalogued in Appendix 9. The complete validation framework is depicted in Figure 89.

The validation process is carried out in two phases. In phase 1, also referred in this work as Individual Refactoring, UML models specifically those used in this work are refactored separately. An intermediate parser based on Java is used to convert the imported models (in XMI) to an intermediate XMI format. The main motivation behind this conversion includes:

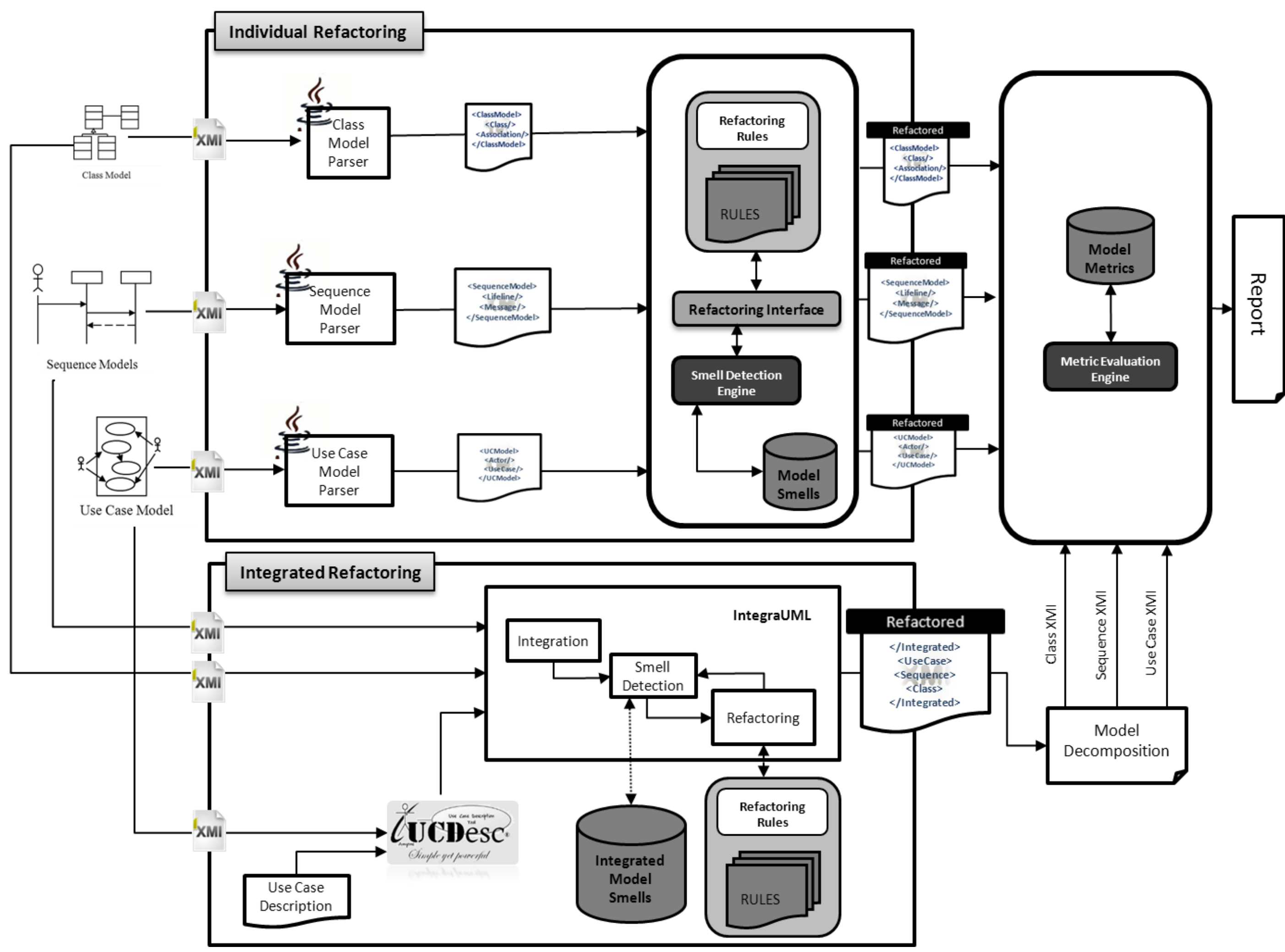


Figure 89 Validation Framework

1. The intermediate model provides a simple lexical view into the UML model.
2. The XMI file exported by major UML Modeling tools includes deeper nesting and a significant amount of cross-references. This in turn makes the models harder to read and navigate. The intermediate model used in our approach removes the deep nesting by resolving diagram related attributes and including only relevant information as tags. Cross-references are also resolved by replacing IDREF and type elements within the XMI with their element names. This makes the intermediate model faster to process, refactor and evaluate.

The intermediate models are then refactored. The refactoring rules and model smells for all the diagrams are obtained from published literature in the field of model-driven refactoring. A catalog of all the supported refactoring rules and models smells is included as part of Appendix 2 and Appendix 10 respectively. Model metrics collected as part of the framework are then applied to these refactored models. These metrics are used as baselines to be compared with the Integrated Refactoring approach proposed in this work.

In phase two, the same models are integrated using the proposed IntegraUML prototype tool. Since the approach requires behavioral information of the use case diagram in order to construct the integrated model, UCDesc is used. As part of the Integrated Refactoring process, the individual models are initially checked for consistency. If inconsistencies are found, the integration process is stopped and the refactoring task is terminated. After the consistency check step, models are integrated and checked for syntactic and semantic compatibility against the Integrated Metamodel. Integrated model smells proposed in this work are then applied over the integrated model to detect refactoring opportunity. The

model is refactored if a model smell is detected. This process is repeated until all the smells included in the Integrated Model Smell repository are exhausted. The output of this phase is a refactored integrated model. In order to evaluate the model, it has to be decomposed into individual models. The Model Decomposition package takes as input the integrated model and outputs individual class, sequence and use case model. No information is lost during the decomposition step. Similar to the previous phase, model metrics are applied to these refactored models. In order to evaluate the effectiveness of the proposed approach, metric values for individual refactoring and integrated refactoring are compared. In this chapter, we perform phase one of the validation approach. Phase 2 is performed and analyzed in the next chapter.

7.2 Case Studies

A major challenge encountered when working with UML based techniques is the availability of quality case studies. Since the UML Class diagram is the most commonly used diagram, most case studies only provide system design through class diagrams. This constrains multi-model approaches such as the one proposed in this work. Hence, to overcome this issue, we decided to construct a suite of case studies collected from two separate yet distinct origin and domain. We used nine different software design case studies to evaluate our approach: five obtained from student projects and four obtained from published research, case studies and industrial white papers.

7.2.1 Student Projects

The case studies from student projects are supplied by a group of undergraduate B.Sc. students with software engineering major of study. The models are the design models of the group's senior project conducted at King Fahd University of Petroleum and Minerals, whose stakeholders are industrial organizations. Out of 16 projects considered, five were selected based on the criteria summarized below.

1. Project is complete and includes all required diagrams: class, sequence, use case diagrams and use case description.
2. Project uses UML 2 concepts such as Combined Fragments in sequence diagrams.
3. Information across all diagrams is consistent and properly documented.
4. Projects scored good grades for design and implementation from the evaluator and the stakeholder.

In the rest of chapter, these case studies or projects are referred to as OFD (Online Form Designer), ESAP (Electronic Student Academic Portfolio), FOMS (Freelancing Online Management System), OG (Our Goal) and ME (My Events). Table 8 summarizes some vital characteristics of each of the student project case studies: use cases, actors, classes, average number of lifelines per sequence diagram, average number of messages per sequence diagram, total number of combined fragments used and total number of interaction use fragments used in the sequence diagram.

Table 8 Summary of each student project case study system

Case Study	# of Use Cases	# of Actors	# of Classes	Avg. # of Lifelines	Avg. # of Messages	# of Combined Fragments	# of Interaction Use
OFD	22	2	23	4	8	14	10
OG	35	7	15	5	9	11	0
ESAP	39	6	28	4	8	19	24
ME	62	9	18	4	11	39	4
FOMS	35	5	16	3	5	0	0

7.2.2 Published Case Studies

Case studies in this category are gathered from two different origins and domains: small-scale industrial systems and published case studies. In the rest of the chapters, these case studies are referred to as ATM (Automated Teller Machine), ORA (On-Road Assistance), SCM (Supply Chain Management System), and O-Comm (OS-Commerce). All these system designs came from multiple sources. ATM is a well-known case study in the field of UML based empirical studies published by Briand et al. [452, 453]. SCM [454], and O-Comm [455] are published case studies and ORA system [456, 457] is part of small-scale industrial system design. Table 9 summarizes some vital characteristics of each of the published case studies.

Table 9 Summary of each published case study system

Case Study	# of Use Cases	# of Actors	# of Classes	Avg. # of Lifelines	Avg. # of Messages	# of Combined Fragments	# of Interaction Use
ATM	15	2	18	4	7	15	0
SCM	8	5	21	4	5	7	3
O-Comm	119	5	59	3	6	32	5
ORA	13	4	14	4	4	1	11

7.3 Individual Refactoring

7.3.1 OFD (Online Form Designer)

(a) Class Diagram

Table 10 Comparison of Class Diagram-level Metrics for OFD System

Metrics	Before Refactoring	After Refactoring
Number of The Classes **(NCM)**	23	21
Number of The Associations **(NASM)**	12	10
Number of The Aggregations **(NAGM)**	6	6
Number of The Inheritance Relations **(NIM)**	10	10

Table 11 Comparison of Class Element-level Metrics for OFD System

Metrics		Minimum	Maximum	Total	Average
Depth of Inheritance **(DIT)**	**Before**	0	2	15	0.65
	After	0	2	15	0.71
Number of Children **(NOC)**	**Before**	0	4	10	0.43
	After	0	4	10	0.48
Fan-In	**Before**	0	3	21	0.91
	After	0	3	18	0.86
Fan-out	**Before**	0	3	21	0.91
	After	0	3	18	0.86
# of Associations Linked to a Class **(NASC)**	**Before**	0	5	36	1.57
	After	0	5	32	1.52
# of Attributes in a Class Unweighted **(NATC1)**	**Before**	0	10	58	2.52
	After	0	10	58	2.76
# of Attributes in a Class Weighted **(NATC2)**	**Before**	0	3	11.5	0.50
	After	0	3	11.5	0.55
# of Operations in a Class Unweighted **(NOPC1)**	**Before**	0	25	127	5.52
	After	0	29	127	6.05
# of Operations in a Class Weighted **(NOPC2)**	**Before**	0	25	127	5.52
	After	0	29	127	6.05
# of Super Classes of a Class **(NSUPC)**	**Before**	0	1	10	0.43
	After	0	1	10	0.48
# of Elements in the Transitive Closure of the Super Classes of a Class **(NSUPC*)**	**Before**	0	2	15	0.65
	After	0	2	15	0.71

Discussion: Two instances of the Lazy Class Model Smell were found in the class diagram of OFD case study. Resolution of this smell although reduced the size of the design (in terms of number of classes) as shown in Table 10, but added more methods to the existing classes making them rich "*God Classes*" as evident by the NOPC1 and NOPC2 metric values in Table 11.

(b) Use Case Diagram

Table 12 Comparison of Use Case Diagram-level Metrics for OFD System

Metrics	Before Refactoring	After Refactoring
Number of Use Cases **(NUM)**	22	22
Number of Actors **(NAM)**	2	2

Table 13 Comparison of Use Case Element-level Metrics for OFD System

Metrics		Minimum	Maximum	Total	Average
# of Use Cases which this Extends **(EXTENDING)**	**Before**	0	7	14	0.64
	After	0	7	14	0.64
# of Use Cases which Extend this Use Case **(EXTENDED)**	**Before**	0	2	14	0.64
	After	0	2	14	0.64
# of Use Cases which this Includes **(INCLUDING)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
# of Use Cases which Includes this Use Case **(INCLUDED)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
# of Extension Points of The Use Case **(ExtPts)**	**Before**	0	7	14	0.64
	After	0	7	14	0.64
Depth of <<Include>> Relationship **(DOIR)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
Depth of <<Extend>> Relationship **(DOER)**	**Before**	0	1	7	0.32
	After	0	1	7	0.32

Discussion: No instances of any use case model smell were found for the OFD case study. Hence, the values remain unchanged before and after the application of refactoring as evident from Table 12 and Table 13.

(c) Sequence Diagram

Table 14 Comparison of Sequence Element-level Metrics for OFD System

Metrics		Minimum	Maximum	Total	Average
# of Lifelines **(LIFELINES)**	**Before**	3	5	92	4.18
	After	3	5	92	4.18
# of Messages **(NMM)**	**Before**	5	15	169	7.68
	After	5	15	169	7.68
# of Messages sent by the Instantiated Objects of a Class **(NMSC)**	**Before**	1	6	169	1.59
	After	1	6	169	1.59
# of Messages received by the Instantiated Objects of a Class **(NMRC)**	**Before**	0	8	169	1.52
	After	0	8	169	1.52

Discussion: Although quite a few instances of the middle man smell were found in the sequence models of OFD, but refactoring was not performed due to the use of MVC pattern in their application. Hence, the metric values shown in Table 14 remain unchanged.

7.3.2 OG (OurGoal)

(a) Class Diagram

Table 15 Comparison of Class Diagram-level Metrics for OG System

Metrics	Before Refactoring	After Refactoring
Number of The Classes **(NCM)**	15	15
Number of The Associations **(NASM)**	14	14
Number of The Aggregations **(NAGM)**	10	10
Number of The Inheritance Relations **(NIM)**	4	4

Table 16 Comparison of Class Element-level Metrics for OG System

Metrics		Minimum	Maximum	Total	Average
Depth of Inheritance **(DIT)**	**Before**	0	1	4	0.27
	After	0	1	4	0.27
Number of Children **(NOC)**	**Before**	0	2	4	0.27
	After	0	2	4	0.27
Fan-In	**Before**	0	12	27	1.80
	After	0	12	27	1.80
Fan-out	**Before**	0	12	27	1.80
	After	0	12	27	1.80
# of Associations Linked to a Class **(NASC)**	**Before**	0	12	48	3.20
	After	0	12	48	3.20
# of Attributes in a Class Unweighted **(NATC1)**	**Before**	0	19	81	5.40
	After	0	19	81	5.40
# of Attributes in a Class Weighted **(NATC2)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
# of Operations in a Class Unweighted **(NOPC1)**	**Before**	0	17	42	2.80
	After	0	12	42	2.80
# of Operations in a Class Weighted **(NOPC2)**	**Before**	0	17	42	2.80
	After	0	12	42	2.80
# of Super Classes of a Class **(NSUPC)**	**Before**	0	1	4	0.27
	After	0	1	4	0.27
# of Elements in the Transitive Closure of the Super Classes of a Class **(NSUPC*)**	**Before**	0	1	4	0.27
	After	0	1	4	0.27

Discussion: Seven instances of the Data Class Model Smell were found in the class diagram of OG case study. Resolution of this smell reduced the maximum number of operations in a class (*NOPC1* and *NOPC2*) by moving related methods to the respective data classes from the behaviorally rich *Profile* class in the model as observed from Table 16.

(b) Use Case Diagram

Table 17 Comparison of Use Case Diagram-level Metrics for OG System

Metrics	Before Refactoring	After Refactoring
Number of Use Cases **(NUM)**	35	35
Number of Actors **(NAM)**	7	7

Table 18 Comparison of Use Case Element-level Metrics for OG System

Metrics		Minimum	Maximum	Total	Average
# of Use Cases which this Extends **(EXTENDING)**	**Before**	0	0	0	0
	After	0	0	0	0
# of Use Cases which Extend this Use Case **(EXTENDED)**	**Before**	0	0	0	0
	After	0	0	0	0
# of Use Cases which this Includes **(INCLUDING)**	**Before**	0	2	3	0.09
	After	0	2	3	0.09
# of Use Cases which Includes this Use Case **(INCLUDED)**	**Before**	0	2	3	0.09
	After	0	2	3	0.09
# of Extension Points of The Use Case **(ExtPts)**	**Before**	0	0	0	0
	After	0	0	0	0
Depth of <<Include>> Relationship **(DOIR)**	**Before**	0	1	2	0.06
	After	0	1	2	0.06
Depth of <<Extend>> Relationship **(DOER)**	**Before**	0	0	0	0
	After	0	0	0	0

Discussion: No instances of any use case model smell were found for the OG case study. Hence, the values remain unchanged before and after the application of refactoring as shown in Table 17 and Table 18.

(c) Sequence Diagram

Table 19 Comparison of Sequence Element-level Metrics for OG System

Metrics		Minimum	Maximum	Total	Average
# of Lifelines **(LIFELINES)**	**Before**	4	6	175	5.00
	After	4	6	175	5.00
# of Messages **(NMM)**	**Before**	3	13	333	9.51
	After	3	13	333	9.51
# of Messages sent by the Instantiated Objects of a Class **(NMSC)**	**Before**	0	5	315	1.54
	After	0	5	315	1.54
# of Messages received by the Instantiated Objects of a Class **(NMRC)**	**Before**	1	5	313	1.53
	After	1	5	313	1.53

Discussion: Although quite a few instances of the middle man smell were found in the sequence models of OG, but refactoring was not performed due to the use of MVC pattern in their application as evident from metric values shown in Table 19.

7.3.3 ESAP (Electronic Student Academic Portfolio)

(a) Class Diagram

Table 20 Comparison of Class Diagram-level Metrics for ESAP System

Metrics	Before Refactoring	After Refactoring
Number of The Classes **(NCM)**	28	28
Number of The Associations **(NASM)**	37	37
Number of The Aggregations **(NAGM)**	6	6
Number of The Inheritance Relations **(NIM)**	7	7

Table 21 Comparison of Class Element-level Metrics for ESAP System

Metrics		Minimum	Maximum	Total	Average
Depth of Inheritance **(DIT)**	**Before**	0	3	12	0.43
	After	0	3	12	0.43
Number of Children (**NOC**)	**Before**	0	3	7	0.25
	After	0	3	7	0.25
Fan-In	**Before**	1	8	78	2.79
	After	1	8	78	2.79
Fan-out	**Before**	1	8	78	2.79
	After	1	8	78	2.79
# of Associations Linked to a Class **(NASC)**	**Before**	1	8	85	3.04
	After	1	8	85	3.04
# of Attributes in a Class Unweighted **(NATC1)**	**Before**	0	10	78	2.79
	After	0	10	78	2.79
# of Attributes in a Class Weighted **(NATC2)**	**Before**	0	1	1	0.04
	After	0	1	1	0.04
# of Operations in a Class Unweighted **(NOPC1)**	**Before**	0	63	253	9.04
	After	0	53	240	8.57
# of Operations in a Class Weighted **(NOPC2)**	**Before**	0	63	253	9.04
	After	0	53	240	8.57
# of Super Classes of a Class **(NSUPC)**	**Before**	0	1	7	0.25
	After	0	1	7	0.25
# of Elements in the Transitive Closure of the Super Classes of a Class **(NSUPC*)**	**Before**	0	3	12	0.43
	After	0	3	12	0.43

Discussion: Ten instances of the Duplication Model Smell were found in the class diagram of ESAP case study. Resolution of this smell reduced the maximum number of operations in a class (*NOPC1* and *NOPC2*) by moving duplicated methods to their respective super classes as evident from Table 21.

(b) Use Case Diagram

Table 22 Comparison of Use Case Diagram-level Metrics for ESAP System

Metrics	Before Refactoring	After Refactoring
Number of Use Cases **(NUM)**	39	39
Number of Actors **(NAM)**	6	6

Table 23 Comparison of Use Case Element-level Metrics for ESAP System

Metrics		Minimum	Maximum	Total	Average
# of Use Cases which this Extends **(EXTENDING)**	**Before**	0	5	27	0.69
	After	0	5	27	0.69
# of Use Cases which Extend this Use Case **(EXTENDED)**	**Before**	0	2	27	0.69
	After	0	2	27	0.69
# of Use Cases which this Includes **(INCLUDING)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
# of Use Cases which Includes this Use Case **(INCLUDED)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
# of Extension Points of The Use Case **(ExtPts)**	**Before**	0	5	27	0.69
	After	0	5	27	0.69
Depth of <<Include>> Relationship **(DOIR)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
Depth of <<Extend>> Relationship **(DOER)**	**Before**	0	3	37	0.95
	After	0	3	37	0.95

Discussion: No instances of any use case model smell were found for the ESAP case study. Hence, the values remain unchanged before and after the application of refactoring as shown in Table 22 and Table 23.

(c) Sequence Diagram

Table 24 Comparison of Sequence Element-level Metrics for ESAP System

Metrics		Minimum	Maximum	Total	Average
# of Lifelines **(LIFELINES)**	**Before**	0	6	155	3.97
	After	0	6	155	3.97
# of Messages **(NMM)**	**Before**	0	20	296	7.59
	After	0	20	296	7.59
# of Messages sent by the Instantiated Objects of a Class **(NMSC)**	**Before**	1	9	292	1.87
	After	1	9	292	1.87
# of Messages received by the Instantiated Objects of a Class **(NMRC)**	**Before**	1	9	293	1.83
	After	1	9	293	1.83

Discussion: The ESAP case study included a number of packages to cluster related functionality together in the class diagram. Because of this, the *Façade* design pattern was used for communication between these packages. Although a Façade class may seem as a middle man lifeline in the sequence diagram, its main purpose is to allow a single point access to entity classes in the model. Hence, the middle man refactoring was not applied to the ESAP sequence diagrams. This is again evident from the sequence diagram metrics presented in Table 24.

7.3.4 ME (MyEvents)

(a) Class Diagram

Table 25 Comparison of Class Diagram-level Metrics for ME System

Metrics	Before Refactoring	After Refactoring
Number of The Classes **(NCM)**	18	15
Number of The Associations **(NASM)**	13	12
Number of The Aggregations **(NAGM)**	15	14
Number of The Inheritance Relations **(NIM)**	6	5

Table 26 Comparison of Class Element-level Metrics for ME System

Metrics		Minimum	Maximum	Total	Average
Depth of Inheritance **(DIT)**	**Before**	0	2	7	0.39
	After	0	2	6	0.40
Number of Children (**NOC**)	**Before**	0	2	6	0.33
	After	0	2	5	0.33
Fan-In	**Before**	0	6	26	1.44
	After	0	6	24	1.60
Fan-out	**Before**	0	6	26	1.44
	After	0	6	24	1.60
# of Associations Linked to a Class **(NASC)**	**Before**	1	9	54	3.00
	After	1	9	50	3.33
# of Attributes in a Class Unweighted **(NATC1)**	**Before**	0	16	103	5.72
	After	1	16	101	6.73
# of Attributes in a Class Weighted **(NATC2)**	**Before**	0	2	3	0.17
	After	0	2	3	0.20
# of Operations in a Class Unweighted **(NOPC1)**	**Before**	0	18	105	5.83
	After	1	18	103	6.87
# of Operations in a Class Weighted **(NOPC2)**	**Before**	0	18	105	5.83
	After	1	18	103	6.87
# of Super Classes of a Class **(NSUPC)**	**Before**	0	1	6	0.33
	After	0	1	5	0.33
# of Elements in the Transitive Closure of the Super Classes of a Class **(NSUPC*)**	**Before**	0	2	7	0.39
	After	0	2	6	0.40

Discussion: Two instances of the Functional Decomposition Model Smell and one instance of the Refused Bequest Model Smell were found in the class diagram of ME case study. Removal of the functionally decomposed classes that resulted in improved coupling (see Table 26) and the overall design size in terms of number of classes (see Table 25). Similarly, the removal of the sub-class with no functionality also affected the design size of the class model.

(b) Use Case Diagram

Table 27 Comparison of Use Case Diagram-level Metrics for ME System

Metrics	Before Refactoring	After Refactoring
Number of Use Cases **(NUM)**	62	62
Number of Actors **(NAM)**	9	9

Table 28 Comparison of Use Case Element-level Metrics for ME System

Metrics		Minimum	Maximum	Total	Average
# of Use Cases which this Extends **(EXTENDING)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
# of Use Cases which Extend this Use Case **(EXTENDED)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
# of Use Cases which this Includes **(INCLUDING)**	**Before**	0	1	2	0.03
	After	0	1	2	0.03
# of Use Cases which Includes this Use Case **(INCLUDED)**	**Before**	0	1	2	0.03
	After	0	1	2	0.03
# of Extension Points of The Use Case **(ExtPts)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
Depth of <<Include>> Relationship **(DOIR)**	**Before**	0	1	2	0.03
	After	0	1	2	0.03
Depth of <<Extend>> Relationship **(DOER)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00

Discussion: No instances of any use case model smell were found for the ME case study. Hence, the values remain unchanged before and after the application of refactoring as shown in Table 27 and Table 28.

(c) Sequence Diagram

Table 29 Comparison of Sequence Element-level Metrics for ME System

Metrics		Minimum	Maximum	Total	Average
# of Lifelines **(LIFELINES)**	**Before**	2	6	260	4.19
	After	2	6	260	4.19
# of Messages **(NMM)**	**Before**	2	25	703	11.34
	After	2	25	703	11.34
# of Messages sent by the Instantiated Objects of a Class **(NMSC)**	**Before**	1	10	678	1.80
	After	1	10	678	1.80
# of Messages received by the Instantiated Objects of a Class **(NMRC)**	**Before**	1	10	674	1.59
	After	1	10	674	1.59

Discussion: Although quite a few instances of the middle man smell were found in the sequence models of ME, but refactoring was not performed due to the use of MVC pattern in their application as evident from metric values shown in Table 29.

7.3.5 FOMS (Freelancing Online Management System)

(a) Class Diagram

Table 30 Comparison of Class Diagram-level Metrics for FOMS System

Metrics	Before Refactoring	After Refactoring
Number of The Classes **(NCM)**	16	16
Number of The Associations **(NASM)**	8	8
Number of The Aggregations **(NAGM)**	10	10
Number of The Inheritance Relations **(NIM)**	7	7

Table 31 Comparison of Class Element-level Metrics for FOMS System

Metrics		Minimum	Maximum	Total	Average
Depth of Inheritance **(DIT)**	**Before**	0	2	9	0.56
	After	0	2	9	0.56
Number of Children (**NOC**)	**Before**	0	5	7	0.44
	After	0	5	7	0.44
Fan-In	**Before**	0	6	21	1.31
	After	0	6	21	1.31
Fan-out	**Before**	0	6	21	1.31
	After	0	6	21	1.31
# of Associations Linked to a Class **(NASC)**	**Before**	0	9	36	2.25
	After	0	9	36	2.25
# of Attributes in a Class Unweighted **(NATC1)**	**Before**	0	16	84	5.25
	After	0	18	82	5.13
# of Attributes in a Class Weighted **(NATC2)**	**Before**	0	2.5	2.5	0.16
	After	0	2.5	2.5	0.16
# of Operations in a Class Unweighted **(NOPC1)**	**Before**	1	39	174	10.88
	After	1	39	174	10.88
# of Operations in a Class Weighted **(NOPC2)**	**Before**	1	37	156	9.75
	After	1	37	156	9.75
# of Super Classes of a Class **(NSUPC)**	**Before**	0	1	7	0.44
	After	0	1	7	0.44
# of Elements in the Transitive Closure of the Super Classes of a Class **(NSUPC*)**	**Before**	0	2	9	0.56
	After	0	2	9	0.56

Discussion: Two instances of the Duplication Model Smell were found in the class diagram of FOMS case study. Resolution of this smell reduced the maximum number of attributes in a class (*NATC1*) by moving duplicated attributes to their respective super classes as shown in Table 31.

(b) Use Case Diagram

Table 32 Comparison of Use Case Diagram-level Metrics for FOMS System

Metrics	Before Refactoring	After Refactoring
Number of Use Cases **(NUM)**	35	35
Number of Actors **(NAM)**	5	5

Table 33 Comparison of Use Case Element-level Metrics for FOMS System

Metrics		Minimum	Maximum	Total	Average
# of Use Cases which this Extends **(EXTENDING)**	**Before**	0	1	1	0.03
	After	0	1	1	0.03
# of Use Cases which Extend this Use Case **(EXTENDED)**	**Before**	0	1	1	0.03
	After	0	1	1	0.03
# of Use Cases which this Includes **(INCLUDING)**	**Before**	0	1	4	0.11
	After	0	1	4	0.11
# of Use Cases which Includes this Use Case **(INCLUDED)**	**Before**	0	3	4	0.11
	After	0	3	4	0.11
# of Extension Points of The Use Case **(ExtPts)**	**Before**	0	1	1	0.03
	After	0	1	1	0.03
Depth of <<Include>> Relationship **(DOIR)**	**Before**	0	1	2	0.06
	After	0	1	2	0.06
Depth of <<Extend>> Relationship **(DOER)**	**Before**	0	1	1	0.03
	After	0	1	1	0.03

Discussion: No instances of any use case model smell were found for the FOMS case study. Hence, the values remain unchanged before and after the application of refactoring as shown in Table 32 and Table 33.

(c) Sequence Diagram

Table 34 Comparison of Sequence Element-level Metrics for FOMS System

Metrics		Minimum	Maximum	Total	Average
# of Lifelines **(LIFELINES)**	**Before**	2	8	109	3.11
	After	2	8	109	3.11
# of Messages **(NMM)**	**Before**	1	18	162	4.63
	After	1	18	162	4.63
# of Messages sent by the Instantiated Objects of a Class **(NMSC)**	**Before**	1	11	148	1.40
	After	1	11	148	1.40
# of Messages received by the Instantiated Objects of a Class **(NMRC)**	**Before**	1	9	154	1.62
	After	1	9	154	1.62

Discussion: Although quite a few instances of the middle man smell were found in the sequence models of ME, but refactoring was not performed due to the use of MVC pattern in their application as shown in Table 34.

7.3.6 ATM (Automated Teller Machine)

(a) Class Diagram

Table 35 Comparison of Class Diagram-level Metrics for ATM System

Metrics	Before Refactoring	After Refactoring
Number of The Classes **(NCM)**	18	14
Number of The Associations **(NASM)**	3	3
Number of The Aggregations **(NAGM)**	8	8
Number of The Inheritance Relations **(NIM)**	6	2

Table 36 Comparison of Class Element-level Metrics for ATM System

Metrics		Minimum	Maximum	Total	Average
Depth of Inheritance **(DIT)**	**Before**	0	1	6	0.33
	After	0	1	2	0.14
Number of Children **(NOC)**	**Before**	0	4	6	0.33
	After	0	2	2	0.14
Fan-In	**Before**	0	2	11	0.61
	After	0	2	11	0.79
Fan-out	**Before**	0	2	11	0.61
	After	0	2	11	0.79
# of Associations Linked to a Class **(NASC)**	**Before**	0	8	22	1.22
	After	0	8	22	1.57
# of Attributes in a Class Unweighted **(NATC1)**	**Before**	0	14	29	1.61
	After	0	14	29	2.07
# of Attributes in a Class Weighted **(NATC2)**	**Before**	0	3	4	0.22
	After	0	3	4	0.29
# of Operations in a Class Unweighted **(NOPC1)**	**Before**	0	14	119	6.61
	After	0	14	77	5.50
# of Operations in a Class Weighted **(NOPC2)**	**Before**	0	14	119	6.61
	After	0	14	77	5.50
# of Super Classes of a Class **(NSUPC)**	**Before**	0	1	6	0.33
	After	0	1	2	0.14
# of Elements in the Transitive Closure of the Super Classes of a Class **(NSUPC*)**	**Before**	0	1	6	0.33
	After	0	1	2	0.14

Discussion: Two instances of the Duplication Model Smell were found in the class diagram of ATM case study. The duplicated methods were moved from all the subclasses to their super class. Removal of the duplicated functionality resulted in the child classes being empty. Hence, they were removed from the class model. This refactoring application improved the overall design size in terms of number of classes and removed unnecessary speculative generality as shown in Table 35 and Table 36.

(b) Use Case Diagram

Table 37 Comparison of Use Case Diagram-level Metrics for ATM System

Metrics	Before Refactoring	After Refactoring
Number of Use Cases **(NUM)**	15	15
Number of Actors **(NAM)**	2	2

Table 38 Comparison of Use Case Element-level Metrics for ATM System

Metrics		Minimum	Maximum	Total	Average
# of Use Cases which this Extends **(EXTENDING)**	**Before**	0	0	0	0
	After	0	0	0	0
# of Use Cases which Extend this Use Case **(EXTENDED)**	**Before**	0	0	0	0
	After	0	0	0	0
# of Use Cases which this Includes **(INCLUDING)**	**Before**	0	0	0	0
	After	0	0	0	0
# of Use Cases which Includes this Use Case **(INCLUDED)**	**Before**	0	0	0	0
	After	0	0	0	0
# of Extension Points of The Use Case **(ExtPts)**	**Before**	0	0	0	0
	After	0	0	0	0
Depth of <<Include>> Relationship **(DOIR)**	**Before**	0	0	0	0
	After	0	0	0	0
Depth of <<Extend>> Relationship **(DOER)**	**Before**	0	0	0	0
	After	0	0	0	0

Discussion: No instances of any use case model smell were found for the ATM case study. Hence, the values remain unchanged before and after the application of refactoring as shown in Table 37 and Table 38.

(c) Sequence Diagram

Table 39 Comparison of Sequence Element-level Metrics for ATM System

Metrics		Minimum	Maximum	Total	Average
# of Lifelines **(LIFELINES)**	**Before**	2	5	57	3.80
	After	2	5	57	3.80
# of Messages **(NMM)**	**Before**	3	25	102	6.80
	After	3	25	102	6.80
# of Messages sent by the Instantiated Objects of a Class **(NMSC)**	**Before**	0	21	97	1.65
	After	0	21	96	1.63
# of Messages received by the Instantiated Objects of a Class **(NMRC)**	**Before**	0	11	104	2.00
	After	0	11	103	1.98

Discussion: A single instance of the middle man smell was found in one of the sequence models of ATM. Refactoring was performed to remove the middle man lifeline which barely affected the total number of messages sent and received in the overall sequence model of the system.

7.3.7 SCM (Supply Chain Management)

(a) Class Diagram

Table 40 Comparison of Class Diagram-level Metrics for SCM System

Metrics	Before Refactoring	After Refactoring
Number of The Classes **(NCM)**	21	21
Number of The Associations **(NASM)**	23	23
Number of The Aggregations **(NAGM)**	4	4
Number of The Inheritance Relations **(NIM)**	2	2

Table 41 Comparison of Class Element-level Metrics for SCM System

Metrics		Minimum	Maximum	Total	Average
Depth of Inheritance **(DIT)**	**Before**	0	1	2	0.10
	After	0	1	2	0.10
Number of Children **(NOC)**	**Before**	0	2	2	0.10
	After	0	2	2	0.10
Fan-In	**Before**	0	5	50	2.38
	After	0	5	50	2.38
Fan-out	**Before**	0	5	50	2.38
	After	0	5	50	2.38
# of Associations Linked to a Class **(NASC)**	**Before**	1	5	54	2.57
	After	1	5	54	2.57
# of Attributes in a Class Unweighted **(NATC1)**	**Before**	0	7	39	1.86
	After	0	7	39	1.86
# of Attributes in a Class Weighted **(NATC2)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
# of Operations in a Class Unweighted **(NOPC1)**	**Before**	0	3	23	1.10
	After	0	3	23	1.10
# of Operations in a Class Weighted **(NOPC2)**	**Before**	0	3	23	1.10
	After	0	3	23	1.10
# of Super Classes of a Class **(NSUPC)**	**Before**	0	1	2	0.10
	After	0	1	2	0.10
# of Elements in the Transitive Closure of the Super Classes of a Class **(NSUPC*)**	**Before**	0	1	2	0.10
	After	0	1	2	0.10

Discussion: Although a single instance of the Functional Decomposition was found in the class diagram of SCM case study, it was not removed. The main reason was that the class had multiple bi-directional and compositional associations with other classes. The detection of this smell was coincidental as the class was named in this manner. Hence, no refactoring operations were applied to the class diagram.

(b) Use Case Diagram

Table 42 Comparison of Use Case Diagram-level Metrics for SCM System

Metrics	Before Refactoring	After Refactoring
Number of Use Cases **(NUM)**	8	8
Number of Actors **(NAM)**	5	5

Table 43 Comparison of Use Case Element-level Metrics for SCM System

Metrics		Minimum	Maximum	Total	Average
# of Use Cases which this Extends **(EXTENDING)**	**Before**	0	0	0	0
	After	0	0	0	0
# of Use Cases which Extend this Use Case **(EXTENDED)**	**Before**	0	0	0	0
	After	0	0	0	0
# of Use Cases which this Includes **(INCLUDING)**	**Before**	0	1	2	0.25
	After	0	1	2	0.25
# of Use Cases which Includes this Use Case **(INCLUDED)**	**Before**	0	1	2	0.25
	After	0	1	2	0.25
# of Extension Points of The Use Case **(ExtPts)**	**Before**	0	0	0	0
	After	0	0	0	0
Depth of <<Include>> Relationship **(DOIR)**	**Before**	0	1	2	0.25
	After	0	1	2	0.25
Depth of <<Extend>> Relationship **(DOER)**	**Before**	0	0	0	0
	After	0	0	0	0

Discussion: No instances of any use case model smell were found for the SCM case study. Hence, the values remain unchanged before and after the application of refactoring as evident from Table 42 and Table 43.

(c) Sequence Diagram

Table 44 Comparison of Sequence Element-level Metrics for SCM System

Metrics		Minimum	Maximum	Total	Average
# of Lifelines **(LIFELINES)**	**Before**	3	5	29	3.63
	After	3	5	29	3.63
# of Messages **(NMM)**	**Before**	3	11	43	5.38
	After	3	11	43	5.38
# of Messages sent by the Instantiated Objects of a Class **(NMSC)**	**Before**	1	7	42	1.64
	After	1	7	42	1.64
# of Messages received by the Instantiated Objects of a Class **(NMRC)**	**Before**	1	4	43	1.51
	After	1	4	43	1.51

Discussion: No instances of any sequence model smell were found for the SCM case study. Hence, the values remain unchanged before and after the application of refactoring as shown by the sequence model metrics in Table 44.

7.3.8 O-Comm (OS Commerce)

(a) Class Diagram

Table 45 Comparison of Class Diagram-level Metrics for O-Comm System

Metrics	Before Refactoring	After Refactoring
Number of The Classes **(NCM)**	59	57
Number of The Associations **(NASM)**	43	41
Number of The Aggregations **(NAGM)**	2	2
Number of The Inheritance Relations **(NIM)**	26	26

Table 46 Comparison of Class Element-level Metrics for O-Comm System

Metrics		Minimum	Maximum	Total	Average
Depth of Inheritance **(DIT)**	**Before**	0	2	31	0.53
	After	0	2	31	0.54
Number of Children **(NOC)**	**Before**	0	7	26	0.44
	After	0	7	26	0.46
Fan-In	**Before**	0	9	88	1.49
	After	0	9	84	1.47
Fan-out	**Before**	0	9	88	1.49
	After	0	9	84	1.49
# of Associations Linked to a Class **(NASC)**	**Before**	0	9	90	1.53
	After	0	9	86	1.51
# of Attributes in a Class Unweighted **(NATC1)**	**Before**	0	15	211	3.58
	After	0	17	211	3.70
# of Attributes in a Class Weighted **(NATC2)**	**Before**	0	7.5	104.5	1.77
	After	0	8.5	104.5	1.83
# of Operations in a Class Unweighted **(NOPC1)**	**Before**	0	15	244	4.14
	After	0	15	202	3.54
# of Operations in a Class Weighted **(NOPC2)**	**Before**	0	15	244	4.14
	After	0	15	202	3.54
# of Super Classes of a Class **(NSUPC)**	**Before**	0	2	26	0.44
	After	0	2	26	0.46
# of Elements in the Transitive Closure of the Super Classes of a Class **(NSUPC*)**	**Before**	0	3	34	0.58
	After	0	3	34	0.60

Discussion: Two instances of the Functional Decomposition Model Smell were found and four instances of Duplication were found in the class diagram of O-Comm case study. The functionally decomposed classes were merged into their source class which although reduced the total number of classes in the model as shown in Table 45 but also increased the maximum number of attributes in the class as shown in Table 46. The duplicated methods were moved from all the subclasses to their super class. This refactoring application improved the overall design size in terms of number of operations in a class.

(b) Use Case Diagram

Table 47 Comparison of Use Case Diagram-level Metrics for O-Comm System

Metrics	Before Refactoring	After Refactoring
Number of Use Cases **(NUM)**	119	119
Number of Actors **(NAM)**	5	5

Table 48 Comparison of Use Case Element-level Metrics for O-Comm System

Metrics		Minimum	Maximum	Total	Average
# of Use Cases which this Extends **(EXTENDING)**	**Before**	0	0	0	0
	After	0	0	0	0
# of Use Cases which Extend this Use Case **(EXTENDED)**	**Before**	0	0	0	0
	After	0	0	0	0
# of Use Cases which this Includes **(INCLUDING)**	**Before**	0	0	0	0
	After	0	0	0	0
# of Use Cases which Includes this Use Case **(INCLUDED)**	**Before**	0	0	0	0
	After	0	0	0	0
# of Extension Points of The Use Case **(ExtPts)**	**Before**	0	0	0	0
	After	0	0	0	0
Depth of <<Include>> Relationship **(DOIR)**	**Before**	0	0	0	0
	After	0	0	0	0
Depth of <<Extend>> Relationship **(DOER)**	**Before**	0	0	0	0
	After	0	0	0	0

Discussion: No instances of any use case model smell were found for the O-Comm case study. Hence, the values remain unchanged before and after the application of refactoring.

(c) Sequence Diagram

Table 49 Comparison of Sequence Element-level Metrics for O-Comm System

Metrics		Minimum	Maximum	Total	Average
# of Lifelines **(LIFELINES)**	**Before**	1	8	396	3.33
	After	1	8	396	3.33
# of Messages **(NMM)**	**Before**	1	35	675	5.67
	After	1	35	675	5.67
# of Messages sent by the Instantiated Objects of a Class **(NMSC)**	**Before**	1	44	675	5.67
	After	1	44	675	5.67
# of Messages received by the Instantiated Objects of a Class **(NMRC)**	**Before**	1	37	675	5.67
	After	1	37	675	5.67

Discussion: Although quite a few instances of the middle man smell were found in the sequence models of O-Comm, but refactoring was not performed due to the use of MVC pattern in their application. Hence, the metric values shown in Table 49 remain unchanged.

7.3.9 ORA (On-Road Assistance)

(a) Class Diagram

Table 50 Comparison of Class Diagram-level Metrics for ORA System

Metrics	Before Refactoring	After Refactoring
Number of The Classes **(NCM)**	14	14
Number of The Associations **(NASM)**	16	16
Number of The Aggregations **(NAGM)**	0	0
Number of The Inheritance Relations **(NIM)**	0	0

Table 51 Comparison of Class Element-level Metrics for ORA System

Metrics		Minimum	Maximum	Total	Average
Depth of Inheritance **(DIT)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
Number of Children **(NOC)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
Fan-In	**Before**	0	4	16	1.14
	After	0	4	16	1.14
Fan-out	**Before**	0	4	16	1.14
	After	0	4	16	1.14
# of Associations Linked to a Class **(NASC)**	**Before**	1	5	32	2.29
	After	1	5	32	2.29
# of Attributes in a Class Unweighted **(NATC1)**	**Before**	1	4	33	2.36
	After	1	4	33	2.36
# of Attributes in a Class Weighted **(NATC2)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
# of Operations in a Class Unweighted **(NOPC1)**	**Before**	1	6	29	2.07
	After	1	6	29	2.07
# of Operations in a Class Weighted **(NOPC2)**	**Before**	1	6	29	2.07
	After	1	6	29	2.07
# of Super Classes of a Class **(NSUPC)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
# of Elements in the Transitive Closure of the Super Classes of a Class **(NSUPC*)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00

Discussion: No instances of any class model smell were found for the ORA case study. Hence, the values remain unchanged before and after the application of refactoring.

(b) Use Case Diagram

Table 52 Comparison of Use Case Diagram-level Metrics for ORA System

Metrics	Before Refactoring	After Refactoring
Number of Use Cases **(NUM)**	13	13
Number of Actors **(NAM)**	4	4

Table 53 Comparison of Use Case Element-level Metrics for ORA System

Metrics		Minimum	Maximum	Total	Average
# of Use Cases which this Extends **(EXTENDING)**	**Before**	0	3	4	0.31
	After	0	3	4	0.31
# of Use Cases which Extend this Use Case **(EXTENDED)**	**Before**	0	1	4	0.31
	After	0	1	4	0.31
# of Use Cases which this Includes **(INCLUDING)**	**Before**	0	6	7	0.54
	After	0	6	7	0.54
# of Use Cases which Includes this Use Case **(INCLUDED)**	**Before**	0	1	7	0.54
	After	0	1	7	0.54
# of Extension Points of The Use Case **(ExtPts)**	**Before**	0	3	4	0.31
	After	0	3	4	0.31
Depth of <<Include>> Relationship **(DOIR)**	**Before**	0	2	8	0.62
	After	0	2	8	0.62
Depth of <<Extend>> Relationship **(DOER)**	**Before**	0	1	4	0.31
	After	0	1	4	0.31

Discussion: No instances of any use case model smell were found for the ORA case study. Hence, the values remain unchanged before and after the application of refactoring.

(c) Sequence Diagram

Table 54 Comparison of Sequence Element-level Metrics for ORA System

Metrics		Minimum	Maximum	Total	Average
# of Lifelines **(LIFELINES)**	**Before**	3	8	49	3.77
	After	2	8	45	3.46
# of Messages **(NMM)**	**Before**	0	6	50	3.85
	After	0	4	42	3.23
# of Messages sent by the Instantiated Objects of a Class **(NMSC)**	**Before**	1	2	47	1.38
	After	1	2	39	1.13
# of Messages received by the Instantiated Objects of a Class **(NMRC)**	**Before**	1	2	50	1.37
	After	1	2	42	1.12

Discussion: Multiple instances of the middle man smell were found in the sequence models of SCM Case Study. A few of these were not refactored or removed as the case study used the Orchestrator design pattern. Other four sequence diagrams simply used middle man classes for delegation and hence were refactored. This in turn reduced the total number of lifelines and the messages exchanged within the sequence models as shown in Table 54.

7.4 Discussion

7.4.1 Identification of Model Smells in Use Case Diagrams

As observed from the results presented above, none of the case studies indicated the existence of models smells over use case diagrams. The main reason behind this is the fact that models smells for use case diagram are defined for the inappropriate use of the

generalization relationship. Generalization relationship in use case diagrams is the least used relationship. This is the reason why no instances of the use case model smells were identified among any of the included case studies.

7.4.2 Identification of Model Smells in Sequence Diagrams

Sequence diagram is the least used artifact for model-driven refactoring as evident from the literature. Hence, only one smell ("Middle Man") is proposed for the sequence diagram. Since the first set of case studies was taken from student projects developed by senior software engineering students, they were motivated to use common design patterns such as Façade and Model-View-Controller (MVC). Since these design patterns make use of middle man lifelines in the sequence diagram to delegate messages from the interface to the entity classes through the controller class, removing them compromises the stability of the design. Hence, the middle man refactoring was not applied whenever the existence of design patterns were detected. Only the ATM (1 instance) and the ORA (4 instances) case study exhibited a few instances of middle-man lifelines use and were refactored. Figure 90 depicts the number of case studies that use design patterns that promote the use of middle man for message delegation.

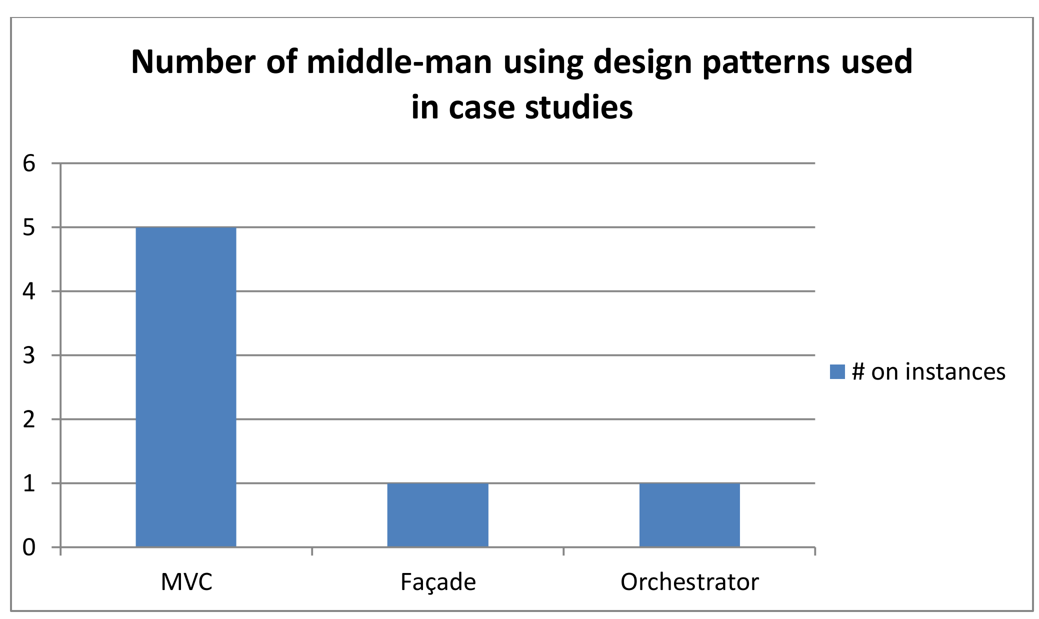


Figure 90 Number of middle-man using design patterns used in case studies

7.4.3 Identification of Model Smells in Class Diagrams

Quite a few instances of model smells proposed over the class diagram were found in the case studies. Figure 91 depicts a distribution of the number of instances of models smells found over the class models of the case studies.

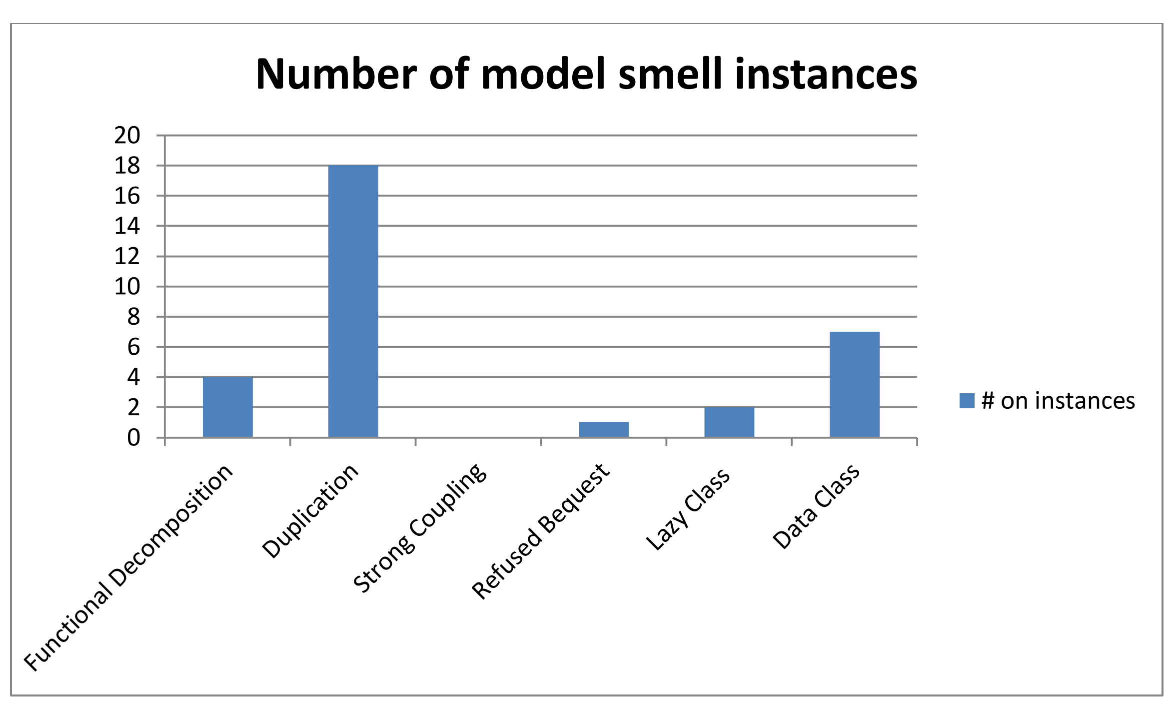


Figure 91 Number of instances of model smells detected over UML Class Diagrams

CHAPTER 8

ANALYSIS AND DISCUSSION

As part of the validation strategy discussed in the previous chapter, baseline experiments were conducted over the selected case studies. Each model in these case studies was individually refactored. In this chapter, integrated refactoring operations are applied over the model composed of the class, sequence and use case models. After the detection and application of refactoring, the integrated models are decomposed and evaluated for comparison with the results of individual refactoring. A detailed analysis and discussion on positive and negative effects of integrated refactoring over individual refactoring is included in Section 8.2.

8.1 Integrated Refactoring

8.1.1 OFD (Online Form Designer)

Two instances of the "Undue Familiarity" Model Smell, single instance of the "Spider's Web" Model Smell and seven instances of the "Duplication" Model Smell were detected within the integrated model of the OFD case study.

(a) Class Diagram

Table 55 Comparison of Class Diagram-level Metrics for OFD System

Metrics	Single-View	Multi-View
Number of The Classes **(NCM)**	21	23
Number of The Associations **(NASM)**	10	10
Number of The Aggregations **(NAGM)**	6	6
Number of The Inheritance Relations **(NIM)**	10	12

Table 56 Comparison of Class Element-level Metrics for OFD System

Metrics		Minimum	Maximum	Total	Average
Depth of Inheritance **(DIT)**	**Single**	0	2	15	0.71
	Multi	0	2	17	0.74
Number of Children (**NOC**)	**Single**	0	4	10	0.48
	Multi	0	4	12	0.52
Fan-In	**Single**	0	3	18	0.86
	Multi	0	3	18	0.78
Fan-out	**Single**	0	3	18	0.86
	Multi	0	3	18	0.78
# of Associations Linked to a Class **(NASC)**	**Single**	0	5	32	1.52
	Multi	0	5	32	1.39
# of Attributes in a Class Unweighted **(NATC1)**	**Single**	0	10	58	2.76
	Multi	0	10	57	2.48
# of Attributes in a Class Weighted **(NATC2)**	**Single**	0	3	11.5	0.55
	Multi	0	3	12	0.52
# of Operations in a Class Unweighted **(NOPC1)**	**Single**	0	29	127	6.05
	Multi	0	25	109	4.74
# of Operations in a Class Weighted **(NOPC2)**	**Single**	0	29	127	6.05
	Multi	0	25	109	4.74
# of Super Classes of a Class **(NSUPC)**	**Single**	0	1	10	0.48
	Multi	0	1	12	0.52
# of Elements in the Transitive Closure of the Super Classes of a Class **(NSUPC*)**	**Single**	0	2	15	0.71
	Multi	0	2	17	0.74

Discussion: As a result of the Duplication Model Smell, two super classes were created within the class model. Similarities between the operations of these classes were detected from the analysis of the sequence models. Although this refactoring resulted in increasing the number of classes and inheritance relations in the class model (from Table 55), it reduced the overall design modularity by reducing the maximum and total number of operations within a class as evident from Table 56.

(b) Use Case Diagram

Table 57 Comparison of Use Case Diagram-level Metrics for OFD System

Metrics	Single View	Multi View
Number of Use Cases **(NUM)**	22	21
Number of Actors **(NAM)**	2	5

Table 58 Comparison of Use Case Element-level Metrics for OFD System

Metrics		Minimum	Maximum	Total	Average
# of Use Cases which this Extends **(EXTENDING)**	**Single**	0	7	14	0.64
	Multi	0	7	7	0.33
# of Use Cases which Extend this Use Case **(EXTENDED)**	**Single**	0	2	14	0.64
	Multi	0	1	7	0.33
# of Use Cases which this Includes **(INCLUDING)**	**Single**	0	0	0	0.00
	Multi	0	0	0	0.00
# of Use Cases which Includes this Use Case **(INCLUDED)**	**Single**	0	0	0	0.00
	Multi	0	0	0	0.00
# of Extension Points of The Use Case **(ExtPts)**	**Single**	0	7	14	0.64
	Multi	0	7	7	0.33
Depth of <<Include>> Relationship **(DOIR)**	**Single**	0	0	0	0.00
	Multi	0	0	0	0.00
Depth of <<Extend>> Relationship **(DOER)**	**Single**	0	1	7	0.32
	Multi	0	1	7	0.33

Discussion: The detection and resolution of the Spider's Web and Duplication Model Smell changed the complete landscape of the use case model of the OFD case study. Resolution of the Duplication model smell reduced the maximum number of extensions in the model by half which is considerable improvement in terms of use case complexity (See Table 58). Apart from that, the maximum number of use cases per actor metric was also reduced from 14 to 5 that although resulted in increasing the total number of actors in the system as shown in Table 57.

(c) Sequence Diagram

Table 59 Comparison of Sequence Element-level Metrics for OFD System

Metrics		Minimum	Maximum	Total	Average
# of Lifelines **(LIFELINES)**	**Single**	3	5	92	4.18
	Multi	3	5	87	4.14
# of Messages **(NMM)**	**Single**	5	15	169	7.68
	Multi	5	15	156	7.43
# of Messages sent by the Instantiated Objects of a Class **(NMSC)**	**Single**	1	6	169	1.59
	Multi	1	6	156	1.60
# of Messages received by the Instantiated Objects of a Class **(NMRC)**	**Single**	1	8	169	1.52
	Multi	1	8	156	1.61

Discussion: No much changed in the sequence diagram models other than decreasing the number of sequence diagrams as a result of the Duplication Model Smell. This mainly because of the instances of integrated refactoring smells detected all dealt with the sequence diagram as whole rather than its internal functionality except the undue familiarity model smell which did not affect the collected metrics.

8.1.2 OG (OurGoal)

Eight instances of the "Undue Familiarity" Model Smell and seven instances of the "Duplication" Model Smell were detected within the integrated model of the OG case study.

(a) Class Diagram

Table 60 Comparison of Class Diagram-level Metrics for OG System

Metrics	Single View	Multi View
Number of The Classes **(NCM)**	15	15
Number of The Associations **(NASM)**	14	14
Number of The Aggregations **(NAGM)**	10	10
Number of The Inheritance Relations **(NIM)**	4	4

Table 61 Comparison of Class Element-level Metrics for OG System

Metrics		Minimum	Maximum	Total	Average
Depth of Inheritance **(DIT)**	**Single**	0	1	4	0.27
	Multi	0	1	4	0.27
Number of Children (**NOC**)	**Single**	0	2	4	0.27
	Multi	0	2	4	0.27
Fan-In	**Single**	0	12	27	1.80
	Multi	0	12	27	1.80
Fan-out	**Single**	0	12	27	1.80
	Multi	0	12	27	1.80
# of Associations Linked to a Class **(NASC)**	**Single**	0	12	48	3.20
	Multi	0	12	48	3.20
# of Attributes in a Class Unweighted **(NATC1)**	**Single**	0	19	81	5.40
	Multi	0	19	81	5.40
# of Attributes in a Class Weighted **(NATC2)**	**Single**	0	0	0	0.00
	Multi	0	0	0	0.00
# of Operations in a Class Unweighted **(NOPC1)**	**Single**	0	12	42	2.80
	Multi	0	12	42	2.80
# of Operations in a Class Weighted **(NOPC2)**	**Single**	0	12	42	2.80
	Multi	0	12	42	2.80
# of Super Classes of a Class **(NSUPC)**	**Single**	0	1	4	0.27
	Multi	0	1	4	0.27
# of Elements in the Transitive Closure of the Super Classes of a Class **(NSUPC*)**	**Single**	0	1	4	0.27
	Multi	0	1	4	0.27

Discussion: When it comes to the effect of the Undue Familiarity Model Smell from the Integrated Model Smell suite and the Data Class Model Smell from the Individual Refactoring Model Smell, the effect on class diagram is minimal. Hence, there is no apparent difference in the metric values depicted in Table 60 and Table 61.

(b) Use Case Diagram

Table 62 Comparison of Use Case Diagram-level Metrics for OG System

Metrics	Single View	Multi View
Number of Use Cases **(NUM)**	35	22
Number of Actors **(NAM)**	7	5

Table 63 Comparison of Use Case Element-level Metrics for OG System

Metrics		Minimum	Maximum	Total	Average
# of Use Cases which this Extends **(EXTENDING)**	**Single**	0	0	0	0
	Multi	0	0	0	0
# of Use Cases which Extend this Use Case **(EXTENDED)**	**Single**	0	0	0	0
	Multi	0	0	0	0
# of Use Cases which this Includes **(INCLUDING)**	**Single**	0	1	3	0.09
	Multi	0	1	3	0.14
# of Use Cases which Includes this Use Case **(INCLUDED)**	**Single**	0	2	3	0.09
	Multi	0	2	3	0.14
# of Extension Points of The Use Case **(ExtPts)**	**Single**	0	0	0	0
	Multi	0	0	0	0
Depth of <<Include>> Relationship **(DOIR)**	**Single**	0	1	2	0.06
	Multi	0	1	2	0.09
Depth of <<Extend>> Relationship **(DOER)**	**Single**	0	0	0	0
	Multi	0	0	0	0

Discussion: The resolution of the duplication model smell had a huge impact structure-wise to the use case model of the OG case study as shown in Table 62. This is a result of identifying duplicate paths in the use case model and resolving it through their respective sequence models. Not only did this refactoring reduce the number of use cases, actors which no longer were associated with use cases were also merged.

(c) Sequence Diagram

Table 64 Comparison of Sequence Element-level Metrics for OG System

Metrics		Minimum	Maximum	Total	Average
# of Lifelines **(LIFELINES)**	**Single**	4	6	175	5.00
	Multi	3	5	87	3.95
# of Messages **(NMM)**	**Single**	3	13	333	9.51
	Multi	3	13	201	9.14
# of Messages sent by the Instantiated Objects of a Class **(NMSC)**	**Single**	0	5	271	1.53
	Multi	1	5	178	1.45
# of Messages received by the Instantiated Objects of a Class **(NMRC)**	**Single**	1	5	269	1.51
	Multi	1	5	188	1.45

Discussion: As a result of the resolution of the undue familiarity model smell, the minimum and maximum number of lifelines within the sequence diagrams also reduced. And as a resolution of the duplication model smell, the total number of messages and messages exchanged between lifelines also reduced considerably as shown in Table 64. Being at the center of the integrated model, the sequence diagram had the maximum effect as a result of detection and resolution of the identified integrated model smells.

8.1.3 ESAP (Electronic Student Academic Portfolio)

Two instances of the “Specters” Model Smell and one instance of the “Duplication” Model Smell were detected within the integrated model of the ESAP case study.

(a) Class Diagram

Table 65 Comparison of Class Diagram-level Metrics for ESAP System

Metrics	Single View	Multi View
Number of The Classes **(NCM)**	28	26
Number of The Associations **(NASM)**	37	35
Number of The Aggregations **(NAGM)**	6	6
Number of The Inheritance Relations **(NIM)**	7	7

Table 66 Comparison of Class Element-level Metrics for ESAP System

Metrics		Minimum	Maximum	Total	Average
Depth of Inheritance **(DIT)**	**Single**	0	3	12	0.43
	Multi	0	3	12	0.46
Number of Children **(NOC)**	**Single**	0	3	7	0.25
	Multi	0	3	7	0.27
Fan-In	**Single**	1	8	78	2.79
	Multi	1	8	74	2.85
Fan-out	**Single**	1	8	78	2.79
	Multi	1	8	74	2.85
# of Associations Linked to a Class **(NASC)**	**Single**	1	8	85	3.04
	Multi	1	8	81	3.12
# of Attributes in a Class Unweighted **(NATC1)**	**Single**	0	10	78	2.79
	Multi	0	10	78	3.00
# of Attributes in a Class Weighted **(NATC2)**	**Single**	0	1	1	0.04
	Multi	0	1	1	0.04
# of Operations in a Class Unweighted **(NOPC1)**	**Single**	0	53	240	8.57
	Multi	0	63	251	9.65
# of Operations in a Class Weighted **(NOPC2)**	**Single**	0	53	240	8.57
	Multi	0	63	251	9.65
# of Super Classes of a Class **(NSUPC)**	**Single**	0	1	7	0.25
	Multi	0	1	7	0.27
# of Elements in the Transitive Closure of the Super Classes of a Class **(NSUPC*)**	**Single**	0	3	12	0.43
	Multi	0	3	12	0.46

Discussion: Since the Specters' model smell target temporary classes within the class model identified from the sequence models, a significant impact can be seen from the improved coupling and design size of the class model as evident from Table 65 and Table 66.

(b) Use Case Diagram

Table 67 Comparison of Use Case Diagram-level Metrics for ESAP System

Metrics	Single View	Multi View
Number of Use Cases **(NUM)**	39	38
Number of Actors **(NAM)**	6	5

Table 68 Comparison of Use Case Element-level Metrics for ESAP System

Metrics		Minimum	Maximum	Total	Average
# of Use Cases which this Extends **(EXTENDING)**	**Single**	0	5	27	0.69
	Multi	0	5	27	0.69
# of Use Cases which Extend this Use Case **(EXTENDED)**	**Single**	0	2	27	0.69
	Multi	0	2	27	0.71
# of Use Cases which this Includes **(INCLUDING)**	**Single**	0	0	0	0.00
	Multi	0	0	0	0.00
# of Use Cases which Includes this Use Case **(INCLUDED)**	**Single**	0	0	0	0.00
	Multi	0	0	0	0.00
# of Extension Points of The Use Case **(ExtPts)**	**Single**	0	5	27	0.69
	Multi	0	5	27	0.71
Depth of <<Include>> Relationship **(DOIR)**	**Single**	0	0	0	0.00
	Multi	0	0	0	0.00
Depth of <<Extend>> Relationship **(DOER)**	**Single**	0	3	37	0.95
	Multi	0	3	37	0.97

Discussion: Due to the detection of a single instance of the duplication model smell, the effect is barely noticeable with the reduction in the number of use cases and actors in the use case model of the ESAP case study.

(c) Sequence Diagram

Table 69 Comparison of Sequence Element-level Metrics for ESAP System

Metrics		Minimum	Maximum	Total	Average
# of Lifelines **(LIFELINES)**	**Single**	0	6	155	3.97
	Multi	0	6	150	3.95
# of Messages **(NMM)**	**Single**	0	20	296	7.59
	Multi	0	20	288	7.58
# of Messages sent by the Instantiated Objects of a Class **(NMSC)**	**Single**	1	9	292	1.87
	Multi	1	9	282	2.01
# of Messages received by the Instantiated Objects of a Class **(NMRC)**	**Single**	1	9	293	1.83
	Multi	1	9	283	1.94

Discussion: A combination of the specters' and duplication model smell resolution reduced the number of lifelines, messages and messages exchanged in the sequence models of the ESAP case study as shown in Table 69.

8.1.4 ME (MyEvents)

Five instances of the "Duplication" Model Smell, single instance of the "Specters" Model Smell and a single instance of the "Spider's Web" Model Smell were detected within the integrated model of the ME case study.

(a) Class Diagram

Table 70 Comparison of Class Diagram-level Metrics for ME System

Metrics	Single View	Multi View
Number of The Classes **(NCM)**	15	14
Number of The Associations **(NASM)**	12	12
Number of The Aggregations **(NAGM)**	14	13
Number of The Inheritance Relations **(NIM)**	5	5

Table 71 Comparison of Class Element-level Metrics for ME System

Metrics		Minimum	Maximum	Total	Average
Depth of Inheritance **(DIT)**	**Single**	0	2	6	0.40
	Multi	0	2	6	0.40
Number of Children **(NOC)**	**Single**	0	2	5	0.33
	Multi	0	2	5	0.33
Fan-In	**Single**	0	6	24	1.60
	Multi	0	6	23	1.53
Fan-out	**Single**	0	6	24	1.60
	Multi	0	6	23	1.53
# of Associations Linked to a Class **(NASC)**	**Single**	1	9	50	3.33
	Multi	1	9	49	3.27
# of Attributes in a Class Unweighted **(NATC1)**	**Single**	1	16	101	6.73
	Multi	1	15	90	6.00
# of Attributes in a Class Weighted **(NATC2)**	**Single**	0	2	3	0.20
	Multi	0	2	3	0.20
# of Operations in a Class Unweighted **(NOPC1)**	**Single**	1	18	103	6.87
	Multi	0	17	95	6.33
# of Operations in a Class Weighted **(NOPC2)**	**Single**	1	18	103	6.87
	Multi	0	17	95	6.33
# of Super Classes of a Class **(NSUPC)**	**Single**	0	1	5	0.33
	Multi	0	1	5	0.33
# of Elements in the Transitive Closure of the Super Classes of a Class **(NSUPC*)**	**Single**	0	2	6	0.40
	Multi	0	2	6	0.40

Discussion: Due to the detection of a single instance of the specters' model smell, the effect is barely noticeable with the reduction in the number of classes and associations in the class model of the ME case study.

(b) Use Case Diagram

Table 72 Comparison of Use Case Diagram-level Metrics for ME System

Metrics	Single View	Multi View
Number of Use Cases **(NUM)**	62	61
Number of Actors **(NAM)**	9	12

Table 73 Comparison of Use Case Element-level Metrics for ME System

Metrics		Minimum	Maximum	Total	Average
# of Use Cases which this Extends **(EXTENDING)**	**Single**	0	0	0	0.00
	Multi	0	2	4	0.07
# of Use Cases which Extend this Use Case **(EXTENDED)**	**Single**	0	0	0	0.00
	Multi	0	2	4	0.07
# of Use Cases which this Includes **(INCLUDING)**	**Single**	0	1	2	0.03
	Multi	0	1	22	0.36
# of Use Cases which Includes this Use Case **(INCLUDED)**	**Single**	0	1	2	0.03
	Multi	0	21	22	0.36
# of Extension Points of The Use Case **(ExtPts)**	**Single**	0	0	0	0.00
	Multi	0	2	4	0.07
Depth of <<Include>> Relationship **(DOIR)**	**Single**	0	1	2	0.03
	Multi	0	1	2	0.03
Depth of <<Extend>> Relationship **(DOER)**	**Single**	0	0	0	0.00
	Multi	0	1	3	0.05

Discussion: Although the ME case study had a huge number of use cases within the model, they were justified except that the maximum number of use cases per actor was 27 which resulted in the spider's web model smell. Resolution of this model smell resulted in increasing the total number of actors in the model but reducing the maximum number of use cases per actor to 12. The resolution of the duplication model smell although did not reduce the number of use cases within the system, but significantly improved the complexity and structure of the use case model by identifying and adding include and extend relationships within the model as shown in Table 73.

(c) Sequence Diagram

Table 74 Comparison of Sequence Element-level Metrics for ME System

Metrics		Minimum	Maximum	Total	Average
# of Lifelines **(LIFELINES)**	**Single**	2	6	260	4.19
	Multi	2	6	264	4.26
# of Messages **(NMM)**	**Single**	2	25	703	11.34
	Multi	2	13	424	6.84
# of Messages sent by the Instantiated Objects of a Class **(NMSC)**	**Single**	1	10	678	1.80
	Multi	0	10	455	1.60
# of Messages received by the Instantiated Objects of a Class **(NMRC)**	**Single**	1	10	674	1.59
	Multi	0	10	471	1.51

Discussion: As a result of the resolution of the duplication model smell, a huge impact was seen in terms of the maximum and total number of messages in the system as evident from the metric values in Table 74. This in turn also affected the number of messages sent and received by lifelines within the sequence models.

8.1.5 FOMS (Freelancing Online Management System)

No instances of the Integrated Model Smells were detected within the integrated model of the FOMS case study.

(a) Class Diagram

Table 75 Comparison of Class Diagram-level Metrics for FOMS System

Metrics	Single View	Multi View
Number of The Classes **(NCM)**	16	16
Number of The Associations **(NASM)**	8	8
Number of The Aggregations **(NAGM)**	10	10
Number of The Inheritance Relations **(NIM)**	7	7

Table 76 Comparison of Class Element-level Metrics for FOMS System

Metrics		Minimum	Maximum	Total	Average
Depth of Inheritance **(DIT)**	**Single**	0	2	9	0.56
	Multi	0	2	9	0.56
Number of Children **(NOC)**	**Single**	0	5	7	0.44
	Multi	0	5	7	0.44
Fan-In	**Single**	0	6	21	1.31
	Multi	0	6	21	1.31
Fan-out	**Single**	0	6	21	1.31
	Multi	0	6	21	1.31
# of Associations Linked to a Class **(NASC)**	**Single**	0	9	36	2.25
	Multi	0	9	36	2.25
# of Attributes in a Class Unweighted **(NATC1)**	**Single**	0	18	82	5.13
	Multi	0	16	84	5.13
# of Attributes in a Class Weighted **(NATC2)**	**Single**	0	2.5	2.5	0.16
	Multi	0	2.5	2.5	0.16
# of Operations in a Class Unweighted **(NOPC1)**	**Single**	1	39	174	10.88
	Multi	1	39	174	10.88
# of Operations in a Class Weighted **(NOPC2)**	**Single**	1	37	156	9.75
	Multi	1	37	156	9.75
# of Super Classes of a Class **(NSUPC)**	**Single**	0	1	7	0.44
	Multi	0	1	7	0.44
# of Elements in the Transitive Closure of the Super Classes of a Class **(NSUPC*)**	**Single**	0	2	9	0.56
	Multi	0	2	9	0.56

Discussion: No instances of any integrated model smells were found for the FOMS case study. Hence, the values remain unchanged before and after the application of refactoring as shown in Table 75 and Table 76.

(b) Use Case Diagram

Table 77 Comparison of Use Case Diagram-level Metrics for FOMS System

Metrics	Single View	Multi View
Number of Use Cases **(NUM)**	35	35
Number of Actors **(NAM)**	5	5

Table 78 Comparison of Use Case Element-level Metrics for FOMS System

Metrics		Minimum	Maximum	Total	Average
# of Use Cases which this Extends **(EXTENDING)**	**Single**	0	1	1	0.03
	Multi	0	1	1	0.03
# of Use Cases which Extend this Use Case **(EXTENDED)**	**Single**	0	1	1	0.03
	Multi	0	1	1	0.03
# of Use Cases which this Includes **(INCLUDING)**	**Single**	0	1	4	0.11
	Multi	0	1	4	0.11
# of Use Cases which Includes this Use Case **(INCLUDED)**	**Single**	0	3	4	0.11
	Multi	0	3	4	0.11
# of Extension Points of The Use Case **(ExtPts)**	**Single**	0	1	1	0.03
	Multi	0	1	1	0.03
Depth of <<Include>> Relationship **(DOIR)**	**Single**	0	1	2	0.06
	Multi	0	1	2	0.06
Depth of <<Extend>> Relationship **(DOER)**	**Single**	0	1	1	0.03
	Multi	0	1	1	0.03

Discussion: No instances of any integrated model smells were found for the FOMS case study. Hence, the values remain unchanged before and after the application of refactoring as shown in Table 77 and Table 78.

(c) Sequence Diagram

Table 79 Comparison of Sequence Element-level Metrics for FOMS System

Metrics		Minimum	Maximum	Total	Average
# of Lifelines **(LIFELINES)**	**Single**	2	8	109	3.11
	Multi	2	8	109	3.11
# of Messages **(NMM)**	**Single**	1	18	162	4.63
	Multi	1	18	162	4.63
# of Messages sent by the Instantiated Objects of a Class **(NMSC)**	**Single**	1	11	148	1.40
	Multi	1	11	148	1.40
# of Messages received by the Instantiated Objects of a Class **(NMRC)**	**Single**	1	9	154	1.62
	Multi	1	9	154	1.62

Discussion: No instances of any integrated model smells were found for the FOMS case study. Hence, the values remain unchanged before and after the application of refactoring as shown in Table 79.

8.1.6 ATM (Automated Teller Machine)

Five instances of the “Duplication” Model Smell, three instances of the “Specters” Model Smell and a single instance of the “Undue Familiarity” Model Smell were detected within the integrated model of the ATM case study.

(a) Class Diagram

Table 80 Comparison of Class Diagram-level Metrics for ATM System

Metrics	Before Refactoring	After Refactoring
Number of The Classes **(NCM)**	14	9
Number of The Associations **(NASM)**	3	3
Number of The Aggregations **(NAGM)**	8	5
Number of The Inheritance Relations **(NIM)**	2	0

Table 81 Comparison of Class Element-level Metrics for ATM System

Metrics		Minimum	Maximum	Total	Average
Depth of Inheritance **(DIT)**	**Before**	0	1	2	0.14
	After	0	0	0	0.00
Number of Children **(NOC)**	**Before**	0	2	2	0.14
	After	0	0	0	0.00
Fan-In	**Before**	0	2	11	0.79
	After	0	2	8	0.89
Fan-out	**Before**	0	2	11	0.79
	After	0	2	8	0.89
# of Associations Linked to a Class **(NASC)**	**Before**	0	8	22	1.57
	After	1	5	16	1.78
# of Attributes in a Class Unweighted **(NATC1)**	**Before**	0	14	29	2.07
	After	0	11	26	2.89
# of Attributes in a Class Weighted **(NATC2)**	**Before**	0	3	4	0.29
	After	0	1.5	2.5	0.28
# of Operations in a Class Unweighted **(NOPC1)**	**Before**	0	14	77	5.50
	After	0	16	62	6.89
# of Operations in a Class Weighted **(NOPC2)**	**Before**	0	14	77	5.50
	After	0	16	62	6.89
# of Super Classes of a Class **(NSUPC)**	**Before**	0	1	2	0.14
	After	0	0	0	0.00
# of Elements in the Transitive Closure of the Super Classes of a Class **(NSUPC*)**	**Before**	0	1	2	0.14
	After	0	0	0	0.00

Discussion: The resolution of the specters' and undue familiarity model smells reduced the number of total number of classes and aggregation relationships within the class model for the ATM case study. Based on the duplication model smell, a number of classes were identified as "speculative generality" and were possibly overriding only a single operation with no attributes. Hence, these classes were removed and additional parameter added to the message in the super class to differentiate the call based on the type of the sequence diagram. This is the reason of reduction in the number of inheritance relations in the class model.

(b) Use Case Diagram

Table 82 Comparison of Use Case Diagram-level Metrics for ATM System

Metrics	Before Refactoring	After Refactoring
Number of Use Cases **(NUM)**	15	17
Number of Actors **(NAM)**	2	2

Table 83 Comparison of Use Case Element-level Metrics for ATM System

Metrics		Minimum	Maximum	Total	Average
# of Use Cases which this Extends **(EXTENDING)**	**Before**	0	0	0	0
	After	0	4	5	0.29
# of Use Cases which Extend this Use Case **(EXTENDED)**	**Before**	0	0	0	0
	After	0	1	5	0.29
# of Use Cases which this Includes **(INCLUDING)**	**Before**	0	0	0	0
	After	0	1	7	0.41
# of Use Cases which Includes this Use Case **(INCLUDED)**	**Before**	0	0	0	0
	After	0	4	7	0.41
# of Extension Points of The Use Case **(ExtPts)**	**Before**	0	0	0	0
	After	0	4	5	0.29
Depth of <<Include>> Relationship **(DOIR)**	**Before**	0	0	0	0
	After	0	1	3	0.18
Depth of <<Extend>> Relationship **(DOER)**	**Before**	0	0	0	0
	After	0	1	5	0.29

Discussion: The removal of the duplication model smell resulted in the identification of include and extend relationships within the use case models as evident from the metrics in Table 83. Although this resulted in increasing the total number of use cases due to extraction of duplicate fragments from the sequence models of the respective use cases.

(c) Sequence Diagram

Table 84 Comparison of Sequence Element-level Metrics for ATM System

Metrics		Minimum	Maximum	Total	Average
# of Lifelines **(LIFELINES)**	**Before**	2	5	57	3.80
	After	1	5	60	3.53
# of Messages **(NMM)**	**Before**	3	25	102	6.80
	After	1	13	74	4.35
# of Messages sent by the Instantiated Objects of a Class **(NMSC)**	**Before**	0	21	96	1.63
	After	0	12	72	1.05
# of Messages received by the Instantiated Objects of a Class **(NMRC)**	**Before**	0	11	103	1.98
	After	0	5	74	1.26

Discussion: The removal of the duplication model smell resulted in significantly improving the maximum number of messages within a sequence diagram. This is mainly due to the use of extract fragment refactoring applied to remove redundant fragments into an independent sequence model.

8.1.7 SCM (Supply Chain Management)

A single instance of the "Specters" Model Smell was detected within the integrated model of the SCM case study.

(a) Class Diagram

Table 85 Comparison of Class Diagram-level Metrics for SCM System

Metrics	Before Refactoring	After Refactoring
Number of The Classes **(NCM)**	21	20
Number of The Associations **(NASM)**	23	22
Number of The Aggregations **(NAGM)**	4	4
Number of The Inheritance Relations **(NIM)**	2	2

Table 86 Comparison of Class Element-level Metrics for SCM System

Metrics		Minimum	Maximum	Total	Average
Depth of Inheritance **(DIT)**	**Before**	0	1	2	0.10
	After	0	1	2	0.10
Number of Children **(NOC)**	**Before**	0	2	2	0.10
	After	0	2	2	0.10
Fan-In	**Before**	0	5	50	2.38
	After	0	5	48	2.40
Fan-out	**Before**	0	5	50	2.38
	After	0	5	48	2.40
# of Associations Linked to a Class **(NASC)**	**Before**	1	5	54	2.57
	After	1	5	52	2.60
# of Attributes in a Class Unweighted **(NATC1)**	**Before**	0	7	39	1.86
	After	0	7	39	1.95
# of Attributes in a Class Weighted **(NATC2)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
# of Operations in a Class Unweighted **(NOPC1)**	**Before**	0	3	23	1.10
	After	0	4	25	1.25
# of Operations in a Class Weighted **(NOPC2)**	**Before**	0	3	23	1.10
	After	0	4	25	1.25
# of Super Classes of a Class **(NSUPC)**	**Before**	0	1	2	0.10
	After	0	1	2	0.10
# of Elements in the Transitive Closure of the Super Classes of a Class **(NSUPC*)**	**Before**	0	1	2	0.10
	After	0	1	2	0.10

Discussion: Due to the detection of a single instance of the specters' model smell, the effect is barely noticeable with the reduction in the number of classes and associations in the class model of the SCM case study.

(b) Use Case Diagram

Table 87 Comparison of Use Case Diagram-level Metrics for SCM System

Metrics	Before Refactoring	After Refactoring
Number of Use Cases **(NUM)**	8	8
Number of Actors **(NAM)**	5	5

Table 88 Comparison of Use Case Element-level Metrics for SCM System

Metrics		Minimum	Maximum	Total	Average
# of Use Cases which this Extends **(EXTENDING)**	**Before**	0	0	0	0
	After	0	0	0	0
# of Use Cases which Extend this Use Case **(EXTENDED)**	**Before**	0	0	0	0
	After	0	0	0	0
# of Use Cases which this Includes **(INCLUDING)**	**Before**	0	1	2	0.25
	After	0	1	2	0.25
# of Use Cases which Includes this Use Case **(INCLUDED)**	**Before**	0	1	2	0.25
	After	0	1	2	0.25
# of Extension Points of The Use Case **(ExtPts)**	**Before**	0	0	0	0
	After	0	0	0	0
Depth of <<Include>> Relationship **(DOIR)**	**Before**	0	1	2	0.25
	After	0	1	2	0.25
Depth of <<Extend>> Relationship **(DOER)**	**Before**	0	0	0	0
	After	0	0	0	0

Discussion: Since the detected instance of an integrated model smell did not affect the use case model of the SCM case study, the values remain unchanged before and after the application of refactoring as shown in Table 87 and Table 88.

(c) Sequence Diagram

Table 89 Comparison of Sequence Element-level Metrics for SCM System

Metrics		Minimum	Maximum	Total	Average
# of Lifelines **(LIFELINES)**	**Before**	3	5	29	3.63
	After	2	5	28	3.50
# of Messages **(NMM)**	**Before**	3	11	43	5.38
	After	3	11	42	5.25
# of Messages sent by the Instantiated Objects of a Class **(NMSC)**	**Before**	1	7	42	1.64
	After	1	7	41	1.65
# of Messages received by the Instantiated Objects of a Class **(NMRC)**	**Before**	1	4	43	1.51
	After	1	4	42	1.51

Discussion: Due to the detection of a single instance of the specters' model smell, the effect is barely noticeable with the reduction in the minimum number of lifelines in a sequence model of the SCM case study.

8.1.8 O-Comm (OS Commerce)

Twenty three instances of the "Duplication" Model Smell, two instances of the "Undue Familiarity" Model Smell and a single instance of the "Multiple Personality" Model Smell were detected within the integrated model of the O-Comm case study.

(a) Class Diagram

Table 90 Comparison of Class Diagram-level Metrics for O-Comm System

Metrics	Before Refactoring	After Refactoring
Number of The Classes **(NCM)**	57	57
Number of The Associations **(NASM)**	41	41
Number of The Aggregations **(NAGM)**	2	2
Number of The Inheritance Relations **(NIM)**	26	26

Table 91 Comparison of Class Element-level Metrics for O-Comm System

Metrics		Minimum	Maximum	Total	Average
Depth of Inheritance **(DIT)**	**Before**	0	2	31	0.54
	After	0	2	31	0.54
Number of Children (**NOC**)	**Before**	0	7	26	0.46
	After	0	7	26	0.46
Fan-In	**Before**	0	9	84	1.47
	After	0	9	84	1.47
Fan-out	**Before**	0	9	84	1.47
	After	0	9	84	1.47
# of Associations Linked to a Class **(NASC)**	**Before**	0	9	86	1.51
	After	0	9	86	1.51
# of Attributes in a Class Unweighted **(NATC1)**	**Before**	0	17	211	3.70
	After	0	17	211	3.70
# of Attributes in a Class Weighted **(NATC2)**	**Before**	0	8.5	104.5	1.83
	After	0	8.5	104.5	1.83
# of Operations in a Class Unweighted **(NOPC1)**	**Before**	0	15	202	3.54
	After	0	15	202	3.54
# of Operations in a Class Weighted **(NOPC2)**	**Before**	0	15	202	3.54
	After	0	15	202	3.54
# of Super Classes of a Class **(NSUPC)**	**Before**	0	2	26	0.46
	After	0	2	26	0.46
# of Elements in the Transitive Closure of the Super Classes of a Class **(NSUPC*)**	**Before**	0	3	34	0.60
	After	0	3	34	0.60

Discussion: As stated earlier, when it comes to the effect of the Undue Familiarity Model Smell from the Integrated Model Smell suite and the Data Class Model Smell from the Individual Refactoring Model Smell, the effect on class diagram is minimal. Hence, there is no apparent difference in the metric values depicted in Table 90 and Table 91.

(b) Use Case Diagram

Table 92 Comparison of Use Case Diagram-level Metrics for O-Comm System

Metrics	Before Refactoring	After Refactoring
Number of Use Cases **(NUM)**	119	80
Number of Actors **(NAM)**	5	5

Table 93 Comparison of Use Case Element-level Metrics for O-Comm System

Metrics		Minimum	Maximum	Total	Average
# of Use Cases which this Extends **(EXTENDING)**	**Before**	0	0	0	0.00
	After	0	5	5	0.06
# of Use Cases which Extend this Use Case **(EXTENDED)**	**Before**	0	0	0	0.00
	After	0	1	5	0.06
# of Use Cases which this Includes **(INCLUDING)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
# of Use Cases which Includes this Use Case **(INCLUDED)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
# of Extension Points of The Use Case **(ExtPts)**	**Before**	0	0	0	0.00
	After	0	5	5	0.06
Depth of <<Include>> Relationship **(DOIR)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
Depth of <<Extend>> Relationship **(DOER)**	**Before**	0	0	0	0.00
	After	0	1	5	0.06

Discussion: Due to the resolution of the integrated model smell instances detected, a significant improvement was seen in the complexity and structure of the use case model for the O-Comm case study. The resolution of the duplication model smell reduced the total number of use cases in the model as shown in Table 92. Identification of the Multiple personality model smell also added multiple extend relationships between use cases as evident from Table 93.

(c) Sequence Diagram

Table 94 Comparison of Sequence Element-level Metrics for O-Comm System

Metrics		Minimum	Maximum	Total	Average
# of Lifelines **(LIFELINES)**	**Before**	1	8	396	3.33
	After	1	8	274	3.43
# of Messages **(NMM)**	**Before**	1	35	675	5.67
	After	1	24	631	7.89
# of Messages sent by the Instantiated Objects of a Class **(NMSC)**	**Before**	1	44	675	5.67
	After	1	24	605	7.56
# of Messages received by the Instantiated Objects of a Class **(NMRC)**	**Before**	1	37	675	5.67
	After	1	24	627	7.84

Discussion: Although the detection and resolution of the duplication model smell improved the use case model metrics considerably, the average number of messages exchanged increased by a fair margin too in the sequence models. This is mainly because the refactoring operation to resolve the model duplication combines two sequence models with the same signature. Since the signature is made up the lifelines involved in a use cases sequence model, the refactoring did not affect the average number of lifelines per sequence model. On the positive side, the resolution of the multiple personality model

smell reduced the maximum number of messages in a sequence model significantly as evident from Table 94.

8.1.9 ORA (On-Road Assistance)

Four instances of the “Specters” Model Smell, three instances of the “Creeping Featurism” Model Smell and a single instance of the “Excessive Alternation” Model Smell were detected within the integrated model of the ORA case study.

(a) Class Diagram

Table 95 Comparison of Class Diagram-level Metrics for ORA System

Metrics	Before Refactoring	After Refactoring
Number of The Classes **(NCM)**	14	10
Number of The Associations **(NASM)**	16	12
Number of The Aggregations **(NAGM)**	0	0
Number of The Inheritance Relations **(NIM)**	0	0

Table 96 Comparison of Class Element-level Metrics for ORA System

Metrics		Minimum	Maximum	Total	Average
Depth of Inheritance **(DIT)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
Number of Children **(NOC)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
Fan-In	**Before**	0	4	16	1.14
	After	0	4	12	1.20
Fan-out	**Before**	0	4	16	1.14
	After	0	4	12	1.20
# of Associations Linked to a Class **(NASC)**	**Before**	1	5	32	2.29
	After	1	5	24	2.40
# of Attributes in a Class Unweighted **(NATC1)**	**Before**	1	4	33	2.36
	After	2	4	29	2.90
# of Attributes in a Class Weighted **(NATC2)**	**Before**	0	0	0	0.00
	After	0	0	0	0.0
# of Operations in a Class Unweighted **(NOPC1)**	**Before**	1	6	29	2.07
	After	1	6	24	2.40
# of Operations in a Class Weighted **(NOPC2)**	**Before**	1	6	29	2.07
	After	1	6	24	2.40
# of Super Classes of a Class **(NSUPC)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00
# of Elements in the Transitive Closure of the Super Classes of a Class **(NSUPC*)**	**Before**	0	0	0	0.00
	After	0	0	0	0.00

Discussion: The resolution of the specters' model smell reduced the number of total number of classes and association relationships within the class model for the ORA case study as evident from the class model metrics shown in Table 95 and Table 96.

(b) Use Case Diagram

Table 97 Comparison of Use Case Diagram-level Metrics for ORA System

Metrics	Before Refactoring	After Refactoring
Number of Use Cases **(NUM)**	13	9
Number of Actors **(NAM)**	4	4

Table 98 Comparison of Use Case Element-level Metrics for ORA System

Metrics		Minimum	Maximum	Total	Average
# of Use Cases which this Extends **(EXTENDING)**	**Before**	0	3	4	0.31
	After	0	1	1	0.11
# of Use Cases which Extend this Use Case **(EXTENDED)**	**Before**	0	1	4	0.31
	After	0	1	1	0.11
# of Use Cases which this Includes **(INCLUDING)**	**Before**	0	6	7	0.54
	After	0	3	4	0.44
# of Use Cases which Includes this Use Case **(INCLUDED)**	**Before**	0	1	7	0.54
	After	0	1	4	0.44
# of Extension Points of The Use Case **(ExtPts)**	**Before**	0	3	4	0.31
	After	0	1	1	0.11
Depth of <<Include>> Relationship **(DOIR)**	**Before**	0	2	8	0.62
	After	0	2	5	0.56
Depth of <<Extend>> Relationship **(DOER)**	**Before**	0	1	4	0.31
	After	0	1	1	0.11

Discussion: As a result of the resolution of the creeping featurism model smell, a couple of functional use cases were merged into their "including" use cases. This in turn reduced the total number of use cases within the model. On the other hand, the resolution of the excessive alternation model smell reduced the number of extend relationships within the use case model. This effect of integrated refactoring on the relationships in the use case model for the ORA case study is evident from Table 98.

(c) Sequence Diagram

Table 99 Comparison of Sequence Element-level Metrics for ORA System

Metrics		Minimum	Maximum	Total	Average
# of Lifelines **(LIFELINES)**	**Before**	2	8	45	3.46
	After	2	7	31	3.44
# of Messages **(NMM)**	**Before**	0	4	42	3.23
	After	0	9	35	3.89
# of Messages sent by the Instantiated Objects of a Class **(NMSC)**	**Before**	1	2	39	1.13
	After	1	4	33	1,20
# of Messages received by the Instantiated Objects of a Class **(NMRC)**	**Before**	1	2	42	1.12
	After	1	4	34	1.20

Discussion: Although the resolution of the specters' model smell reduced the maximum and total number of lifelines per sequence model, the resolution of the creeping featurism increased the number of messages exchanged within a sequence model.

8.2 Analysis and Discussion

In this chapter, we evaluated the effect of refactoring, considering both single model at a time and a multi-view integrated model, on indicative metrics for class, sequence and use

case models. The evaluation and discussion demonstrated that the impact of refactoring on these metrics was non-trivial. Hence, it is not feasible to generalize that any application of a refactoring to remove the identified smell improves or impairs one of the external quality attributes.

Based on this information, instead of stating that refactoring a model smell has partial impact on the metrics, we found that it is beneficial if the effect on the metrics is described by an impact spectrum rather than specific values. A collection of metrics are loosely associated with a design characteristic such as size, modularity and so on. These associations are based on the work of Seidl and Sneed [458] who tried associating UML model metrics with characteristics such as quantity, complexity, quality and size. Although their work is intended for the application of testing, it can be used for our analysis as well. Each metric is then given a value from the set {+, -, =} which designates the impact as a result of refactoring.

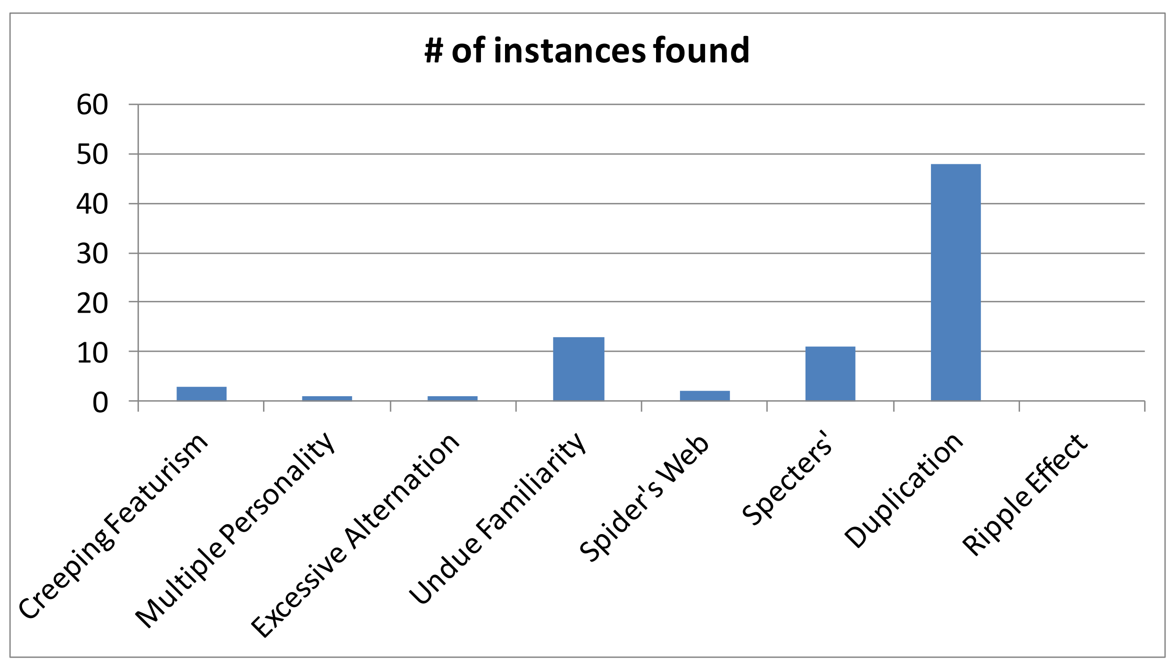


Figure 92 Number of instances of Integrated Model Smells detected

Figure 92 shows the number of instances of integrated model smells detected over the selected case studies. It is evident that the Duplication model smell is the most popular among all the other smells. The only model smells instance not detected within the existing case studies is Ripple Effect.

8.2.1 Integrated Refactoring Impact on Use Case Diagram

Depicted in Figure 93 (a) and (b) are use case metric associations with their design characteristics.

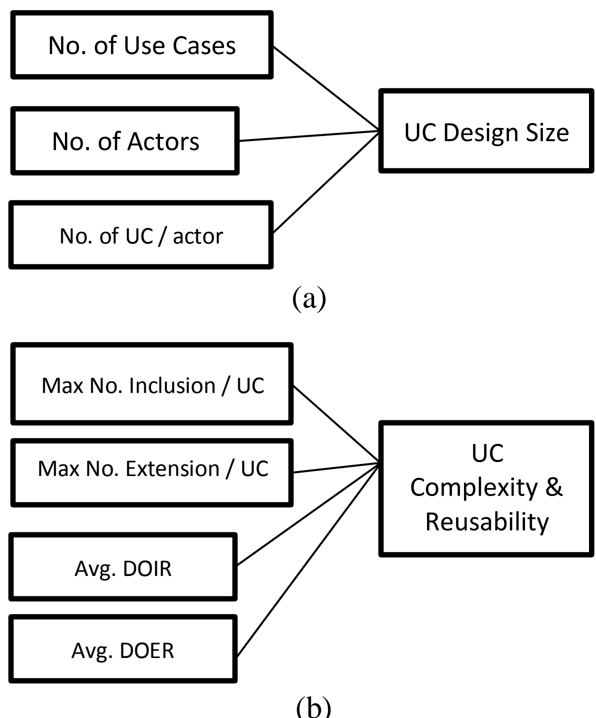


Figure 93 Use case metrics association with model characteristics

Table 100 Refactoring impact spectrum over use case design size metrics

Case Study	No. of Use Cases	No. of Actors	No. of UC/Actor
OFD	-	+	-
OG	-	-	-
ESAP	-	-	-
ME	-	+	-
FOMS	-	+	-
ATM	+	=	-
SCM	=	=	=
O-Comm	-	=	-
ORA	-	=	+

When it comes to design size, the smaller the number of elements the better it is for analysis. Table 100 shows a consistent reduction in the number of use cases and number of use cases per actor metric as a result of integrated model refactoring. Few instances where there is an increase in the number of actors is mainly due to resolution of the Spider's Web Model Smell which usually is accompanied by a significant reduction in the number of use cases associated with actors. There is only a single instance when the number of use cases per actor increases. This is because of the resolution of the Excessive Alternation Model Smell that associates extension use cases directly to the actor removing the extend relationship.

Table 101 Refactoring impact spectrum over use case complexity metrics

Case Study	Max No. of Inclusion/UC	Max No. of Extension/UC	Avg. DOIR	Avg. DOER
OFD	=	-	=	=
OG	=	=	=	=
ESAP	=	=	=	=
ME	+	+	=	+
FOMS	+	=	=	=
ATM	+	+	+	+
SCM	=	=	=	=
O-Comm	=	+	=	+
ORA	-	-	-	-

With the set of metric chosen for use case complexity and reusability, the lower the value the better it is for use case analysis. Exceptions include increase in relationships when there are actually no relationships in the original model. As seen from the results summarized in Table 101, it is evident that most of the time the impact on relationships is either the same or increased. The three case studies (ME, ATM, O-Comm) that actually resulted in increasing the values of the metrics had no relationships between use cases within the use case model. Integrated Refactoring over these case studies identified these relationships mainly by removing duplication and adding structure to the overall model. Hence, the increase in these cases is more beneficial rather than considered a side-effect.

8.2.2 Integrated Refactoring Impact on Class Diagram

For the analysis of the impact of class diagrams, we used a slightly different approach. Based on the work of Seidl and Sneed [458], we analyze the impact of integrated refactoring on the case studies on the following characteristics:

- **Data Complexity:** The more data attributes a class has the higher its complexity

- **Functional Complexity:** More methods a class have, the higher its complexity.
- **Hierarchical Complexity:** More hierarchical levels, the more dependent the lower level classes are on the higher level ones.
- **Coupling:** Classes with a high coupling have a greater impact domain.
- **Reusability:** The more, associations and interactions there are, the more difficult it is to take out individual classes and methods from the current architecture and to reuse them

Table 102 Refactoring impact spectrum over class metrics

Case Study	Data Complexity (NOA)	Functional Complexity (NOM)	Hierarchical Complexity (DIT)	Coupling (DCC)	Reusability (NASM + NAGM)
OFD	-	-	=	=	+
OG	=	=	=	=	=
ESAP	=	+	=	-	-
ME	-	-	=	-	-
FOMS	-	-	=	-	=
ATM	-	-	-	-	-
SCM	=	+	=	-	-
O-Comm	=	=	=	=	=
ORA	-	-	=	-	-

As the characteristics of the class model analyzed deal with complexity and reusability, the lower the values the better it is for class diagram use and analysis. As seen from Table 102, there are notable improvements in the degree of complexity, reusability and modularity of the class model after integrated refactoring. Although there are a few exceptions, these is mainly due to the resolution of undue familiarity wherein operations are distributed among classes resulting in increase in the NOM metric values.

8.2.3 Integrated Refactoring Impact on Sequence Diagram

Depicted in Figure 94 (a) and (b) are sequence model metric associations with their design characteristics.

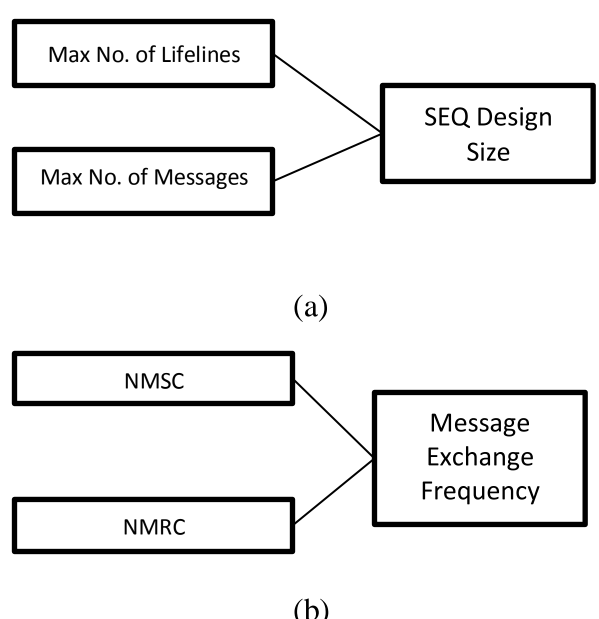


Figure 94 Sequence model metrics association with model characteristics

Table 103 Refactoring impact spectrum over sequence model design size metrics

Case Study	Max No. of Lifelines	Max No. of Messages
OFD	=	=
OG	-	=
ESAP	=	=
ME	=	-
FOMS	=	-
ATM	=	-
SCM	=	=
O-Comm	=	-
ORA	=	+

When it comes to design size, the smaller the number of elements the better it is for analysis. Table 103 shows a notable improvement to sequence diagram design size mainly in terms of the number of messages. Although there is an exception in one case study (ORA) where the max number of messages exchanged between lifelines has actually increased. This is mainly contributed due to the side-effect of the creeping featursm model smell resolution as the messages of the inclusion use case are typically combined with its base use case when the inclusion is a functional decomposition.

Table 104 Refactoring impact spectrum over sequence model message frequency

Case Study	NMSC	NMRC
OFD	-	-
OG	-	-
ESAP	-	-
ME	-	-
FOMS	-	-
ATM	-	-
SCM	=	=
O-Comm	-	-
ORA	-	-

As seen from the results summarized in Table 104, it is evident that the message exchange frequency had consistency improved as a result of integrated model refactoring.

CHAPTER 9

CONCLUSION AND FUTURE WORK

9.1 Summary

Model-driven engineering, an emerging trend in software engineering, has enabled the application of refactoring to UML models. The concept of refactoring was initially used for source code restructuring. The main goal of refactoring is to reduce software complexity by modifying the system without altering its external behavior. With the popularity of MDE and UML, recent approaches for refactoring have elevated it to a more abstract level of design models. Hence the term model refactoring or model-driven refactoring was coined.

An Object-Oriented system modeled by UML is built up from many different views. Model refactoring, in recent proposals, is applied to a single view at a given time. Hence, information from other views are either not considered or later synchronized for consistency preservation. The objective of this research was to develop a multi-view integrated approach to model-driven refactoring using UML models. Due to feasibility, we restricted our scope to one diagram from each UML view, class diagram (structural view), sequence diagram (behavioral view) and use case diagram (functional view). An integrated metamodel composed from the metamodels of the selected UML models was initially constructed. In order to ensure proper integration of metamodels, metamodels of the sequence diagram and use case diagram were initially extended prior to composition.

Refactoring opportunities and transformation operations were defined at the metamodel level (integrated), which is based on the M2 level of the UML architecture in order to utilize the extension capability of the language.

Our approach to refactor the integrated model consisted of two main steps. First, we identified where to apply refactoring by detecting refactoring opportunities identified in this work. Finally, we applied a set of composite refactorings used to remove the model smell from the integrated model. We proposed a total of eight integrated refactoring opportunities that can be used to improve the design models where these opportunities appear. For each of the proposed refactoring opportunities, we also described a set of primitive refactorings that can be used to remove the identified smells. The main objective of identifying these refactoring opportunities was to make the design models more maintainable by improving the overall organization of the software system.

We developed a tool that fully supported the integrated model refactoring approach from integration to refactoring and evaluation. We performed an empirical validation using nine case studies to explore the effectiveness of our approach. The validation study compared our integrated refactoring approach with refactoring applied to models individually in terms of quality improvement. From the results, we found that more opportunities can be detected using the integrated approach rather than the individual refactoring approach.

Quality improvement through refactoring was measured by the difference between metrics before and after the application of refactoring. As a result of the evaluation and analysis we found that the use of an integrated model aided in identification of more

design flaws than individual refactoring of models. Inter-model flaws such as duplication, specters', undue familiarity etc. were easy to detect and resolve when information from multiple views was considered. The resulting use case models depicted reusability through inclusion & extension and better responsibility assignment. The resulting sequence models depicted reduction in the total number of messages within each diagram and reduction in the message passing frequency. The resulting class models depicted modularity through reduction in coupling, distribution of behavior to their familiar classes and use of OO concepts such as inheritance and abstraction.

9.2 Contributions

The research work presented in this dissertation makes the following contributions to the field of model-driven software refactoring:

1. Provides a state-of-the–art survey and systematic literature review in the field of model-driven software refactoring.
2. Provides a process model consisting of a number of distinct activities essential for model-driven refactoring along with a comparison framework for evaluating existing refactoring approaches.
3. Provides an extension to the UML metamodel for sequence diagram by adding model elements to enable model extensibility and enhance code traceability.
4. Provides an extension to the UML metamodel for use case diagram that includes representation for all its elements and relationships (structural and textual) in a

conflict-free manner and one that includes information for model analysis, evaluation and interchange among modeling tools.

5. Provides an integrated metamodel built taking into consideration the three views of UML models: structural, behavioral and functional.
6. Provides a catalog of eight model smells based on the integrated metamodel. These smells take into consideration information from the functional, behavioral and structural view (in the form of an integrated metamodel) and propose refactoring opportunities to correct design defects and anti-patterns covering the different views of UML models.
7. Provides a prototype tool to develop use case diagrams authoring both its structural and behavioral components.
8. Provides automated tool support for model smell detection, resolution and evaluation over the proposed integrated metamodel.

9.3 Threats to Validity

Threats to validity for an empirical study are divided into three types: Construct Validity, Internal Validity and External Validity [459].

Construct Validity: This measures the extent to which the independent and dependent variables accurately model the study hypotheses. In our work, the dependent variable which is the quality improvement achieved by refactoring, has to address the degree the quality model accurately measures the quality of the software. Due to the lack of a quality model that is evaluated through empirical experiments and expert opinion in the field of

model-driven refactoring, we evaluated the effect of refactoring on indicative metrics for class, sequence and use case models. Another threat to construct validity is the choice of threshold values for a few smell detection strategies. These values were obtained from design guidelines and metric authors from the literature. The choice of a threshold value can vary the effectiveness of the strategy in identifying model smells in the integrated metamodel.

Internal Validity: This measures the extent to which changes in the dependent variable can be safely attributed to changes in the independent variables. In our validation, there are two threats pertaining to this category: unavailability of a model quality framework and semi-automatic application of refactoring. Due to lack of a mapping framework between the internal quality metrics and external attributes, comparison between the internal quality metrics was performed. Second, although identification of refactoring opportunities is performed automatically through the proposed prototype tool, a semi-automatic approach is employed for resolution. This means each refactoring operation before its application over the model is consulted from the user. There is a possibility that the we may have misclassified a few false-positive cases as opportunities for refactoring. This threat was considerably mitigated by the fact that we are well versed with the case studies and that the size of the case studies was relatively small.

External Validity: This measures the extent to which results of the study are generic and negate the effects of environmental variables. In our validation, there is only one threat pertaining to this category: choice of case studies for evaluation. Case studies considered in this work were obtained from senior software engineering projects and small-sized case studies published in the literature and books. These case studies may not be

representative of all types of systems, specifically industrial case studies developed by professionals and practitioners. Due to the lack of usable large sized case studies in the domain of this research, the behavior of the integrated model refactoring approach could not be assessed on a wider scale.

9.4 Future Works

Refactoring software, especially models of software is a relatively new discipline and a highly active area of research. The work developed in this research considered application of refactoring over multiple views of UML in an integrated manner, which is a novel achievement in this constantly evolving area. Hence, the approach must go through several adjustments based on substantial experience of practical applications to obtain relevance in the industry. Based on our review of literature in the field of model-driven refactoring and the work presented in this dissertation, several possible directions for future investigations were identified.

Formal systems add preciseness to the process of refactoring at the expense of interoperability and ease of use. On the other hand, text based approaches (like XMI) are easy to understand and are portable but makes the task of model refactoring difficult due to size, manual handling of transformation and behavior preservation and impreciseness. Techniques to integrate formality within text-based approaches will improve usability of these approaches.

There is a significant gap between the model smells and anti-patterns proposed for source code and models. An initial attempt to bridge this gap has been proposed in this dissertation that considers more than one UML view to identify model smells. Identification of more refactoring opportunities based on the integrated model is hence required. Pattern-based model refactoring is another refactoring opportunity detection approach gaining immense popularity. The use of the integrated metamodel identifying the application of behavioral and structural design patterns can also be investigated.

Apart from the UML class diagram, other diagrams are rarely used for refactoring. The use of multiple UML models for detection of smells may motivate the researchers to look into refactoring operations for other UML models. Defining refactoring opportunities including other models in the integrated framework, namely the state and object diagrams, will allow addition of more information to the structural and behavioral view.

Research in the area of model quality evaluation is significantly lacking behind. Hence, there is a vital requirement of a model metrics catalog for all UML models, framework to establish correlation between these metrics and external model quality attributes and empirical studies to evaluate the effect of model metrics and design patterns over model refactoring techniques.

Other avenues for future work include investigation of interaction information from models for behavior specification and preservation (call preservation). An plugin version of the Integrated Refactoring tool for popular CASE tools such as Eclipse can be developed for wider use. Finally, there is need for approaches to determine an appropriate model smell application and resolution schedule is required to maximize quality

improvements. Further studies should also be performed to evaluate the effectiveness of the proposed integrated model refactoring large real-world project designs.

Appendix 1: Formal Description for the UML Metamodel

A1.1 Class Diagram

A class diagram is a 4-tuple $CD = \{\mathbb{C}, \mathbb{A}, \mathbb{R}, WF_{CD}\}$ where

- $\mathbb{C}$ is a non-empty finite set of classes
- $\mathbb{A}$ is a finite set of associations
- $\mathbb{R} \in \mathbb{C}\ x\ \mathbb{C}$ is the relationship between classes
- $WF_{CD}$is a set of well-formedness rules on the Class Diagram CD

In this subsection, a detailed description of the abstract syntax of UML class diagrams is initially provided followed by a list of formalized well-formedness rules.

- **[CLASS]** A class $\mathrm{C} \in \mathbb{C}$ consists of the following components:
  - $name\ (\mathbf{C}) \in$ Name where Name is the name space of a class diagram.
  - $upper\ (\mathbf{C})$ is an optional integer specifying the upper multiplicity.
  - $lower\ (\mathbf{C})$ is an optional integer specifying the lower multiplicity.
  - $isAbstract\ (\mathbf{C})$ specifies that the class does not provide a complete declaration.
  - $isRoot\ (\mathbf{C})$ is a Boolean that specifies whether the class has ancestors or not.
  - $isLeaf\ (\mathbf{C})$ is a Boolean that specifies whether the class has descendents or not.
- **[ATTRIBUTE]** A class is composed of a set of attributes and operations. An attribute $\boldsymbol{Attr(C)}$ of a class is represented by instances of Property and consists of the following components:
  - $visibility$ (**Attr**) $\in \{ public, private, protected\}$.

- $name$ (**Attr**).
- $type$ (**Attr**) which may be one of the basic types or other classes.
- $upper$ (**Attr**) is an optional integer specifying the upper multiplicity.
- $lower$ (**Attr**) is an optional integer specifying the lower multiplicity.
- $default$ (**Attr**) which is an initial value of the attribute of type $type$ (**Attr**).
- $isReadOnly$ (**Attr**) is a Boolean that specifies whether the attribute is fixed (true) or changeable.
- $isDervied$ (**Attr**) is a Boolean that specifies whether the attribute is derived from other attributes or not.

The default syntax of an attribute declaration given in the UML specification is:

$$[< visibility >][< multiplicity >] < name >:< type > [<= default - value >] \{property\ string\}$$

- **[OPERATION]** An operation $\mathrm{Op}(\mathbb{C})$ of a class is a function that can be performed to alter the behavior of a class. It consists of the following components
    - $visibility$ (**Op**) $\in \{public, private, protected\}$.
    - $name$ (**Op**).
    - $ret - type$ (**Op**) is an optional return type which may be one of the basic types or other classes.
    - is$Query$ (**Op**) is a Boolean that specifies whether its execution changes that system state or not.
    - $isAbstract$ (**Op**) is a Boolean that specifies whether the details of the operation are provided or by a descendent.

- $isUnique$ (**Op**) is a Boolean that specifies whether the return parameter is unique or not.

- **[PARAMETER]** An operation is composed of a list of zero or more formal parameters Param (Op). Each parameter has the following components
  - $name$ (**Param**).
  - $directionKind$ (**Param**) $\in \{in, out, inout, return\}$.
  - $type$ (**Param**) which may be one of the basic types or other classes.
  - $default$ (**Param**) which is an initial value of the parameter of type type(Param).

The default syntax of an operation is given as

$$[< visibility >] < name > [< Parameter\ List >]: [< return - type >]\{property\ string\}$$

and each parameter in the $< Parameter\ List >$ is described as

$$[< directionKind >] < parameter - name >: < parameter - type > [= < default - value >]$$

Classes in a class diagram are related to each other by different types of relationships. Relationships in a UML class diagram are classified into three categories: *Association, Generalization* and *Dependency*.

- **[ASSOCIATION]** An association $\mathbb{A}$ consists of an association name and a set of association ends End ($\mathbb{A}$). An association end $\{e : e \in End(\mathbb{A})\}$ consists of the following components:
  - $class$ (**e**) is the class connected to the end.

- $roleName(\mathbf{e})$ which can be used to traverse from the source end to the target end.
- $lower\ (\mathbf{e})$ is an integer that specifies the lower bound on the number of target instances that can be associated with a source instance.
- $upper\ (\mathbf{e})$ is an integer that specifies the upper bound on the number of target instances that can be associated with a source instance.
- $aggregationKind\ (\mathbf{e})$ specifies whether the end is an aggregation with respect to another end. $\text{aggregationKind}(\text{e}) \in \{\text{none}, \text{shared}, \text{composite}\}$.
- $navigable\ (\mathbf{e})$ is a Boolean that specifies whether traversing from source to the target instances is possible or not.

- **[GENERALIZATION]** A generalization $\mathbb{G} \in \mathbb{R}$ is a directed relationship between two classes. It consists of:
  - $super\ (\mathbb{G}) \in \mathbb{C}$is the super class.
  - $sub\ (\mathbb{G}) \in \mathbb{C}$ is the sub class.
  - $isSubstitutable\ (\mathbb{G})$ is a Boolean.
- **[ASSOCIATION CLASS]** In the UML Metamodel, an Association Class is a declaration between classes, which has a set of attributes of its own. Association Class is both an Association and a Class. An association class $\mathbb{C}_{\text{assoc}}$ consists of the following component:
  - $name\ (\mathbb{C}_{\text{assoc}}) \in$ Name where Name is the name space of a class diagram.

Apart from the abstract syntax, the UML specification also provides a set of well-formedness rules. Well-formedness rules for class diagrams written in a formal description can be found in [460]. These set of well-formedness rules (WF) for the UML class diagram are written here in a formal notation. In this subsection, a detailed description of the well-formedness rules of UML class diagrams are provided.

- $\mathrm{WF_{CD}}$ **Rule 1:** A well-formed class has unique attribute names

$$\forall\, att_1, att_2 : Attribute\,.$$
$$(att_1 \in \boldsymbol{Attr}(\mathbb{C}) \wedge att_2 \in \boldsymbol{Attr}(\mathbb{C}) \wedge att_1 \neq att_2) \Rightarrow name(att_1) \neq name(att_2)$$

- $\mathrm{WF_{CD}}$ **Rule 2:** Operations can have same names if they differ in scope, types or number of parameters or result type

$$\begin{aligned}&\forall\, op_1, op_2 : Operation\,.\\ &\quad (op_1 \in \boldsymbol{Op}(\mathbb{C}) \wedge op_2 \in \boldsymbol{Op}(\mathbb{C}) \wedge op_1 \neq op_2 \wedge name(op_1) = name(op_2))\\ &\qquad \Rightarrow \big(visibility(op_1)\\ &\qquad \neq visibility(op_2) \vee diff_param\big(Param(op_1), Param(op_2)\big)\big)\end{aligned}$$

In this rule diff_param is an auxiliary function that checks whether the parameters (also known as message signature) of the operations are different. This function can be formally written as:

$$\begin{aligned}&diff_param(Param(op_1), Param(op_2)) \equiv\\ &\quad length\big(Param(op_1)\big) = length\big(Param(op_1)\big)\\ &\qquad \Longrightarrow \big(\exists\, i : i \leq length\big(Param(op_1)\big) \wedge type\big(Param(op_1)\big)\\ &\qquad \neq type\big(Param(op_2)\big)\big)\end{aligned}$$

- WF_{CD} **Rule 3:** A class with an abstract operation must be abstract

$$\exists\, op : Operation\,.$$
$$(op \in \boldsymbol{Op}(\mathbb{C}) \wedge isAbstract(op)) \Longrightarrow isAbstract(\mathbb{C})$$

- WF_{CD} **Rule 4:** An abstract class must have at least one abstract operation

$$isAbstract(\mathbb{C}) \Longrightarrow \big(\exists\, op : op \in \boldsymbol{Op}(\mathbb{C}) \wedge isAbstract(op)\big)$$

- WF_{CD} **Rule 5:** Multiplicity of the class must be valid

$$lower(\mathbb{C}) \leq upper(\mathbb{C})$$

- WF_{CD} **Rule 6:** An abstract class must be inherited by another concrete class

$$isAbstract(\mathbb{C}) \Longrightarrow \sim isLeaf(\mathbb{C})$$

- WF_{CD} **Rule 7:** An operation can have at most one return parameter

$$\forall\, op : Operation\,.$$
$$(op \in \boldsymbol{Op}(\mathbb{C}) \Longrightarrow length\big(directionKind\big(Param(op)\big) = return\big) \leq 1$$

- WF_{CD} **Rule 8:** An association is n-ary when $n \geq 2$

$$cardinality\big(End(\mathbb{A})\big) \geq 2$$

- WF_{CD} **Rule 9:** Multiplicities of association ends must be well-formed

$$\forall\, e : Association\,End\,.$$
$$\big(e \in End(\mathbb{A}) \Longrightarrow lower(e) \leq upper(e)\big)$$

- $\mathrm{WF_{CD}}$ **Rule 10:** An association end with one end as "shared" or "composite" aggregation-kind must be a binary association

$$\forall\, e : Association\ End\ .$$
$$((e \in End(\mathbb{A}) \wedge (aggregationKind(e) = composite \vee aggregationKind(e) = shared)) \Longrightarrow cardinality(End(\mathbb{A})) = 2)$$

- $\mathrm{WF_{CD}}$ **Rule 11:** An association end with one end as "composite" aggregation-kind must be navigable

$$\forall\, e_1, e_2 : Association\ End\ .$$
$$\big((e_1 \in End(\mathbb{A}) \wedge aggregationKind(e_1) = composite \wedge e_2 \in End(\mathbb{A}) \wedge e_2 \neq e_1) \Longrightarrow navigable(e_2)\big)$$

- $\mathrm{WF_{CD}}$ **Rule 12:** Only one end of an association can be "shared" or "composite"

$$\forall\, e : Association\ End\ .$$
$$\Big(\big(e \in End(\mathbb{A}) \wedge (aggregationKind(e) = composite \vee aggregationKind(e) = shared)\big) \Longrightarrow \sim (\exists\, e_1 : (e_1 \in End(\mathbb{A}) \wedge e_1 \neq e \wedge (aggregationKind(e_1) = composite \vee aggregationKind(e_1) = shared))\Big)$$

- WF_{CD} **Rule 13:** An association end with one end as “composite” aggregation-kind, that end cannot have multiplicity greater than 1

$$\forall\, e : Association\ End\ .$$
$$\big((\ e\ \in End(\mathbb{A}) \wedge aggregationKind(e) = composite\ ) \Longrightarrow upper(e) \leq 1\big)$$

- WF_{CD} **Rule 14:** In an association, at least one end must be navigable

$$\forall\, e : Association\ End\ .$$
$$\big(\ e\ \in End(\mathbb{A}) \Longrightarrow navigable(e)\big)$$

- WF_{CD} **Rule 15:** In a generalization relationship, the subclass cannot be a root

$$\forall\, g_1 : Generalization\ .$$
$$\Big(g_1 \in \mathbb{R} \Longrightarrow \sim isRoot\big(sub(g_1)\big)\Big)$$

- WF_{CD} **Rule 16:** In a generalization relationship, the super class cannot be a leaf

$$\forall\, g_1 : Generalization\ .$$
$$\Big(g_1 \in \mathbb{R} \Longrightarrow \sim isLeaf\big(super(g_1)\big)\Big)$$

- WF_{CD} **Rule 17:** In a generalization relationship, the subclass cannot redefine the attributes of the super class

$$\forall\, g_1 : Generalization\ .$$
$$g_1 \in \mathbb{R} \wedge \Big(\forall\, att_1, att_2 : \Big(att_1 \in \boldsymbol{Attr}\big(super(g_1)\big) \wedge\ att_2 \in \boldsymbol{Attr}\big(sub(g_1)\big)\Big)$$
$$\Rightarrow name(att_1) \neq name(att_2)\Big)$$

- $\mathrm{WF_{CD}}$ **Rule 18:** Each class in the class diagram has a unique name

$$\forall\, c_1, c_2 : Class\,.$$
$$((c_1 \in \mathbb{C} \land c_2 \in \mathbb{C} \land c_1 \neq c_2) \Longrightarrow name(c_1) \neq name(c_2))$$

- $\mathrm{WF_{CD}}$ **Rule 19:** Two different associations relating to a common class cannot have the same name

$$\forall\, a_1, a_2 : Association\,.$$
$$((a_1 \in \mathbb{A} \land a_2 \in \mathbb{A} \land a_1 \neq a_2 \land name(a_1) = name(a_2)) \Longrightarrow End(a_1) \cap End(a_2) = \{\}$$

- $\mathrm{WF_{CD}}$ **Rule 20:** An abstract class in the class diagram must be the super class of at least one concrete class

$$\forall c : Class\,.$$
$$(c \in \mathbb{C} \land isAbstract(c)) \Longrightarrow (\exists\, g : Generalization\,.\; super(g) = c)$$

- $\mathrm{WF_{CD}}$ **Rule 21:** There should be no loops among generalizations in a class diagram

$$\forall\, uc_1 : Use\ Case\,.$$
$$uc_1 \in \mathbb{U} \Longrightarrow \sim\!\left(uc_1 \in allIncluded(uc_1)\right)$$

In this rule $\mathrm{allIncluded(uc)}$ is an auxiliary function that returns the transitive closure of all the use cases included by this use case directly or indirectly. This function can be formally written as

$$allIncluded : \mathbb{U} \rightarrow \mathbb{U} - \boldsymbol{set}$$

$$allIncluded(uc) \equiv \{(uc_1) \mid (uc_1) : UseCase\ .$$

$$(\exists\ inc_1 : Inclusion\ .$$

$$inc_1 \in \mathbb{r} \wedge (including(\ inc_1) = \ uc) \wedge$$

$$addition(\ inc_1) \in uc_1\ \wedge allincluded(addition(\ inc_1)))\}$$

A1.2 Sequence Diagram

A sequence diagram is a 7-tuple $SEQ = \{\mathbb{L}, End, Mes, \mathbb{E}, \leq, fragment, WF_{SEQ}\}$ where

- $\mathbb{L}$ is a finite set of lifelines
- End is a finite set of end locations
- Mes is a finite set of message labels
- $\mathbb{E} \subseteq End\ \times\ Mes\ \times\ End$ is the relationship (event) between lifelines
- $\leq \subseteq End\ \times End$ is a partial order providing the position of ends within each of the lifelines
- $fragment$ is an ordered set of fragments in the sequence diagram
- WF_{SEQ} is a set of well-formedness rules on the Sequence Diagram SEQ

Similar to that of the Class diagram, the UML Specification document also describes the Sequence Diagram metamodel by an abstract syntax in the form of a class diagram and the well-formedness rules. In this subsection, a detailed description of the abstract syntax of UML sequence diagrams will be provided.

- **[LIFELINE]** A lifeline l $\in \mathbb{L}$ consists of the following components
    - $name$ (**l**).
    - $endList$ (**l**) $\subseteq$ End is a set of all end locations part of the lifeline whose ordering is provided by using the " $\leq$ " relational operator.
    - $attributes$ (**l**) is a set of attributes that belongs to a lifeline.
    - $decomposedAs$ (**l**) is the name of the decomposed fragment that shows the interactions for the decomposed lifeline.
- **[DECOMPOSITION]** A decomposed fragment of a lifeline l is given by an external sequence diagram SEQ_l.
- **[END LOCATION]** An end location end $\in$ End consists of the following components:
    - $name$ (**end**).
    - $lifeline$ (**end**) $\in \mathbb{L}$ is the lifeline to which this end belongs to.
    - $isGate$ (**end**) is a Boolean that specifies whether the end is a gate or not.
    - $formalGate$ (**end**) $\in$ fragment is the fragment to which end belongs if the end is a gate.
- **[MESSAGE]** A message M $\in \mathbb{E}$consists of the following components:
    - $name$ (**M**) $\in$ Mes.
    - $messageKind$ (**M**) $\in$ $\{ syncCall, asyncCall, createMsg, deleteMsg, return \}$.
    - $messageSort$ (**M**) $\in \{ complete, lost, found, unknown \}$.
    - $retAttr$ (**M**) is an optional attribute to which the return value is assigned.
    - $retVal$ (**M**) is the return value of the message.

- $sendEnd$ (**M**) ∈ End specifies the sending end of a message.
- $recieveEnd$ (**M**) ∈ End specifies the receiving end of a message.

- **[ARGUMENTS]** A message is composed of a list of zero or more argumentsArg (M). Each argument has the following components:
  - $name$ (**Arg**).
  - val (**Arg**) is a value assigned to the argument or '-' if not assigned.

The default syntax of a message is given as
$[< retAttr >'= '] < name > ['('[< Argument\ List >]')'][':' < retVal >]$
and each argument in the $< Argument\ List >$ is described as
$([< name >'='] < val >) \mid ' - '$

Fragments fragment in a sequence diagram are classified into three categories: *Combined Fragments, Interaction Use Fragments* and *State Invariants.*

- **[COMBINED FRAGMENT]** A combined fragment cf ∈ fragment consists of the following components:
  - $covered$(**cf**) ⊆ $\mathbb{L}$ is a set of lifelines covered by the fragment.
  - $operator$ (**cf**) ∈ $\{opt, loop, break, neg, par, alt, assert, strict, seq, consider, ignore\}$.
  - $operand$ (**cf**) is a set of operands of the combined fragment.
  - $fragmentgate$ (**cf**) ⊆ End is a set of gates between the fragment and its enclosing interaction.

- **[OPERAND]** An operand $\mathrm{opd} \in \mathrm{operand(cf)}$ consists of an interaction constraint $\mathrm{constraint(opd)}$ and an operand body. An operand body is given by an inline sequence diagram $\mathrm{SEQ_{opd}}$.
  - **[CONSTRAINT]** $\mathrm{constraint(opd)}$ is an interaction constraint given as a Boolean expression which guards the entry into an operand. It includes the following components:
    - $minint\ (\mathbf{constraint})$ is an optional value or an expression that specifies the minimum number of iterations.
    - $maxint\ (\mathbf{constraint})$ is an optional value or an expression that specifies the maximum number of iterations.

The default syntax of an interaction constraint is given by $('['(< boolean\ expression > \mid '\boldsymbol{else}' \mid)']'\ )$

- **[INTERACTION USE]** An interaction use $\mathrm{ref} \in \mathrm{fragment}$ is given by the same default syntax as that of a message but the name in this case refers to the referred interaction. The referred interaction is an external sequence diagram $\mathrm{SEQ_{ref}}$. An interaction use fragment also consists of $\mathrm{actualgate\ (ref)} \subseteq \mathrm{End}$ is a set of gates between the fragment and its enclosing interaction.
- **[STATE INVARIANT]** A state invariant $\mathrm{inv} \in \mathrm{fragment}$ consists of the following components
  - $covered\ (\mathbf{inv}) \in \mathbb{L}$ is the lifeline covered by the state invariant.
  - $condition\ (\mathbf{inv})$ is the constraint that should hold at runtime.

Also in this subsection, a detailed description of the well-formedness rules of UML sequence diagrams is provided.

- WF_{SEQ} **Rule 1:** If in a sequence diagram a lifeline is decomposed, the sequence of constructs in the diagram such as combined fragments and interaction use covering this lifeline must also appear in the decomposed interaction. This is also known as extra-global.

$$\forall\, l_1 : Lifeline\,.$$
$$l_1 \in \mathbb{L} \wedge decomposedAs(l_1) \neq \emptyset \implies partOf(l_1) = fragment(SEQ_{l_1})$$

In this rule partOf(l)is an auxiliary function that returns all the fragments that the lifeline l is part of. This function can be formally written as

$$partOf : \mathbb{L} \rightarrow fragment - \boldsymbol{set}$$
$$partOf(l) \equiv$$
$$\{f \mid f : fragment\,.$$
$$(\exists\, cf_1 : fragment\,.$$
$$l \in covered(\,cf_1) \wedge\ (f) =\ (cf_1))\}$$

- WF_{SEQ} **Rule 2:** The Send event must be ordered before the receive event if both the send and the receive event belonging to a message are on the same lifeline

$$\forall\, E_1 : Message\,.$$
$$E_1 \in \mathbb{E}\ \wedge (lifeline(sendEnd(E_1)) = lifeline(recieveEnd(E_1))) \implies sendEnd(E_1) \leq recieveEnd(E_1)$$

- $\mathrm{WF_{SEQ}}$ **Rule 3:** If a return attribute is specified in a message, it must be an attribute of the lifeline sending the message

$$\forall\, E_1 : Message\,.$$
$$E_1 \in \mathbb{E} \,\wedge\, isGate\big(recieveEnd(E_1)\big) \wedge\!\Longrightarrow covered(cf_1) = \mathbb{L}$$

- $\mathrm{WF_{SEQ}}$ **Rule 4:** Arguments of a message must be attributes of the sending lifeline or constants

$$\forall\, E_1 : Message\,.$$
$$E_1 \in \mathbb{E} \Longrightarrow name\big(\boldsymbol{Arg}(E_1)\big) = attributes\Big(lifeline\big(sendEnd(E_1)\big)\Big) \vee Nat_Num$$

- $\mathrm{WF_{SEQ}}$ **Rule 5:** Messages inside of a combined fragment should not cross its boundaries or its operands within the combined fragment

$$\forall\, cf_1 : Fragment\,.$$
$$cf_1 \in Fragment \,\wedge (\exists\, opd_1,\, opd_2 : Operand\,.$$
$$opd_1 \in operand(cf_1) \wedge\, opd_2 \in operand(cf_1)) \Longrightarrow End(\,opd_1) \,\cap\, End(\,opd_2) = \{\,\}$$

- $\mathrm{WF_{SEQ}}$ **Rule 6:** A combined fragment with operator opt, loop, break or neg must have exactly one operand

$$\forall\, cf_1 : Fragment\,.$$
$$\begin{aligned} cf_1 \in fragment & \\ & \wedge \big((operator(cf_1) = opt) \vee (operator(cf_1) = loop) \\ & \vee (operator(cf_1) = break) \vee (operator(cf_1) = neg)\big) \\ & \Longrightarrow size\big(operand(cf_1)\big) = 1 \end{aligned}$$

- WF_{SEQ} **Rule 7:** The interaction constraint with minint and maxint applies only to a combined fragment with operator loop

$$\forall\, cf_1 : Fragment\,.$$
$$cf_1 \in fragment\ \wedge \sim (operator(cf_1) = loop)$$
$$\Rightarrow \left(minint\left(constraint\big(operand(cf_1)\big)\right) = \emptyset\right)$$
$$\wedge\ \left(maxint\left(constraint\big(operand(cf_1)\big)\right) = \emptyset\right)$$

- WF_{SEQ} **Rule 8:** A combined fragment with operator break should cover all the lifelines within the enclosing sequence diagram

$$\forall\, cf_1 : Fragment\,.$$
$$cf_1 \in fragment\ \wedge\ (operator(cf_1) = break) \Rightarrow covered(cf_1) = \mathbb{L}$$

- WF_{SEQ} **Rule 9:** A combined fragment with operator loop and minint interaction constraint specified then the evaluation of minint should be a non-negative integer

$$\forall\, cf_1 : Fragment\,.$$
$$cf_1 \in fragment\ \wedge\ operator(cf_1)$$
$$= loop \wedge \left(minint\left(constraint\big(operand(cf_1)\big)\right) \neq \emptyset\right)$$
$$\Rightarrow \left(evaluate\Big(minint\Big(constraint\big(operand(cf_1)\big)\Big)\Big) \geq 0\right)$$

- $\mathrm{WF}_{\mathrm{SEQ}}$ **Rule 10:** A combined fragment with operator loop and maxint interaction constraint specified then the evaluation of maxint should be a positive integer

$$\forall\, cf_1 : Fragment\,.$$
$$cf_1 \in fragment \wedge operator(cf_1) = loop \wedge \Big(maxint\Big(constraint\big(operand(cf_1)\big)\Big) \neq \emptyset\Big) \Longrightarrow \Big(evaluate\Big(maxint\Big(constraint\big(operand(cf_1)\big)\Big)\Big) > 0\Big)$$

- $\mathbf{WF}_{\mathbf{SEQ}}$ **Rule 11:** A combined fragment with operator loop and both minint and maxint interaction constraint specified, then the evaluation of maxint should be greater than or equal to the evaluation of minint

$$\forall\, cf_1 : Fragment\,.$$
$$cf_1 \in fragment \wedge operator(cf_1) = loop \wedge \Big(minint\Big(constraint\big(operand(cf_1)\big)\Big) \neq \emptyset\Big) \wedge \Big(maxint\Big(constraint\big(operand(cf_1)\big)\Big) \neq \emptyset\Big) \Longrightarrow \Big(evaluate\Big(maxint\Big(constraint\big(operand(cf_1)\big)\Big)\Big) \geq evaluate\Big(maxint\Big(constraint\big(operand(cf_1)\big)\Big)\Big)\Big)$$

A1.3 Use Case Diagram

A use case diagram is a 5-tuple $UC = \{\mathbb{U}, \mathbb{a}, \mathbb{m}, \mathbb{r}, WF_{UC}\}$ where

- $\mathbb{U}$ is a finite set of use cases
- $\mathbb{a}$ is a finite set of actors
- $\mathbb{m} \subseteq \mathbb{a} \times \mathbb{U}$ is a finite set of associations
- $\mathbb{r} \subseteq \mathbb{U} \times \mathbb{U}$ is the relationship between use cases

- $WF_{UC}$is a set of well-formedness rules on the Use Case Diagram UC

In this subsection, a detailed description of the abstract syntax of UML use case diagrams will be provided.

- **[ACTOR]** An actor $a \in \mathbb{a}$consists of the following components
  - $name(a)$.
- **[USE CASE]** A use case $\mathrm{uc} \in \mathbb{U}$consists of the following components
  - $\mathrm{name(uc)}$.
  - $\mathrm{extPoint(uc)}$is a set of all extension points owned by the use case.
- **[EXTENSION POINT]** An extension $\mathrm{Point\ ep}$ belonging to a use case has a name $\mathrm{name(ep)}$.

The default syntax of an extension point is given by $< name > [: < explanation >]$

- **[ASSOCIATION]** An association relationship $\mathrm{m} \in \mathbb{M}$ consists of the following components:
  - $\mathrm{subject(m)} \in \mathbb{a}$is the actor.
  - $\mathrm{use(m)} \in \mathbb{U}$ is the use case.

Use cases in a use case diagram are related to each other by different types of relationships. These relationships are generalization, inclusion and extension.

- **[GENERALIZATION]** A generalization relationship $\mathrm{gen} \in \mathbb{\Gamma}$ consists of the following components:
  - $\mathrm{super(gen)} \in \mathbb{U}$is the general use case.

- sub(gen) $\in \mathbb{U}$ is the specialized use case.

- **[INCLUSION]** An inclusion relationship inc $\in \mathbb{r}$ consists of the following components:
  - addition(inc) $\in \mathbb{U}$ is the use case that is to be included.
  - including(inc) $\in \mathbb{U}$ is the use case that will include the addition.
- **[EXTENSION]** An extension relationship ext $\in \mathbb{r}$ consists of the following components:
  - extended(ext) $\in \mathbb{U}$ is the use case that is being extended (base).
  - extension(ext) $\in \mathbb{U}$ is the use case that is represents the extension.
  - condition(ext) is condition that must hold for the extension to take place.
  - extLoc(ext) is an ordered list of extension points ep where fragments of the extending use case are to be inserted.
- **[ACTOR GENERALIZATION]** An actor generalization relationship gen_a consists of the following components:
  - super(gen_a) $\in \mathbb{a}$ is the general actor.
  - sub(gen_a) $\in \mathbb{a}$ is the specialized actor.

Also in this subsection, a detailed description of the well-formedness rules of UML use case diagrams is provided.

- WF_{UC} **Rule 1:** An actor must have a name

$$\forall\, a_1 : Actor\ .$$
$$a_1 \in \mathbb{a} \implies name(a_1) \neq \emptyset$$

- WF_{UC} **Rule 2:** A use case must have a name

$$\forall\, uc_1 : Use\ Case\ .$$
$$uc_1 \in \mathbb{U} \implies name(uc_1) \neq \emptyset$$

- WF_{UC} **Rule 3:** A use case cannot include use cases that directly or indirectly include it.

$$\forall\, uc_1 : Use\ Case\ .$$
$$uc_1 \in \mathbb{U} \implies \sim\big(uc_1 \in allIncluded(uc_1)\big)$$

In this rule allIncluded(uc) is an auxiliary function that returns the transitive closure of all the use cases included by this use case directly or indirectly. This function can be formally written as

$$allIncluded : \mathbb{U} \rightarrow \mathbb{U} - \boldsymbol{set}$$
$$allIncluded(uc) \equiv$$
$$\{(uc_1) \mid$$
$$(uc_1) : UseCase\ .$$
$$(\exists\, inc_1 : Inclusion\ .$$
$$inc_1 \in \mathbb{r} \wedge (including(\, inc_1) = \ uc) \wedge$$
$$addition(\, inc_1) \in uc_1 \ \wedge allincluded(addition(\, inc_1)))\}$$

- $\mathrm{WF_{UC}}$ **Rule 4:** An extension point must have a name

$\forall\, uc_1 : Use\ Case\ .$

$$uc_1 \in \mathbb{U} \implies name(extPoint((uc_1)) \neq \emptyset$$

- $\mathrm{WF_{UC}}$ **Rule 5:** The extension locations referenced by the extend relationship must belong to the use case being extended

$\forall\, ext_1 : Extension\ .$

$$ext_1 \in \mathbb{r} \implies extLoc(ext_1) = extPoint(extended(ext_1))$$

Appendix 2: Model Refactoring Catalog

This section provides the specification of all model level refactorings. These refactorings are grouped into three categories based on the model they transform: Use Case, Class and Sequence. Each refactoring is described in detail. Refactoring pre-conditions and post-conditions are defined using notations and functions described in Appendix 1. These refactorings are provided as a Java API (library – jar). In order to invoke these refactorings, the UML model should be parsed and used as a DOM tree. The document node of that tree is passed on each invocation.

A2.1 Use Case Model Refactoring

1. Create Use Case

Description: This refactoring creates a new empty use case without any associated actors and any associated interaction.

Origin: From Rui [286] [page 134]

Parameters: String $newUC$

Preconditions: The name of the new use case ($newUC$) does not conflict with the name of an existing use case within the model.

$\forall\, uc_1 : Use\ Case\ .$

$$uc_1 \in \mathbb{U} \implies name(uc_1) \neq newUC$$

Post-conditions:

$$\exists\, uc \in \mathbb{U}' \wedge uc \notin \mathbb{U} \wedge name(uc) = newUC$$

Mechanism &Verification: The behavior of the use case model is not affected with the addition of the newly created use case. The precondition ensures preservation of distinct entity name invariant.

Implementation:

- Method Name: create_UseCase
- Arguments: Document doc, String name where
    - *doc* is the document node of the source model
    - *name* is the name for the use case
- Return Value: String *status*

2. Create Actor

Description: This refactoring creates a new actor without any reference to a use case(s).

Origin: From Rui [286] [page 135]

Parameters: String *newActor*

Preconditions: The name of the new actor ($newActor$) does not conflict with the name of an existing actor within the model.

$$\forall\, a_1 : Actor\ .$$
$$a_1 \in \mathbb{a} \Rightarrow name(a_1) \neq newActor$$

Post-conditions:

$$\exists\ a \in \mathbb{a}' \wedge a\ \notin\ \mathbb{a}\ \wedge name(\ a) = newActor$$

Mechanism &Verification: The newly created actor does not interact with any use case and is isolated from other actors. Therefore, the behavior of the use case model does not change with the addition of a new actor. The precondition ensures preservation of distinct entity name invariant.

Implementation:

- Method Name: create_Actor
- Arguments: String name where
    - *name* is the name for the actor
- Return Value: String *status*

3. Delete Use Case

Description: This refactoring deletes an unreferenced use case from the use case model.

Origin: From Rui [286] [page 137]

Parameters: Use case *uc*

Preconditions: The use case is isolated from other use cases and actors. Isolation from other use cases means

- No inclusions
- No extensions

- Not included and extended by other use cases
- Not a super use case to other use cases

$$\exists\, uc: UseCase\ .$$
$$\forall\, m: Association.$$
$$m \in \mathbb{M}\ \wedge use(m) \neq uc$$
$$\forall\, rel: Relationship.$$
$$rel \in \mathbb{r}\ \wedge\ including(rel) \neq uc\ \vee extended(rel)\ \neq uc\ \vee super\ (rel)\ \neq uc$$
$$\forall\, uc_1: Use\ Case.$$
$$uc_1 \in \mathbb{U}\ \wedge uc\ \notin allIncluded(uc_1) \wedge uc\ \notin allExtended(uc_1)$$

Post-conditions:

$$uc \notin\ \mathbb{U}'$$

Mechanism &Verification: Since the use case is isolated from other use cases and actors, it does not affect interactions between them. Hence, this deletion does not change the behavior of the use case model.

Implementation:

- Method Name: delete_UseCase
- Arguments: String name where
  - *name* is the name of the use case
- Return Value: String *status*

4. Delete Actor

Description: This refactoring deletes an unreferenced actor from the use case model.

Origin: From Rui [286] [page 138]

Parameters: Actor *a*

Preconditions: The actor is isolated from other use cases and actors. Isolation from other actors means that the actor is not a super-actor to any other actor.

$$\exists\, a: Actor\ .$$
$$\forall\, m: Association.$$
$$m \in \mathbb{M}\ \wedge\ subject(m) \neq a$$
$$\forall\, rel: Relationship.$$
$$rel \in \mathbb{g}\ \wedge\ super\ (rel)\ \neq a$$

Post-conditions:

$$a \notin\ \mathbb{a}'$$

Mechanism &Verification: Since the actor is isolated from other use cases and actors, it does not participate in interactions between them. Hence, this deletion does not change the behavior of the use case model.

Implementation:

- Method Name: delete_Actor
- Arguments: String name where
    - *name* is the name of the actor
- Return Value: String *status*

5. Generalize Use Cases

Description: This refactoring creates a generalization relationship between two or more use cases. This refactoring reduces redundancy in use cases by moving common interactions to the parent use case and hence improves reusability.

Origin: From Rui [286] [page 154]

Parameters: A set of use cases $\{uc_1, uc_2 \dots uc_n\}$, String *newUC*

Preconditions:

(i) The use cases $\{uc_1, uc_2 \dots uc_n\}$ are used by the same set of actors. In order to formally write this condition, we define an auxiliary function that returns all the actors associated with a given use case. This function can be written as

$$
\begin{aligned}
& allActors : \mathbb{U} \rightarrow \mathbb{A} - set \\
& allActors(uc) \equiv \\
& \{(a_i) | \\
& \qquad (a_i) : Actor . \\
& \qquad (\exists\, m : Association . \\
& \qquad m \in \mathbb{M} \wedge use(m) = uc \wedge \\
& \qquad subject(m) \in a_i)\}
\end{aligned}
$$

Then the precondition can be written as

$$
\begin{aligned}
& \forall uc : UseCases \in \{uc_1, uc_2 \dots uc_n\} . \\
& \qquad uc \in \mathbb{U} \wedge \forall\, uc_i, uc_j : \in \{uc_1, uc_2 \dots uc_n\} \wedge \\
& \qquad allActors(uc_i) = allActors(uc_j)
\end{aligned}
$$

(ii) There is no relationship among the use cases $\{uc_1, uc_2 \ldots uc_n\}$. These use cases are not referenced by any other use case.

$$\forall\, uc{:}\, UseCases \in \{uc_1, uc_2 \ldots uc_n\}\,. \quad uc \in \mathbb{U} \wedge allIncluded(uc) = \{\} \wedge allExtended(uc) = \{\}$$

(iii) The name of the new super use case ($newUC$) does not conflict with the name of an existing use case within the model.

$$\forall\, uc_1{:}\, Use\ Case\,. \quad uc_1 \in \mathbb{U} \Longrightarrow name(uc_1) \neq newUC$$

Post-conditions:

A new use case is created and it is the parent of use cases $\{uc_1, uc_2 \ldots uc_n\}$.

$$\exists\, uc \in \mathbb{U}' \wedge uc \notin \mathbb{U} \wedge name(uc) = newUC \wedge$$
$$\forall\, uc \in \{uc_1, uc_2 \ldots uc_n\}\ .$$
$$\exists\, rel \in \mathbb{r}' \wedge rel \notin \mathbb{r} \wedge super(rel) = newUC \wedge sub(rel) = uc$$

Mechanism &Verification: A new empty use case is created and is assigned as the parent or super use cases of the given use cases. In the behavioral view, common interaction fragment is moved to this use case.

The precondition (i) ensures that the use cases $\{uc_1, uc_2 \ldots uc_n\}$ has the same set of actors. According to the definition of generalization relationship, moving common interaction elements to the parent use case does not change the behavior of the use cases.

Precondition (ii) ensures that the use cases are isolated. Precondition (iii) ensures distinct entity name for the newly added parent use case.

Implementation:

- Method Name: generalize_UseCases
- Arguments: ArrayList *subUCNames*, String *superUCName* where
  - *subUCNames* are the names of the child use cases
  - *superUCName* is the name of the parent use case
- Return Value: String *status*

6. Generalize Actors

Description: This refactoring creates a generalization relationship between two or more actors using a common set of use cases. A new actor is created which uses the above common set of use cases.

Origin: From Rui [286] [page 157]

Parameters: A set of actors $\{a_1, a_2 \ldots a_n\}$, String *newActor*

Preconditions:

(i) The actors $\{a_1, a_2 \ldots a_n\}$ use a common set of use cases $\{uc_1, uc_2 \ldots uc_n\}$. In order to formally write this condition, we define an auxiliary function that returns all the actors associated with a given use case. This function can be written as

$$allUC: \mathbb{A} \rightarrow \mathbb{U} - set$$
$$allUC(a) \equiv$$
$$\{(uc_i) | \quad (uc_i) : UseCase \,. \quad (\exists\, m : Association \,. \quad m \in \mathbb{M} \wedge subject(m) = a \wedge use(m) \in uc_i)\}$$

Then the precondition can be written as

$$\forall\, a{:}Actor \in \{a_1, a_2 \ldots a_n\} \,. \quad a \in \mathbb{a} \wedge \forall\, a_i, a_j{:} \in \{a_1, a_2 \ldots a_n\} \wedge allUC(a_i) = allUC(a_j)$$

(ii) There is no actor relationship among actors $\{a_1, a_2 \ldots a_n\}$, and any other actor does not reference them.

$$\exists\, a_1, a_2{:}Actors \,. \quad a_1, a_2 \in \mathbb{a} \wedge a_1 \neq a_2 \wedge \nexists\, rel \in \mathbb{g}. super(rel) = a_1 \wedge sub(rel) = a_2$$

(iii) The name of the new super actor ($newActor$) does not conflict with the name of an existing actor within the model.

$$\forall\, a_1{:}Actor \,. \quad a_1 \in \mathbb{a} \Rightarrow name(a_1) \neq newActor$$

Post-conditions:

(i) A new actor is created and it is the parent of actors $\{a_1, a_2 \dots a_n\}$.

$$\exists\, a \in \mathbb{a}' \wedge a \notin \mathbb{a} \wedge name(a) = newActor \wedge$$

$$\forall\, a \in \{a_1, a_2 \dots a_n\} \;.$$

$$\exists\, rel \in \mathbb{g}' \wedge rel \notin \mathbb{g} \wedge super(rel) = newActor \wedge sub(rel) = a$$

(ii) The new actor has association relationship with use cases $\{uc_1, uc_2 \dots uc_n\}$.

$$\forall\, uc \in \{uc_1, uc_2 \dots uc_n\}.$$

$$\exists\, m \in \mathbb{M}' \wedge m \notin \mathbb{M} \wedge subject(m) = newActor \wedge use(m) = uc$$

(ii) Association relationships between use cases $\{uc_1, uc_2 \dots uc_n\}$ and actors $\{a_1, a_2 \dots a_n\}$ are removed. Actors inherit these relationships from the parent actor newActor.

$$\forall\, a \in \{a_1, a_2 \dots a_n\}.$$

$$allUC(a) \not\subset \{uc_1, uc_2 \dots uc_n\}$$

Mechanism &Verification:

A generalization relationship between actors means that the child actors participate in all relationships of the parent actor. All common use cases are associated with the new parent actor and are removed from the child actors.

No new association between actors and use cases are added. Actors $\{a_1, a_2 \dots a_n\}$ inherit association relationships between newActor and use cases $\{uc_1, uc_2 \dots uc_n\}$. Hence all interactions between actors and use cases are preserved. Precondition (ii) ensures that actors $\{a_1, a_2 \dots a_n\}$ are isolated from other actors so that the newActor does not affect

other actors. Precondition (iii) ensures distinct entity name for the newly added parent actor.

Implementation:

- Method Name: create_ActorGeneralization
- Arguments: ArrayList s*ubActorNames*, String *superActorName* where
    - *subActorNames*are the names of the child actors
    - *superActorName*is the name of the parent actor
- Return Value: String *status*

7. Merge Use Cases

Description: This refactoring merges two independent use cases that are used by the same set of actors. This refactoring helps manage the use case granularity by avoiding fragment use cases.

Origin: From Rui [286] [page 152]

Parameters: Use case $uc_1$and uc_2

Preconditions:

(i) Use cases uc_1 and uc_2 are not referenced by any use case.

$$\forall\, uc \in \{uc_1, uc_2\}.$$
$$uc \in \mathbb{U} \wedge$$
$$\forall\, rel{:}\, Relationship.$$
$$rel \in \mathbb{r} \wedge including(rel) \neq uc \vee extended(rel) \neq uc \vee super\,(rel) \neq uc$$
$$\forall\, uc_i{:}\, Use\ Case.$$

$$uc_i \in \mathbb{U} \wedge uc \notin allIncluded(uc_i) \wedge uc \notin allExtended(uc_i)$$

(ii) Use cases uc_1 and uc_2 are used by the same set of actors.

$$\forall\, uc_1, uc_2 : UseCases\ .$$
$$allActors(uc_1) = allActors(uc_2)$$

Post-conditions:

The use case uc_2 is deleted.

$$uc_2 \notin \mathbb{U}'$$

Mechanism &Verification: This refactoring keeps one use case and deletes the other one.

The precondition (i) ensures that the use cases {uc_1, uc_2} are isolated from other use cases. This ensures that merging them together does not affect the behavior of the use case model. Precondition (ii) ensures that the use cases are used by the same set of actors.

Implementation:

- Method Name: merge_UseCases
- Arguments: String *UC1,* String *UC2* where
    - *UC1* and*UC2* are the names of the use cases to be merged
- Return Value: String *status*

8. Merge Actors

Description: This refactoring merges two actors into one. This refactoring helps manage actors.

Origin: From Rui [286] [page 156]

Parameters: Actor $a_1$and a_2

Preconditions:

Actors a_1 and a_2 are not referenced by any other actor. However, actor a_2 can be the parent of actor a_1.

$$\exists\, a_1, a_2 : Actors\ .$$
$$a_1, a_2 \in 𝕒 \ \wedge\ a_1 \neq a_2 \wedge$$
$$\nexists\, rel \in\ 𝕘\,.\ super(rel) \in \{a_1\,, a_2\,\} \vee\ sub(rel) \in \{a_1\,, a_2\,\}$$
$$\exists\, rel \in\ 𝕘\,.\ super(rel) = a_2 \Longrightarrow sub(rel) = a_1$$

Post-conditions:

(i) Use case references by actor a_2 are used by the actor a_1.

$$\forall\, uc\!:\in allUC(a_2)\ .$$
$$\exists\, m \in \mathbb{M}' \wedge m\ \notin\ \mathbb{M}\ \wedge subject(\,m) = a_1 \wedge use(\,m) = uc$$

(ii) The actor a_2 is deleted.

$$a_2 \notin\ 𝕒'$$

Mechanism &Verification: This refactoring keeps one actor and deletes the other one.

The precondition ensures that the actors$\{a_1, a_2\}$ are isolated from other actors. This ensures that merging them together does not affect the behavior of the use case model.

Implementation:

- Method Name: merge_Actors
- Arguments: String *A1*, String *A2* where
    - *A1* and*A2* are the names of the actors to be merged
- Return Value: String *status*

9. Merge Use Case Generalization

Description: This refactoring merges two use cases that are related to each other by generalization and the interaction of the parent use case is empty (abstract). This refactoring helps maintain the abstraction level of use cases.

Origin: From Rui [286] [page 146]

Parameters: Use Case uc_1 and its parent uc_2

Preconditions:

(i) There is a generalization relationship between use cases uc_1 and uc_2.

$$\exists\, uc_1, uc_2 \in \mathbb{U}$$

$$\exists\, rel \in \mathbb{r}\,.\ super(rel) = uc_2 \wedge sub(rel) = uc_1$$

(ii) The use cases uc_2 is not referenced by any other use case except uc_1.

$$\exists\, uc_2 \in \mathbb{U}$$
$$\forall\, rel{:}\, Relationship.$$
$$rel \in \mathbb{r} \land including(rel) \neq uc_2 \lor extended(rel) \neq uc_2 \lor (super\ (rel) = uc_2$$
$$\Rightarrow sub(rel) = uc_1)$$
$$\forall\, uc_i{:}\, Use\ Case.$$
$$uc_i \in \mathbb{U} \land uc_2 \notin allIncluded(uc_i) \land uc_2 \notin allExtended(uc_i)$$

Post-conditions:

(i) Use case uc_1 takes over all association relationships between use case uc_2 and its actors.

$$\exists\, uc_1, uc_2 \in \mathbb{U}'.$$
$$allActors(uc_2) \subseteq allActors(uc_1)$$

(ii) The generalization relationship between uc_1 and uc_2 is deleted.

$$\nexists\, rel \in \mathbb{r}'.\ super\ (rel) = uc_2 \land sub\ (rel) = uc_1$$

(ii) The use case uc_2 is deleted.

$$uc_2 \notin \mathbb{U}'$$

Mechanism &Verification: This refactoring merges the parent use case into the child use case.

The precondition (i) ensures a generalization relationship between the use cases. Precondition (ii) isolates the use case uc_2 from other use cases than uc_1. Since the use case

uc_2has an empty interaction, it can be merged into the use case uc_1. The interaction between the use case uc_2 and related actors is not changed. Hence behavior is preserved.

Implementation:

- Method Name: merge_UCGeneralization
- Arguments: String *subUC*, String *superUC* where
    - *subUC* is the name of the child use case
    - *superUC* is the name of the parent use case
- Return Value: String *status*

10. Merge Use Case Inclusion

Description: This refactoring merges two use cases that are related to each other by inclusion relationship. The included use case is merged into the base use case. This refactoring helps manage use case granularity and maintain the abstraction level of use cases.

Origin: From Rui [286] [page 147]

Parameters: Base Use Case uc_1 and included Use Case uc_2

Preconditions:

(i) There is an inclusion relationship between use cases uc_1 and uc_2. The use case uc_1 includes the use case uc_2.

$$\exists\, uc_1, uc_2 \in \mathbb{U}$$

$$\exists\, rel \in \mathbb{r}\,.\ addition(rel) = uc_2 \wedge including(rel) = uc_1$$

(ii) Use case uc_2 is not referenced by other use cases except uc_1.

$$\exists\, uc_2 \in \mathbb{U}$$

$$\forall\, rel{:}\, Relationship.$$

$$rel \in \mathbb{r} \wedge including(rel) \neq uc_2 \vee extended(rel) \neq uc_2 \vee super\,(rel) \neq uc_2 \vee sub\,(rel) \neq uc_2$$

$$\forall\, uc_i{:}\, Use\ Case.$$

$$uc_i \in \{\mathbb{U} - uc_1\} \wedge uc_2 \notin allIncluded(uc_i) \wedge uc_2 \notin allExtended(uc_i)$$

Post-conditions:

(i) The inclusion relationship between uc_1 and uc_2 is deleted.

$$\nexists\, rel \in \mathbb{r}'.\ addition(rel) = uc_2 \wedge including(rel) = uc_1$$

(ii) The use case uc_2 is deleted.

$$uc_2 \notin \mathbb{U}'$$

Mechanism &Verification: This refactoring merges the inclusion use case into the base use case at the point of inclusion.

The precondition (i) ensures an inclusion relationship between the use cases. Precondition (ii) isolates the use case uc_2 from other use cases than uc_1. Merging the included use case into its base use case does not alter the behavior of the use case model. Hence behavior is preserved.

Implementation:

- Method Name: merge_UCInclusion
- Arguments: String *incUC,* String *baseUC* where
    - *incUC* is the name of the inclusion use case
    - *baseUC*is the name of the base use case
- Return Value: String *status*

11. Merge Use Case Extension

Description: This refactoring merges two use cases that are related to each other by extension relationship. The extending use case is merged into the base use case. This refactoring helps manage use case granularity and maintain the abstraction level of use cases.

Origin: From Rui [286] [page 148]

Parameters: Base Use Case $uc_1$and extending Use Case uc_2

Preconditions:

(i) There is an extension relationship between use cases uc_1 and uc_2. The use case $uc_{2\ \text{Push}}$ extends the use case uc_1.

$$\exists\, uc_1, uc_2 \in \mathbb{U}$$

$$\exists\, rel \in \mathbb{r}\,.\; extension(rel) = uc_2 \wedge extended(rel) = uc_1$$

(ii) Use case uc_2 is not referenced by other use cases except uc_1.

$$\exists\, uc_2 \in \mathbb{U}$$

$$\forall\, rel{:}\, Relationship.$$

$$rel \in \mathbb{r} \;\wedge\; including(rel) \neq uc_2 \;\vee\; extended(rel) \neq uc_2 \;\vee\; super\,(rel) \neq uc_2 \;\vee\; sub\,(rel) \neq uc_2$$

$$\forall\, uc_i{:}\, Use\ Case.$$

$$uc_i \in \{\mathbb{U} - uc_1\} \;\wedge\; uc_2 \notin allIncluded(uc_i) \wedge uc_2 \notin allExtended(uc_i)$$

Post-conditions:

(i) The extension relationship between uc_1 and uc_2 is deleted.

$$\nexists\, rel \in \mathbb{r}'.\; extension(rel) = uc_2 \wedge extended(rel) = uc_1$$

(ii) The use case uc_2 is deleted.

$$uc_2 \notin \mathbb{U}'$$

Mechanism &Verification: This refactoring merges the extension use case into the base use case at the point of extension.

The precondition (i) ensures an extension relationship between the use cases. Precondition (ii) isolates the use case uc_2 from other use cases than uc_1. Merging the extension use case into its base use case does not alter the behavior of the use case model. Hence behavior is preserved.

Implementation:

- Method Name: merge_UCExtension
- Arguments: String *extUC*, String *baseUC* where
    - *extUC* is the name of the extension use case
    - *baseUC*is the name of the base use case
- Return Value: String *status*

12. Split Use Case

Description: This refactoring splits one use case into two use cases. This refactoring helps manage use case granularity.

Origin: From Rui [286] [page 159]

Parameters: Use Case *uc* and String *newUC*

Preconditions:

(i) The use case uc is not referenced by any other use case.

$$
\begin{array}{l}
\exists\, uc \in \mathbb{U} \\
\quad \forall\, rel: Relationship. \\
\qquad rel \in \mathbb{r} \wedge including(rel) \neq uc \vee extended(rel) \neq uc \vee super\,(rel) \neq uc \\
\qquad\qquad \vee\, sub\,(rel) \neq uc \\
\quad \forall\, uc_i: Use\ Case. \\
\qquad uc_i \in \mathbb{U} \wedge uc \notin allIncluded(uc_i) \wedge uc \notin allExtended(uc_i)
\end{array}
$$

(ii) The name of the new use case ($newUC$) does not conflict with the name of an existing use case within the model.

$$\forall\, uc_1 : Use\ Case\ .$$
$$uc_1 \in \mathbb{U} \implies name(uc_1) \neq newUC$$

Post-conditions:

(i) The new use case newUC is created.

$$\exists\, uc \in \mathbb{U}' \wedge uc \notin \mathbb{U} \wedge name(uc) = newUC$$

(ii) The new use case newUC is used by all actors that have an association relationship with the use case uc.

$$\exists\, uc, newUC \in \mathbb{U}'.$$
$$allActors(uc) \subseteq allActors(newUC)$$

(iii) There is no use case relationship between uc and newUC.

$$\exists\, uc,\ newUC \in \mathbb{U}'$$
$$newUC \notin allIncluded(uc) \wedge newUC \notin allExtended(uc)$$

Mechanism &Verification: This refactoring splits one use case into two use cases. The new use case has no relationship with the split use case.

The precondition (i) ensures that the use case uc has no relationship with the other use cases so that splitting uc does not change the behavior of other use cases. Precondition (ii) ensures distinct entity name invariant.

Implementation:

- Method Name: split_UC
- Arguments: String *UC*, String *newUC* where
  - *UC* is the name of the use case to be used for splitting
  - *newUC* is the name of the new use case
- Return Value: String *status*

13. Split Actor

Description: This refactoring splits one actor into two actors. This refactoring helps manage granularity. It also improves reusability of the use case model.

Origin: From Rui [286] [page 166]

Parameters: Actor a and String newActor

Preconditions:

(i) The actor a interacts with one use case in the use case model.

$$\exists\, a \in \mathbb{a}$$
$$length\left(allUC(a)\right) = 1$$

(ii) The actor a has no actor relationship with any other actor.

$$\exists\, a{:}\, Actor\ .$$
$$\forall\, rel \in \mathbb{g}\ .$$
$$super\ (rel)\ \neq a\ \vee sub(rel) \neq a$$

(iii) The name of the new actor ($newActor$) does not conflict with the name of an existing actor within the model.

$$\forall\, a_1 : Actor\ .\quad a_1 \in \mathbb{a} \Longrightarrow name(a_1) \neq newActor$$

Post-conditions:

(i) A new actor a' with name newActor is created.

$$\exists\, a' \in \mathbb{a}' \wedge a' \notin \mathbb{a} \wedge name(a') = newActor$$

(ii) The new actor newActor interacts with all use cases that the actor a interacts with.

$$\exists\, a, newActor \in \mathbb{a}'.\quad allUC(a) \subseteq allUC(newActor)$$

(iii) There is no actor relationship between a and a'.

$$\exists\, a, a' \in \mathbb{a}'.\quad \nexists\, rel \in \mathbb{g}'\ .\quad (super\ (rel) = a \vee sub(rel) = a') \vee (super\ (rel) = a' \vee sub(rel) = a)$$

Mechanism &Verification: This refactoring splits one actor into two actors. The new actor interacts with the same use cases uc that the old actor interacts with. The interaction between actor a and the use case u is preserved by the interaction between the actor a' and the use case u. Hence behavior is preserved.

The precondition (i) ensures that actor a interacts with only one use case. This simplifies the definition if the refactoring. Precondition (ii) ensures that the actor a has no

relationship with the other actors so that splitting a does not change the behavior of other actors. Precondition (iii) ensures distinct entity name invariant.

Implementation:

- Method Name: split_Actor
- Arguments: String *Actor*, String *newActor* where
  - *Actor* is the name of the actor to be used for splitting
  - *newActor* is the name of the new actor
- Return Value: String *status*

14. Use Case Generalize Generation

Description: This refactoring splits one use case into two and creates a generalization relationship between two use cases. This refactoring helps manage use case granularity. It is a special case of the "Generalize Use Case" refactoring.

Origin: From Rui [286] [page 161]

Parameters: Use case *uc*, String *newUC*

Preconditions:

(i) The use case is not referenced by any other use case.

$$\forall\, uc{:}\, UseCase\ .\quad uc \in \mathbb{U}\ \wedge allIncluded(uc) = \{\} \wedge allExtended(uc) = \{\}$$

(ii) The name of the new use case ($newUC$) does not conflict with the name of an existing use case within the model.

$$\forall\, uc_1 : Use\ Case\ .$$
$$uc_1 \in \mathbb{U} \implies name(uc_1) \neq newUC$$

Post-conditions:

A new use case is created and it is the parent of the use case uc.

$$\exists\, uc \in \mathbb{U}' \wedge uc \notin \mathbb{U} \wedge name(uc) = newUC \wedge$$
$$\exists\, rel \in \mathbb{r}' \wedge rel \notin \mathbb{r} \wedge super(rel) = newUC \wedge sub(rel) = uc$$

Mechanism &Verification: A new empty use case is created and is assigned as the parent or super use cases of the given use case uc. In the behavioral view, common interaction fragment is moved to this use case.

The precondition (i) ensures that the use case is isolated. Precondition (ii) ensures distinct entity name for the newly added parent use case.

Implementation:

- Method Name: generate_UCGeneralization
- Arguments: String *subUCName*, String *superUCName* where
    - *subUCName* is the names of the child use case
    - *superUCName* is the name of the parent use case
- Return Value: String *status*

15. Use Case Inclusion Generation

Description: This refactoring splits one use case into two and creates an inclusion relationship between the two use cases. This refactoring helps manage use case granularity and reduce redundancy.

Origin: From Rui [286] [page 162]

Parameters: Use case *uc*, String *newUC*

Preconditions:

(i) The name of the new use case ($newUC$) does not conflict with the name of an existing use case within the model.

$$\forall\, uc_1 : Use\ Case\ .\quad uc_1 \in \mathbb{U} \implies name(uc_1) \neq newUC$$

Post-conditions:

(i) A new use case uc’ with the name newUC is created.

$$\exists\, uc' \in \mathbb{U}' \wedge uc' \notin \mathbb{U} \wedge name(uc') = newUC$$

(ii) The use case uc includes the newly created use case uc’

$$\exists\, rel \in \mathbb{r}' \wedge rel \notin \mathbb{r} \wedge addition(rel) = uc' \wedge including(rel) = uc$$

Mechanism &Verification: A new empty use case is created and is assigned as the inclusion use case of the given base use case uc. The precondition (i) ensures distinct entity name for the newly added inclusion use case.

Implementation:

- Method Name: generate_UCInclusion
- Arguments: String *baseUC*, String *newUC* where
  - *baseUC* is the name of the base use case
  - *newUC* is the name of the inclusion use case
- Return Value: String *status*

16. <u>Use Case Extension Generation</u>

Description: This refactoring splits one use case into two and creates an extension relationship between the two use cases. This refactoring helps manage use case granularity and reduce redundancy.

Origin: From Rui [286] [page 163]

Parameters: Use case *uc*, String *newUC*

Preconditions:

(i) The name of the new use case ($newUC$) does not conflict with the name of an existing use case within the model.

$$\forall\, uc_1 : Use\ Case\ .$$
$$uc_1 \in \mathbb{U} \implies name(uc_1) \neq newUC$$

Post-conditions:

(i) A new use case uc' with the name newUC is created.

$$\exists\, uc' \in \mathbb{U}' \wedge uc' \notin \mathbb{U} \wedge name(uc') = newUC$$

(ii) The newly added use case uc' extends the use case uc.

$$\exists\, rel \in \mathbb{r}' \wedge rel \notin \mathbb{r} \wedge extension(rel) = uc' \wedge extended(rel) = uc$$

Mechanism &Verification: A new empty use case is created and is assigned as the extension use case of the given base use case uc. The precondition (i) ensures distinct entity name for the newly added extensionuse case.

Implementation:

- Method Name: generate_UCExtension
- Arguments: String *baseUC*, String *newUC* where
  - *baseUC* is the name of the base use case
  - *newUC* is the name of the extension use case
- Return Value: String *status*

17. Actor Generalize Generation

Description: This refactoring splits one actor into two and creates a generalization relationship between the two actors. This refactoring helps manage improve the understandability and reusability of the use case model.

Origin: From Rui [286] [page 168]

Parameters: Actor *a*, String *newActor*

Preconditions:

(i) The actor does not have a parent actor.

$$\forall\, a{:}\, Actor\, .$$
$$a \in \mathbb{a} \;\wedge$$
$$\forall\, rel\, \in\, \mathbb{g}\, .\; sub(rel) \neq a$$

(ii) The name of the new actor ($newActor$) does not conflict with the name of an existing actor within the model.

$$\forall\, a_1{:}\, Actor\, .$$
$$a_1 \in \mathbb{a} \;\Longrightarrow name(a_1) \neq newActor$$

Post-conditions:

(i) A new actor a’ with the name newActor is created and it is the parent of actors a.

$$\exists\, a' \in \mathbb{a}' \wedge a' \notin \mathbb{a} \wedge name(\, a') = newActor \;\wedge$$
$$\exists\, rel \in \mathbb{g}' \wedge rel \notin \mathbb{g} \wedge super(\, rel) = a' \wedge sub(\, rel) = a$$

Mechanism &Verification: A new actor is created and is assigned as the parent or super actor of the given actor a. The precondition (i) ensures unique parent. Precondition (ii) ensures distinct entity name for the newly added parent actor.

Implementation:

- Method Name: generate_ActorGeneralization
- Arguments: String Actor, String newActor where

- *Actor* is the names of the actor used for splitting
- *newActor* is the name of the new parent actor

- Return Value: String *status*

A2.2 Class Model Refactoring

1. Pull Up Attribute

Description: This refactoring removes one attribute from a class or a set of classes and inserts it into one of its superclasses. It is the analogous to Fowler et al.'s Pull Up Attribute for Code Refactoring. If you pull up an attribute, the new visibility should be set to the maximum visibility of this attribute in the subclasses. At least all subclasses should still have access to the attribute after refactoring.

Origin: From Mantz [461] [page 95]

Parameters: String *superClass*, String *attr*

Preconditions:

(i) The attribute ($attr$) is owned by the same type by all classes that has the super class ($superClass$) as their parent class.

$$\forall\, g_1 : Generalization\,.$$

$$\exists\, atrib : name(atrib) = attr.$$

$$g_1 \in \mathbb{R} \wedge super(g_1) = superClass \;\Rightarrow\; atrib \in \boldsymbol{Attr}\big(sub(g_1)\big) \wedge type(atrib) = type(\boldsymbol{Attr}\big(sub(g_1)\big))$$

(ii) The super class ($superClass$) must not have an attribute with the same name.

$$\forall\, g_1 : Generalization\,.$$
$$\exists\, atrib : name(atrib) = attr.$$
$$g_1 \in \mathbb{R} \wedge super(g_1) = superClass \;\Rightarrow\; atrib \notin \boldsymbol{Attr}\big(super(g_1)\big)$$

Post-conditions:

(i) The super class ($superClass$) has an attribute with the same name and type as the attributes in the subclasses.

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge attr \in \boldsymbol{Attr}\big(super(g_2)\big)$$

(ii) The child classes of the super class ($superClass$) has no attribute with the name ($attr$).

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge attr \notin \boldsymbol{Attr}\big(sub(g_2)\big)$$

Mechanism &Verification: The behavior of the class model is not affected with the pulling up of the attribute. Based on the laws of inheritance, these attributes can still be accessed from the super class and since the attribute visibility is changed to the maximum (either public or protected); they can be accessed from the subclasses without any restriction.

Implementation:

- Method Name: pullup_Attribute
- Arguments: String *superClass* , String *attr* where
    - *superClass* is the name of the parent class

- *attr* is the name for the attribute to be pulled into the parent class

- Return Value: String *status*

2. Pull Up Method

Description: This refactoring moves a method of a class to its super class. Usually this refactoring is used simultaneously on several classes which inherit from the same super class. The aim of this refactoring is often to extract identical methods. This refactoring is analogous to Fowler et al.'s Pull Up Method for Code Refactoring. In order to keep the view consistent, Pull Up Method is often used with Pull Up Attribute. In most cases, it is also important that the operation is still visible in the subclass after refactoring Pull Up Method.

Origin: From Mantz [461] [page 106]

Parameters: String *superClass*, String *method*, ArrayList *signature*

Preconditions:

(i) The super class ($superClass$) must not have a method with the same name and signature.

$$\forall\, g_1 : Generalization\,.$$
$$\exists\, op : name(op) = method.$$
$$g_1 \in \mathbb{R} \wedge super(g_1) = superClass \;\Rightarrow\; op \notin \boldsymbol{Op}(super(g_1))$$

(ii) All the sub classes of the parent ($superClass$) must have a method with the same name and signature.

$$\forall\ g_1 : Generalization\ .$$
$$\exists\ op : name(op) = method.$$
$$g_1 \in \mathbb{R} \wedge super(g_1) = superClass \Rightarrow op \in \boldsymbol{Op}(sub(g_1)) \wedge same_param(Param(op), Param\ (\boldsymbol{Op}(sub(g_1))))$$

In the above precondition, we define an auxiliary function $same_param$ that checks whether the parameters (also known as method signature) of the methods are same. This Function can be formally written as

$$same_param(Param(op_1), Param(op_2)) \equiv$$
$$length(Param(op_1)) = length(Param(op_2))$$
$$\Longrightarrow (\exists i : i \leq length\ (Param(op_1)) \wedge type(Param(op_1)) = type(Param(op_2)))$$

Post-conditions:

(i) The super class ($superClass$) has a method with the same name and signature as the method in the subclasses.

$$\exists\ g_2 \in \mathbb{R}' \wedge super(g_2) = superClass\ \wedge op \in \boldsymbol{Op}(super(g_2))$$

(ii) The child classes of the super class ($superClass$) has no method with the name ($method$) and signature.

$$\exists\ g_2 \in \mathbb{R}' \wedge super(g_2) = superClass\ \wedge op \notin \boldsymbol{Op}(sub(g_2))$$

Mechanism &Verification: The behavior of the class model is not affected with the pulling up of the method. Based on the laws of inheritance, this method can still be accessed from the super class and since the method visibility is changed to the maximum (either public or protected); it can be accessed from the subclasses without any restriction.

Implementation:

- Method Name: pullup_Method
- Arguments: String *superClass* , String *method,* ArrayList *signature* where
    - *superClass* is the name of the parent class
    - *method* is the name for the method to be pulled into the parent class
    - *signature* is the parameter list of the method to be pulled
- Return Value: String *status*

3. <u>Push Down Attribute</u>

Description: Refactoring Push Down Attribute moves an attribute to all subclasses. In the literature, refactoring Push Down Attribute is often limited to subclasses that require the attribute. In case of code refactoring these classes can be indicated. In case of UML models this is usually not possible, but it can be nevertheless useful to push down a property to all subclasses e.g. as preparation before deleting the superclass.

Origin: From Mantz [461] [page 109]

Parameters: String *superClass*, String *attr*

Preconditions:

(i) No direct subclass contains an attribute with the same name as the attribute that is being pushed down.

$$\forall\, g_1 : Generalization\,.$$
$$\exists\, atrib : name(atrib) = attr.$$
$$g_1 \in \mathbb{R} \wedge super(g_1) = superClass \;\Rightarrow\; atrib \notin \boldsymbol{Attr}\big(sub(g_1)\big)$$

Post-conditions:

(i) The attribute $(attr)$ is defined in all subclasses.

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge attr \in \boldsymbol{Attr}\big(sub(g_2)\big)$$

(ii) The attribute $(attr)$ does not exist in the super class any more.

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge attr \notin \boldsymbol{Attr}\big(super(g_2)\big)$$

Mechanism &Verification: The behavior of the class model is not affected with the pushing down of the attribute. Precondition (i) ensures that the attribute is not overwritten in the sub classes. Any subclass not using the attribute can be later deleted.

Implementation:

- Method Name: pushdown_Attribute
- Arguments: String *superClass* , String *attr,* where
    - *superClass* is the name of the parent class
    - *attr* is the name for the attribute to be pushed down into the child classes

- Return Value: String *status*

4. Push Down Method

Description: The refactoring Push Down Method pushes a method down to all its subclasses. It is analogous to Fowler et al.'s refactoring Push Down Method. In the literature, the Push Down Operation refactoring is often limited to subclasses that really require the operation. In case of code refactoring these classes can be indicated. However, in case of UML models the necessity of pushing down an operation can usually not be automatically construed (a possible solution is to inspect sequence diagrams).

Origin: From Mantz [461] [page 115]

Parameters: String *superClass*, String *method*, ArrayList *signature*

Preconditions:

(i) The super class ($superClass$) has subclasses.

$$\forall\, g_1 : Generalization\,.$$
$$g_1 \in \mathbb{R} \Rightarrow super(g_1) = superClass$$

(ii) The method ($method$) does not exist in any direct subclass.

$$\forall\, g_1 : Generalization\,.$$
$$\exists\, op : name(op) = method.$$
$$g_1 \in \mathbb{R} \wedge super(g_1) = superClass \Rightarrow op \notin \boldsymbol{Op}\big(sub(g_1)\big)$$

Post-conditions:

(i) The method ($method$) does not exist anymore in the super class ($superClass$).

$$\exists\ g_2 \in\ \mathbb{R}' \wedge super(g_2) = superClass\ \wedge op \notin \boldsymbol{Op}\big(super(g_2)\big)$$

(ii) The method ($method$) exists in all subclasses.

$$\exists\ g_2 \in\ \mathbb{R}' \wedge super(g_2) = superClass\ \wedge op \in \boldsymbol{Op}\big(sub(g_2)\big)$$

Mechanism &Verification: The behavior of the class model is not affected with the pushing down of the method. Precondition (ii) ensures that the method is not overwritten in the sub classes. Any subclass not using the method can be later deleted.

Implementation:

- Method Name: pushdown_Method
- Arguments: String *superClass* , String *method,* ArrayList *signature* where
    - *superClass* is the name of the parent class
    - *method* is the name for the method to be pushed into the child classes
    - *signature* is the parameter list of the method to be pushed
- Return Value: String *status*

5. Remove Empty Superclass

Description: A set of classes has an empty super class which shall be removed. This refactoring often follows Push Down Attribute and Push Down Method Refactoring or in

the intermediate version also by the Pull Up Attribute or Pull Up Method Refactoring. In the intermediate version of this refactoring the empty super class inherits from a super class.

Origin: From Mantz [461] [page 112]

Parameters: String *superClass*

Preconditions:

(i) The super class ($superClass$) has no attributes and methods (it should be empty).

$$\forall\, g_1 : Generalization\,.$$
$$\exists\, op : name(op) = method.$$
$$g_1 \in \mathbb{R} \wedge super(g_1) = superClass \;\Rightarrow\; op \notin \boldsymbol{Op}\big(super(g_1)\big)$$

Post-conditions:

(i) The super class ($superClass$) does not exist anymore.

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge op \in \boldsymbol{Op}\big(super(g_2)\big)$$

(ii) All classes still inherit all operations and attributes of potential super classes of the ($superClass$)

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge op \notin \boldsymbol{Op}\big(sub(g_2)\big)$$

Mechanism &Verification: The behavior of the class model is not affected with the deletion of the superclass. Precondition (i) and (ii) ensures that the class is empty and isolated from other attribute and method references. Precondition (iii) ensures that no

behavior is lost with the refactoring as the super class was an abstract class. Postcondition (ii) ensures that any inheritance relationship that exists between the superClass and other classes (i.e. the deleted super class was a sub class to other super classes) is preserved as these features are inherited in all the sub classes.

Implementation:

- Method Name: remove_SuperClass
- Arguments: String *superClass* where
    - *superClass* is the name of the parent class to be removed
- Return Value: String *status*

6. Remove Empty Subclass

Description: Refactoring Remove Empty Subclass removes an empty subclass from the model.

Origin: From Mantz [461] [page 99]

Parameters: String subClass

Preconditions:

(i) The subclass ($subClass$) has no attributes and methods (it should be empty).

$$\forall\, g_1 : Generalization\,.$$

$$\exists\, op : name(op) = method.$$

$$g_1 \in \mathbb{R} \wedge sub(g_1) = subClass \Rightarrow op \notin \boldsymbol{Op}(sub(g_1))$$

Post-conditions:

(i) The subclass ($subClass$) and its inheritance relation do not exist anymore.

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge op \in \boldsymbol{Op}(super(g_2))$$

Mechanism &Verification: The behavior of the class model is not affected with the deletion of the subclass. Precondition (i) ensures that the class is empty and isolated from other attribute and method references.

Implementation:

- Method Name: remove_SubClass
- Arguments: String *subClass* where
    - *subClass* is the name of the child class to be removed
- Return Value: String *status*

7. Create Super Class

Description: Refactoring Create Super Class is used to create a super class for at least one class which is normally followed by Pull Up Attribute and Pull Up Method Refactorings. In addition, this refactoring can create an intermediate super class that is a super class that is introduced between a set of classes and their former super classes.

Origin: From Mantz [461] [page 103]

Parameters: String newClass, ArrayList subClasses, Boolean abstract_flag, Boolean intermediate

Preconditions:

(i) The class name for the new super class ($newClass$) must be unique.

$$\forall\, g_1 : Generalization\,.$$
$$\exists\, op : name(op) = method.$$
$$g_1 \in \mathbb{R} \wedge super(g_1) = superClass \;\Rightarrow\; op \notin \boldsymbol{Op}\big(super(g_1)\big)$$

(ii) In the case that the ($intermediate$) flag is true, the classes within the selected set of classes ($subClasses$) must have at least one common super class.

$$\forall\, g_1 : Generalization\,.$$
$$\exists\, op : name(op) = method.$$
$$g_1 \in \mathbb{R} \wedge super(g_1) = superClass \;\Rightarrow\; op \notin \boldsymbol{Op}\big(super(g_1)\big)$$

Post-conditions:

(i) The new Class ($newClass$) exists in the Class Model.

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge op \in \boldsymbol{Op}\big(super(g_2)\big)$$

(ii) There exists an inheritance relation to the super class ($newClass$) for each input class in the set ($subClasses$).

$$\exists\, g_2 \in \mathbb{R}' \wedge super(g_2) = superClass \wedge op \in \boldsymbol{Op}\big(super(g_2)\big)$$

(iii) If the super class ($newClass$) is an intermediate one, it must inherit from all common super classes of the selected set of subclasses ($subClasses$). Furthermore, there is no direct relation anymore between these super classes and the classes of this set ($subClasses$).

$$\exists\ g_2 \in\ \mathbb{R}' \wedge super(g_2) = superClass\ \wedge op \in \boldsymbol{Op}\big(super(g_2)\big)$$

Mechanism &Verification: The behavior of the class model is not affected with the creation of the new super class. Precondition (i) ensures that the new class is unique to the class model. In case the new class is an intermediate class between an existing inheritance, Precondition (ii) and Postcondition (iii) ensure that the new class inherits from all the common superclasses of the set of subclasses and that these subclasses have no more direct access to the superclasses.

Implementation:

- Method Name: create_SuperClass
- Arguments: String *newClass,* ArrayList *subclasses,* Boolean *isabstract,* Boolean *intermediate* where
  - *newClass* is the name of the new class to be created
  - *subClasses* is the set of classes that will be child classes to the newly created super class
  - *isabstract* is a flag that identifies whether the newly created flag is set to either abstract or concrete.

- *intermediate* is a flag which is set when the newly created flag is an intermediate class in an existing inheritance relationship.

- Return Value: String *status*

A2.3 Sequence Model Refactoring

1. Create Lifeline

Description: Create Lifeline Refactoring is used to introduce a new lifeline into a sequence diagram.

Origin: From Meng and Barbosa [462]

Parameters: String *newLifeline*

Preconditions:

(i) The lifeline ($newLifeline$) must be unique in the sequence diagram.

$$\forall\, l_1 : Lifeline\ .\quad l_1 \in \mathbb{L} \implies name(l_1) \neq newLifeline$$

Post-conditions:

$$\exists\, l \in \mathbb{L}' \wedge l \notin \mathbb{L} \wedge name(l) = newLifeline$$

Mechanism &Verification: The behavior of the sequence model is not affected with adding a new lifeline since there is no message exchanges between the new lifeline and

the exisiting lifelines within the sequence diagram. The precondition (i) ensures that the new lifeline is unique to the sequence model.

Implementation:

- Method Name: create_Lifeline
- Arguments: String *newLifeline* where
  - *newLifeline* is the name of the new lifeline to be added
- Return Value: String *status*

2. Remove Lifeline

Description: Refactoring Remove Lifeline is used to remove a lifeline that does not interact with other participants and has no local actions within the sequence diagram.

Origin: From Meng and Barbosa [462]

Parameters: String *Lifeline*

Preconditions: The lifeline is isolated from other particpants of the sequence diagram. Isolation from other participants means

- No message exchanges
- No local actions

$$\exists\, l : SEQ\ .$$
$$name(l) = Lifeline\ \wedge\ length\big(endList(l)\big) = 0$$

Post-conditions:

$$l \notin \mathbb{L}'$$

Mechanism &Verification: Since the lifeline is isolated from other participating lifelines within the sequence diagram, it does not affect interactions between them. Hence, this deletion does not change the behavior of the sequence model.

Implementation:

- Method Name: remove_Lifeline
- Arguments: String *lifeline* where
    - *lifeline* is the name of the lifeline
- Return Value: String *status*

Appendix 3: XML & Associated Standards

This section introduces the Extensible Markup Language (XML) and other technologies associated with it such as the XML Schema, XPath, XSLT and XML Metadata Interchange (XMI) format.

A3.1 eXtensible Markup Language (XML)

XML [463] is a *"World Wide Web Consortium (W3C)-recommended general-purpose specification for creating custom markup languages".* The Extensible Markup Language is a simple and flexible text format used widely in the exchange of varied data on the web and elsewhere. It is a free, platform-independent open-standard derived from the Standardized Generalized Markup Language (SGML) in order to meet the challenges of large-scale electronic publishing. Although a lot similar to the Hypertext Markup Language (HTML), XML was designed to describe data instead of focusing on how data looks and how it is displayed. For example, Figure A - 1 shows how XML can be used to describe this dissertation.

```
<?xml version="1.0" encoding="UTF-8"?>
<dissertation_file>
        <dissertation>
                <id>Fall2010_001</id>
                <author>Mohammed Misbhauddin</author>
                <title>Towards an Integrated Metamodel based approach to
                       Software Refactoring </title>
                <advisor>Mohammad Alshayeb</advisor>
                <co_advisor>Radwan Abdel-Aal</co_advisor>
                <committee_member>Moataz Ahmed</committee_member>
                <committee_member>Mohammed Elish</committee_member>
                <committee_member>Aiman El-Maleh</committee_member>
        </dissertation>
</dissertation_file>
```

Figure A - 1: XML Document for Dissertation Example

Elements inside an XML document represent structured values. Element names with or without attributes are referred to as tags. A general form of an XML element is given as

<name attributes>content</name>

All XML elements begin with the element's start tag (formatted as <name>) and close with the element's end tag (formatted as </name>). The end tag is mandatory but can be omitted if there is no content by using the format <name attributes />. The attributes is an optional list of attributes and their values. For instance, we can add an attribute to the *<committee_member>* tag mentioned in the above XML example as follows:

<committee_member rank = "Associate">Moataz Ahmed</committee_member>

Names used for elements and attributes in an XML document can contain nearly every letter, number or special character with the exception of white space characters and punctuation characters (such as :, &).

One of the major advantages of XML is that it allows designers to create their own customized tags. Hence, tag names describe the data they contain and are regarded as metadata. Tag names should be meaningful so that information labeled is reusable. Although flexible, XML still requires the document to be well-formed and valid to be considered correct and usable. An XML document is considered “well-formed” if it conforms to the rules of the XML specification (such as using lowercase letters in tags, including closing tags on all elements, and including single and double quotation marks on all attribute tags).

The structure of an XML document can be defined by a schema language and is validated based on definitions in that language. A “valid” XML document apart from being well-

formed also conforms to the rules defined by the schema language. Two of the most widely used schema languages are the Document Type Definition (DTD) language and XML Schema.

A3.2 Document Type Definition (DFD)

A DTD is used to define the building blocks of an XML document and describe the document structure with a list of valid elements. Defining a DTD allows an XML designer to build his own set of rules and restrictions to be enforced on the resulting XML document. Figure A - 2provides a DTD for the example XML shown in Figure A - 1.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT dissertation_file ((dissertation))>
<!ELEMENT dissertation ((id, author, title, advisor, co_advisor,
                              committee_member+))>
<!ELEMENT id (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT advisor (#PCDATA)>
<!ELEMENT co_advisor (#PCDATA)>
<!ELEMENT committee_member (#PCDATA)>
```

Figure A - 2 DTD for the Dissertation example

If an XML document has elements that match element declarations in a DTD, the document is considered a valid document. An element declaration in a DTD consists of the name of the element, its content and its attributes. The general format of an element declaration is as follows:

```
<!ELEMENT name content>
<!ATTLIST name attribute-decls>
```

The name is the tag name and content specifies what kind of data can be included. The content in an element declaration can be EMPTY, ANY (text of other XML elements) or #PCDATA (text). The attribute-decls is of the form <type default> where type can be CDATA (character data), set of valid values, ID, IDREF or IDREFS. The default is optional and can be #REQUIRED, #IMPLIED, #FIXED (along with a fixed value) or an attribute value. ID, IDREF and IDREFS enable XML elements to be related to each other.

An XML parser (also known as a validating parser) can be used to validate an XML document. In order for the parser to know about the DTD of an XML document, it is specified as a DOCTYPE statement in the document to be validated. Its format is given as follow:

```
<!DOCTYPE name SYSTEM " sample.dtd">
```

The name is the tag name of the root element and sample.dtd is the Uniform Resource Locator (URI) that specifies the location of the DTD.

A3.3 XML Schema

XML Schemas are another important leap in the evolution of XML. They deprecated the use of DTDs by allowing designers to specify more constraints on XML documents than DTDs. Since the discussion about XML Schemas is exhaustive, we include only the relevant ones in this section.

The XML Schema language is also referred to as XML Schema Definition (XSD). A schema document is an XML document. The context of the XML elements is defined by

the schema namespace. All schema documents need to have a schema XML element as the root XML element.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

Schemas usually contain element and type declarations. Each element declared uses the XML element called element. The name attribute of that element is the name of the element. An example declaration is as follows:

```
<xs:element name="id">
```

Schemas also allow the creation of types. These types actually do not appear in the XML documents, but are used to declare other elements and attributes that may appear. Types in schemas are of two kinds: simple and complex types. Simple types represent data values and complex types represent data structure. Contents of an XML element can be specified by using XML elements in the content of the complexType element. An example complex type declaration with content is as follows:

```
<xs:complexType>
    <xs:sequence>
        <xs:element name="dissertation"/>
    </xs:sequence>
</xs:complexType>
```

Schema designers can express repetition in element content by using the sequence XML element and express alternatives by using the choice XML element. In order to specify the number of occurrences of elements in the element content, we can use the minOccurs and maxOccurs attributes in the element, sequence or choice XML elements. An example declaration that demonstrates this is as follows:

```
<xs:element name="committee_member" minOccurs="2" maxOccurs="3"/>
```

Attributes can also be declared in schemas. The *xs:attribute* XML element can be used for this purpose. The attribute element has name, type, use, default and fixed as its attributes. The type should be simple and not complex. The use attribute constrains whether the attribute is optional, prohibited or required. A *default* value of an attribute can be assigned using the default XML attribute and a specific value can be assigned using the fixed XML attribute.

An XML Schema for the XML example shown in Figure A - 1 is given in Figure A - 3.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
	<xs:element name="dissertation_file">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="dissertation"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
	<xs:element name="dissertation">
		<xs:complexType>
			<xs:sequence>
				<xs:element name="id"/>
				<xs:element name="author"/>
				<xs:element name="title"/>
				<xs:element name="advisor"/>
				<xs:element name="co_advisor"/>
				<xs:element name="committee_member"
minOccurs="2" maxOccurs="3"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
</xs:schema>
```

Figure A - 3 XML Schema Definition for Dissertation Example

Similar to that of DTDs, an XML schema definition should be included in the XML document for it to be validated by an XML parser. This is done by including the

xmlns:xsi attribute at the top level element and the schemaLocation attribute identifies the location of a particular XML schema.

```
<dissertation_file xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:SchemaLocation="E:\sample.xsd">
```

A3.4 XML Path Language (XPath)

The main purpose of XPath is to address parts of an XML document. It operates on the abstract, logical structure of an XML document, rather than its surface syntax. XPath gets its name from its use of a path notation as in Uniform Resource Locators (URL) for navigating through the hierarchical structure of an XML document. XPath can be seen as an expression language that works on a data model defined by XQuery/XPath Data Model (XDM). XDM provides a tree representation of XML documents. An XPath expression then can be used for the selection of nodes from the input documents.

The main construct in XPath is the path expression. A path expression is used to locate nodes within an XML tree and consists of one or more steps. Each step in a path expression is separated by a / or //. Steps in the expression are either axis steps or filter steps. Axis steps define the direction of traversal within the tree and filter steps define conditional selection of nodes. All axes supported by XPath are given in Table A - 1. Some frequently used axes are abbreviated for ease of use.

Table A - 1 Axes provided by XPath

Axis	Description	Abbreviations
self::	The context node itself	.
attribute::	Attributes of the context node	@
parent::	The parent of the context node	..
child::	Children of the context node	Can be omitted
descendant::	All children of the context node	
descendant-or-self::	The context node and its descendants	//
ancestor::	All ancestors of the context node	
ancestor-or-self::	The context node and all its ancestors	
preceding::	All nodes that precede the context node in the document	
preceding-sibling::	All the siblings of the context node that precede it	
following::	All nodes that follow the context node in the document	
following-sibling::	All siblings of the context node that follow it	

An example XPath expression to retrieve all committee members with a rank of associate is given as follows:

// /committee_member[@rank='Associate']

Path expressions are evaluated from the left to the right side: the slashes // traverse the descendant-or-self axis of the XML tree starting from the root node, searching element nodes named committee_member, and selecting each as the current context node. A filter expression [condition] is now applied to the context node and the attribute axis is inspected for an attribute named rank holding a value “Associate”. If this condition is true, the context node is included in the resulting sequence. All matching nodes are returned. The result of an XPath expression can be a node-set (an unordered collection of

nodes without duplicates), Boolean, Number or a String. The manner in which an expression is evaluated is based on a context.

The latest version XPath 2.0, is a superset of XPath 1.0 with added ability to support a set of new data types and also to make use of the type information that becomes available when documents are validated using XML Schema.

A3.5 XML Query Language (XQuery)

XQuery is *"a standardized language for combining documents, databases, Web pages, and almost anything else. It is very widely implemented. It is powerful and easy to learn."* [464] It is a language maintained by W3C in order to express queries across XML documents. It allows designers to select XML elements from the source file, reorganize and transform them. XPath and XQuery go hand in hand with each other. XPath is a complete subset of XQuery. Both XPath and XQuery documents are built around expressions rather than statements. The major difference between XQuery and XPath is that XPath only allows the capability to retrieve nodes from an XML document. The former allows creation of new nodes and modification of existing nodes.

The basic structure of most queries in XQuery is the FLWOR (pronounced as flower) expression. It stands for **F**or, **L**et, **W**here, **O**rder by and **R**eturn. An example XQuery expression using the FLWOR expression to restructure the Dissertation XML tree (Figure A - 1) and returning a sequence containing all committee members sorted by their rank is shown in Figure A - 4.

```
xqueryversion"1.0";
for $committee in doc("dissertation.xml")    //committee_member
let $rank := data($committee/@rank)
orderby $rank
return element committee-members { $committee}
```

Figure A - 4 Example XQuery expression

A3.6 eXtensible Stylesheet Language Transformations (XSLT)

XSLT [465] is a functional transformation language for manipulating XML data. Being a functional language, rules have to be called explicitly. There is no built-in traceability support and rules are strictly unidirectional. Transformations are stateful, so there is no support for incremental transformation. XSLT transformation descriptions are themselves XML documents, so higher-order transformations can be realized.

An XSL processor parses an XML source document and tries to find a matching template rule. If it does, instructions inside matching template are evaluated. A template rule is written as follows:

```
<xsl:template match="string">
        instructions
</xsl:template>
```

Contents of the original elements from the source XML can be obtained by making use of the *xsl:value-of* construct. Location paths determine parts of XML document to which template should be applied. The required syntax is specified in the XPath specification. XPath along with XSLT is used in transformation of XML documents.

A3.7 XML Metadata Interchange (XMI)

XMI [35] is a standard interchange format for data objects in XML. It is defined and maintained by the Object Management Group (OMG). Since XMI provides a standard representation of objects in XML, it is used effectively to exchange objects using XML. The main purpose of XMI is to allow for exchange of objects from the OMG Object Design and Analysis Facility. These objects are more commonly known as UML (Unified Modeling Language) and MOF (Meta Objects Facility).

An example UML class diagram is given in Figure A - 5 and its corresponding XMI created using the open-source UML modeling tool ArgoUML [445] is given in Figure A - 6. Some attributes are left out and values for xmi.id and xmi.idref (which are automatically generated) are changed for brevity and readability.

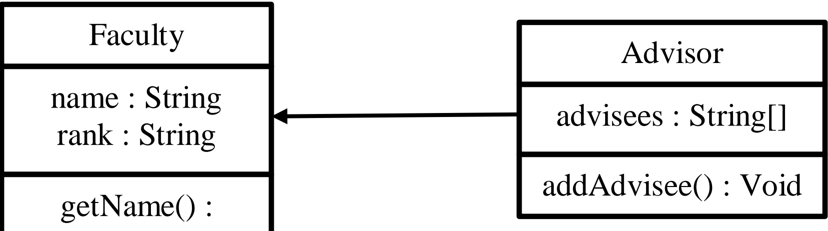


Figure A - 5 UML Class Diagram: Class Advisor inherits from class Faculty

```
<UML:Class xmi.id='_classAdvisor' name='Advisor' visibility='public'>
	<UML:Generalization xmi.idref='_genFacultyAdvisor'/>
		<UML:Classifier.feature>
			<UML:Attribute name='advisees' visibility='private'/>
			<UML:Operation name='addAdvisee' visibility='public'/>
		</UML:Classifier.feature>
</UML:Class>
<UML:Class xmi.id='_classFaculty' name='Faculty' visibility='public'>
	<UML:Generalization xmi.idref='_genFacultyAdvisor'/>
		<UML:Classifier.feature>
			<UML:Attribute name='name' visibility='private'/>
			<UML:Attribute name='rank' visibility='private'/>
			<UML:Operation name='getRank' visibility='public'/>
		</UML:Classifier.feature>
</UML:Class>
<UML:Generalization xmi.id='_genFacultyAdvisor'>
	<UML:Generalization.child>
		<UML:Class xmi.idref='_classAdvisor'/>
	</UML:Generalization.child>
	<UML:Generalization.parent>
		<UML:Class xmi.idref='_classFaculty'/>
	</UML:Generalization.parent>
</UML:Generalization>
```

Figure A - 6 XMI Representation of the UML Class Diagram in Figure A-5

Appendix 4: XQuery Functions for Integrated Model Smells

A4.1 Creeping Featurism

Listing 1 shows the XQuery function to detect instances of the bad smells *Creeping Featurism.* The function `Func-Decompose` is parameterized with the source model $model. Using a *"for loop"*, each use case in the integrated model is bound to the variable $x one after another in line 3. The function `inclusion` called in line 4 returns the number of times a use case is included in other use cases and is shown in Listing 2. If the inclusion count is exactly one and there are no actors associated with the use case, function `inc` is called for behavioral analysis of the use case.

```
1  declare function local:Func-Decompose($model as node())
2  {
3      for $x in $model//IntegratedModel/UseCase
4        return if ((count(local:inclusion($model, $x))=1) and (empty($x/@actor-ref)))
5                   then local:inc($x,$model)
6                 else ()
7  } ;
```

Listing 1: XQuery function to detect the bad smell Creeping Featurism

Listing 2 shows the function `inclusion`. Each use case in the integrated model is composed of a number of inclusion use cases identified by the "*includes*" tag in the XMI file. The function `inclusion` returns the number of times a use case is included in other use cases.

```
1  declare function local:inclusion($a as node(), $x as node())
2  {
3    for $y in $a//IntegratedModel/UseCase/includes
4            return if (data($x/@name)=data($y/@uc-ref))
5                          then data($x/@name)
6                     else ()
7  } ;
```

Listing 2: XQuery function to count the number of inclusions of a use case in other use cases

Listing 3 shows the function `inc`. If this function returns a non-empty sequence, then the existence of this bad smell is confirmed. The *"if statement"* in lines 4-6 identifies the including use case of the parameter $uc. Then the `value-intersect` function, that returns the intersection of the values in two sequences, is used to first identify dissimilar lifelines in both the use cases. This intersection sequence is then checked to see if it consists of data classes. If true, this use case is returned as a candidate for refactoring.

```
1  declare function local:inc($uc as node(),$model as node())
2  {
3           for $y in $model//IntegratedModel/UseCase
4              return if (data($uc/@name) = data($y/includes/@uc-ref) and
5                             hr:value-intersect(local:data-class($model),
6                             hr:value-intersect(local:lifelines($uc),local:lifelines($y))))
7                       then data($uc/@name)
8                    else ()
9  } ;
```

Listing 3: XQuery function to analyze use case inclusion behavior

A4.2 Multiple Personality

Listing 4 shows the XQuery function to detect instances of the bad smell *Multiple Personality*. The function `Multi-Personality` is parameterized with the source model $model. Using a *"for loop"*, each use case in the integrated model is bound to the

variable $x one after another in line 3. The functions `lifelines` (see Listing 5) and `transactions` (see Listing 6) called in line 4 and 5 returns the number of lifelines and transactions in a use case respectively. The function `lazy-class` called in line 6 returns a sequence of all lazy classes in the model (see Listing 7). If the number of lifelines is greater than ten and the numbers of transactions are greater than seven and the number of lifelines that are subsets of the lazy class sequence are more than or equal to two, function `middle-man` (see Listing 8) is called for analyzing the inter-lifeline behavior in the use case. If this function returns a non-empty sequence, then the existence of this bad smell is confirmed.

```
1  declare function local:Multi-Personality($model as node())
2  {
3    for $x in $model//IntegratedModel/UseCase
4       return if ((count(local:lifelines($x)) > 10) and
5            (count(local:transactions($x)) > 7) and
6            count(hr:value-intersect(local:lazy-class($model),local:lifelines($x))) >= 2)
7                then local:middle-man($model,$x)
8                else ()
9  } ;
```

Listing 4: XQuery function to detect the bad smell Multiple Personality

Listing 5 and Listing 6 depicts the pseudo-code for two simple functions that return the number of lifelines and transactions within a use case behavior respectively.

```
1  declare function local:lifelines($a as node())
2  {
3          let $sequence := for $y in $a/Interaction/Lifeline
4             return data($y/@name)
5          return $sequence
6  } ;
```

Listing 5: XQuery function to count the number of lifelines within a use case interaction

```
1  declare function local:transactions($a as node())
2  {
3            let $sequence := for $y in $a/Interaction/Message
4               return data($y/@id)
5            return $sequence
6  } ;
```

Listing 6: XQuery function to count the number of transactions within a use case interaction

A class is termed as a lazy class when it has more attributes than functions. Listing 7 provides the pseudo-code for detecting whether a class is a lazy class or not.

```
1  declare function local:lazy-class($a as node())
2  {
3            let $sequence := for $y in $a//IntegratedModel/Class
4               return if (count($y/Property) >count($y/Message))
5               then data($y/@name)
6                     else ()
7            return $sequence
8  } ;
```

Listing 7: XQuery function to check whether a class is a Lazy Class

In Listing 8, for each lifeline in the identified God Use Case, the patterns of the event ends are checked. In line 5, if the number of ends incident to a lifeline are more than two and are even in number, a recursive function called `mm-pattern` (see Listing 9) is called to check the middle man pattern.

```
1  declare function local:middle-man($model as node(), $uc as node())
2  {
3            for $x in $uc/Interaction/Lifeline
4                     let $val :=  count($x/end)
5                     return if ($val > 2 and local:isEven($val))
6                          then local:mm-pattern($model, $x, count($x/end))
7                       else data(1)
8  } ;
```

Listing 8: XQuery function to detect a middle-man lifeline within a use case interaction

Listing 9 shows the recursive function `mm-pattern`. This function breaks down the end list into sub-sequences of size 2 and compares them with the sequence {receiveEvent, sendEvent}. In case of a complete match, the lifeline can be safely classified as a middle-man.

```
1   declare function local:mm-pattern($Life as node(), $ends as xs:integer)
2   {
3      let $end-list := local:end-list($Life)
4            return if ($ends = 0)
5                        then data(2)
6            else
7                        if (subsequence($end-list, $ends, 2) = ('receiveEvent', 'sendEvent'))
8                                then local:mm-pattern($Life, $ends - 2)
9                        else data(1)
10  } ;
```

Listing 9: XQuery function to detect middle-man pattern recursively

A4.3 Excessive Alternation

Listing 10 shows the XQuery function to detect instances of the bad smell *Excessive Alternation*. The function `Excessive-Alternation` is parameterized with the source model $model. Using a *"for loop"*, each use case in the integrated model is bound to the variable $x one after another in line 3. The functions `extPoints` (see Listing 11) called in line 4 returns the number of extension points in a use case. If the number of extension points is greater than or equal to two, function `analyse-interaction` (see Listing 12) is called for analyzing the behavior of the use case interaction. If this function returns a non-empty sequence, then the existence of this bad smell is confirmed.

```
1  declare function local:Excessive-Alternation($model as node())
2  {
3          for $x in $model//IntegratedModel/UseCase
4                  return if (count(local:extPoints($x)) >= 3)
5                      then local:analyze-interaction($model,$x)
6              else ()
7  } ;
```

Listing 10: XQuery function to detect the bad smell Multiple Personality

```
1  declare function local:extPoints($a as node())
2  {
3          let $points := for $y in $a/ExtensionPoint
4             return data($y/@name)
5          return $points
6  } ;
```

Listing 11: XQuery function to count the number Extension Points in the Use Case

```
1   declare function local:analyze-interaction($model as node(), $uc as node())
2   {
3           for $x in $uc/Interaction/Fragment/MessageOccurance
4               let $val :=  count($x/Message)
5               return if (xs:integer(data($x/@order)) = 1 and ($val > 2) and
6                     (max(hr:value-union(hr:value-union(local:msg-occurance($x),
7                                                  local:cf($x)),local:use($x)))=2))
8                         then local:switch-pattern($model, $uc)
9                else ()
10  } ;
```

Listing 12: XQuery function to Analyze the Interaction Behavior of the Use Case

Listing 12 shows the function `analyse-interaction`. Interaction of a use case describes its dynamic behavior. Each interaction is composed of a number of lifelines and fragments. A fragment can be one of the three types acceptable by UML standards: *Message Occurance, Combined Fragment* and *Interaction Use*. In order to ensure that the interaction behavior is similar to that of a switch pattern, we check the preamble, body

and post sections of the interaction fragments. The functions `msg-occurance, cf and use` called in lines 6 and 7 returns the sequence of the fragments within the interaction of the use case. The function `switch-pattern` (see Listing 16) is called when the interaction has only two fragments and the preamble is composed of a sequence of messages (line 4) that are more than two. Listing 13, Listing 14, and Listing 15 show the functions `msg-occurance`, `cf` and `use` respectively.

```
1  declare function local:msg-occurance($a as node())
2  {
3          let $sequence := for $y in $a/Interaction/Fragments/MessageOccurance
4             return xs:integer(data($y/@order))
5          return $sequence
6  } ;
```

Listing 13: XQuery function to sequence the Message Occurance Fragments in the Interaction

```
1  declare function local:cf($a as node())
2  {
3          let $sequence := for $y in $a/Interaction/Fragments/CombinedFragments
4             return xs:integer(data($y/@order))
5          return $sequence
6  } ;
```

Listing 14: XQuery function to sequence the Combined Fragments in the Interaction

```
1  declare function local:use($a as node())
2  {
3          let $sequence := for $y in $a/Interaction/Fragments/InteractionUse
4             return xs:integer(data($y/@order))
5          return $sequence
6  } ;
```

Listing 15: XQuery function to sequence the Interaction Use Fragments in the Interaction

Listing 16 shows the function `switch-pattern`. A Combined Fragment with an “alt” interaction operator and more than two operands indicates that the use case spends more time switching between extension use cases and is a candidate for refactoring.

```
1  declare function local:switch-pattern($model as node(), $uc as node())
2  {
3     for $x in $uc/Interaction/Fragment/CombinedFragment
4                    let $val :=  count($x/Operands)
5                    return if ($val > 2 and xs:integer(data($x/@order)) = 2 and
6                                        data($x/@interactionOperator) = 'alt')
7                        then data($uc/@name)
8                      else ()
9  };
```

Listing 16: XQuery function to detect switch pattern

A4.4 Undue Familiarity

Listing 17 shows the XQuery function to detect instances of the bad smell *Undue Familiarity*. The function `Undue-Familiar` is parameterized with the source model $model. Using a *“for loop”*, each association in the integrated model is bound to the variable $x one after another in line 3. The function `isBidirectional` called in line 4 returns a value one if the association end of the particular association is both owned and navigable. The function `isBidirectional` is shown in Listing 18. If both ends of the association share ownership and are navigable, function `analyze-association` is called for behavioral analysis of the association.

```
1   declare function local:Undue-Familiar($model as node())
2   {
3           for $x in $model//IntegratedModel/Association
4                   return if (count(local:isBidirectional($x)) = 2 and
5                                                   (empty($x/@aggregationKind)))
6                       then local:analyze-association($model,$x)
7               else ()
8   } ;
```

Listing 17: XQuery function to detect the bad smell Undue Familiarity

```
1   declare function local:isBidirectional($a as node())
2   {
3           let $sequence := for $y in $a/AssociationEnd
4           return if (data($y/@isOwner) = 'true' and data($y/@isNavigable) = 'true')
5                   then data(1)
6           else ()
7           return $sequence
8   } ;
```

Listing 18: XQuery function to check ownership and navigability of an Association End

Listing 18 shows the function `analyze-association`. The condition for the if statement in line 4 first finds an intersection set between a two sequences, one composed of the two classes involved in the association relationship (result of the function call *class-list* shown in Listing 20) and the other composed of all lifelines for a given use case (result of the function call `lifelines` shown in Listing 5). If the size of this intersection is equal to two, it is safe to say that both classes participate in the interaction of this use case. Hence, the function `msg-frequency` is called in line 6 to evaluate the communication frequency between the classes within the selected use case interaction.

```
1  declare function local:analyze-association($model as node(), $Assoc as node())
2  {
3       for $x in $model//IntegratedModel/UseCase
4        return if (count(hr:value-intersect(local:class-list($model, $Assoc),
5                                                    local:lifelines($x))) = 2)
6                         then local:msg-frequency($model, $x, $Assoc)
7        else ()
8  } ;
```

Listing 19: XQuery function to Analyze the Association Relationship

Association Ends for an association include the reference id (type) of the class it associates with. In order for the function `analyze-association` to check whether these ends are participating in the lifelines of a use case, we needed to resolve its name from its reference. Function `resolve-class` called in line 4 of Listing 20 is implemented to carry out this purpose (see Listing 21).

```
1  declare function local:class-list($model as node(), $a as node())
2  {
3            let $sequence := for $y in $a/AssociationEnd
4                return local:resolve-class($model, $y)
5            return $sequence
6  } ;
```

Listing 20: XQuery function to return the list of classes the Association is in between

```
1  declare function local:resolve-class($model as node(), $type as node())
2  {
3            for $x in $model//IntegratedModel/Class
4                     return if (data($x/@id) = data($type/@type))
5                          then data($x/@name)
6                 else ()
7  } ;
```

Listing 21: XQuery function to resolve the name of a class given its id

A4.5 Spider's Web

Listing 22 shows the XQuery function to detect instances of the bad smell *Spider's Web*. The function `Spider-Web` is parameterized with the source model $model. Using a *"for loop"*, each actor in the integrated model is bound to the variable $x one after another in line 3. The function `NACU` called in line 4 returns the Number of Actors per Use Case value and the function `NAM` returns the total number of Use Cases in the model. The codes for functions `NACU` and `NUM` are not shown as they are simple counting functions. As in line 4, if the number of use cases associated with an actor is more than 30% of the total number of use cases then the existence of Spider's Web model smell is suspected and the function `actor-uc` (see Listing 23) is called to analyze the relationship between the actor and all its use cases.

```
1  declare function local:Spider-Web($model as node())
2  {
3          for $x in $model//IntegratedModel/Actor
4                  return if (count(local:NACU($model, $x))>0.30*(local:NUM($model)))
5                      then local:actor-uc($x,$model)
6              else ()
7  };
```

Listing 22: XQuery function to detect the bad smell Spider's Web

The `actor-uc` function shown in Listing 23 iterates through all the use cases associated with an actor (line 3-4) and checks whether the signature of each use case is similar or different. The function `signature` (see Listing 23) calculates the signature of each use case and the function `spider` (see Listing 24) confirms the existence of the model smell.

```
1  declare function local:actor-uc($a as node(), $iModel as node())
2  {
3                  for $y in $iModel//IntegratedModel/UseCase
4                     return if (data($a/@name)=data($y/@actor-ref))
5                         then local:spider($a, count(hr:value-intersect(local:signature($y,$iModel))
6                     else ()
7  };
```

Listing 23: XQuery function to analyze all use cases associated with an actor

```
1  declare function local:spider($actor as node(),$val as xs:integer)
2  {
3     let $v := $val
4            return
5            if ($v > 2)
6               then data($actor/@name)
7  };
```

Listing 24: XQuery function to check if the more than two signatures are different

The `signature` function shown in Listing 25 iterates through all the lifelines within the use case (line 3) and adds it to the signature sequence. If ant lifeline is a child class, then the `check-parent` (see Listing 26) function is invoked to add the parent to the signature. Finally, the returned sequence represents the signature of a use case.

```
1  declare function local:signature($a as node(), $model as node())
2  {
3            let $sequence := for $y in $a/Interaction/Lifeline
4               return local:check-parent($y,$model)
5            return $sequence
6  };
```

Listing 25: XQuery function to compute the signature of a use case

The `check-parent` function shown in Listing 26 is a simple function that returns the parent class of a child class if the class is part of an inheritance hierarchy.

```
1  declare function local:check-parent($x as node(),$model as node())
2  {
3          for $y in $model//IntegratedModel/Class
4             return if (data($y/@name) = data($x/@name))
5                          then data($y/SuperClass/@name)
6                   else data($x/@name)
7  };
```

Listing 26: XQuery function to return the parent of a child class

A4.6 Specters'

Listing 27 shows the XQuery function to detect instances of the bad smell *Specters'*. The function `Specters` is parameterized with the source model $model. Using a *"for loop"*, each class in the integrated model is bound to the variable $x one after another in line 3. The function `num-attr` called in line 4 returns the Number of Attributes in a class and if this value is equal to zero, the function `analyze-class` (see Listing 28) is called to check the behavior of the class within the use cases it is included in. The codes for functions `num-attr` is not shown as they are simple counting functions.

```
1  declare function local:Specters($model as node())
2  {
3          for $x in $model//IntegratedModel/Class
4                  return if (local:num-attr($x) = 0)
5                       then local:analyze-class($model,$x)
6             else ()
7  };
```

Listing 27: XQuery function to detect the bad smell Specters'

The `analyze-class` function shown in Listing 28 iterates through all the use cases within the Integrated Model which are *"inclusion"* use cases and checks whether the no-attribute class found is part of any one of them. If found, the behavior of this class (or lifeline) within that inclusion use case is analyzed in function `analyze-lifeline` (see Listing 29).

```
1  declare function local:analyze-class($a as node(), $x as node())
2  {
3          for $y in $a//IntegratedModel/UseCase/includes
4                  return if (hr:value-intersect(data($x/@name),local:lifelines($y)))
5                          then local:analyze-lifeline($y,$x)
6                  else ()
7  };
```

Listing 28: XQuery function to analyze no-attribute classes

The `analyze-lifeline` function shown in Listing 29 checks the event ends throughout the lifecycle of the lifeline within the use case. If all the end events are of type *"send"* (identified through the `recv-Events` function), then the class (or lifeline) is identified as a Specter class and needs to be refactored.

```
1  declare function local:analyze-lifeline($a as node(), $x as node())
2  {
3     for $y in $a/Lifeline
4       return if (data($y/@name) = data($x/@name) and (count(local:recv-Events($y))=0))
5             then data($y/@name)
6       else ()
7  };
```

Listing 29: XQuery function to analyze the no-attribute lifeline behavior in a use case

The `rec-Events` function shown in Listing 30 is a simple function to check whether the end type of an event over the lifeline is of type *"receive"* or not.

```
1  declare function local:recv-Events($a as node())
2  {
3          for $y in $a/End
4             return if (data($y/@endType)="receiveEvent")
5                         then     data($y/@endType)
6          else ()
7  };
```

Listing 30: XQuery function to identify the number of receive events on a lifeline

A4.7 Ripple Effect

Listing 31 shows the XQuery function to detect instances of the bad smell *Ripple Effect.* The function `Ripple-Effect` is parameterized with the source model $model. Using a *"for loop"*, each use case in the integrated model is bound to the variable $x one after another in line 3. The function `impact-factor` called in line 4 returns the value of the impact factor metric proposed in this work in Section 5.4.8.

```
1  declare function local:Ripple-Effect($model as node())
2  {
3          for $x in $model//IntegratedModel/UseCase
4                 return local:impact-factor($x, $model)
5  };
```

Listing 31: XQuery function to detect the bad smell Ripple Effect

The function `impact-factor` shown in Listing 32 iterates through all the lifelines of a use case to calculate the metric *NOEC* (Number of external connections) for each lifeline. If this value is found to me more than 30% of the total number of classes (from the `NCM` function) then the use case is suspected of including a lifeline (or a class) that exhibits the ripple effect model smell. The function `NOEC` is shown in Listing 33 and the function code for `NCM` is not included as it is a simple counting function.

```
1  declare function local:impact-factor($a as node(), $model as node())
2  {
3          for $y in $a/Lifeline
4                  return if (sum(local:NOEC($y,$model))=(0.3*(local:NCM($model))))
5                          then data($a/@name)
6                  else ()
7  };
```

Listing 32: XQuery function to calculate the impact factor

The function `NOEC` shown in Listing 33 calculates the NOEC metric which is the difference between the number of associations of a class (*NASC*) and the Number of Internal Connection (*NOIC*) as evident from Line 5 of the function code. As earlier, the function code for NASC is not included as it is a simple counting function.

```
1  declare function local:NOEC($x as node(), $iModel as node())
2  {
3      for $y in $iModel//IntegratedModel/Class
4          return if (data($y/@name) = data($x/@name))
5                  then (local:NASC($y,$iModel) - local:NOIC($x,$iModel))
6          else ()
7  };
```

Listing 33: XQuery function to calculate the NOEC metric

Listing 34 shows the function code for `NOIC`. The function iterates through all the use cases in the integrated model and identifies all sequence diagrams of which the selected lifeline is part of. This is done using the built-in function in XQuery called `is-value-in-sequence` and using the `lifelines` function to return all the lifelines participating in any use case (see Listing 5) at Line 4. If found, the number of distinct lifelines it interacts it with in the use case is counted (using the built in `count` function and the `distinct` function). The interacting lifelines are obtained using the function participating-ends (see Listing 35).

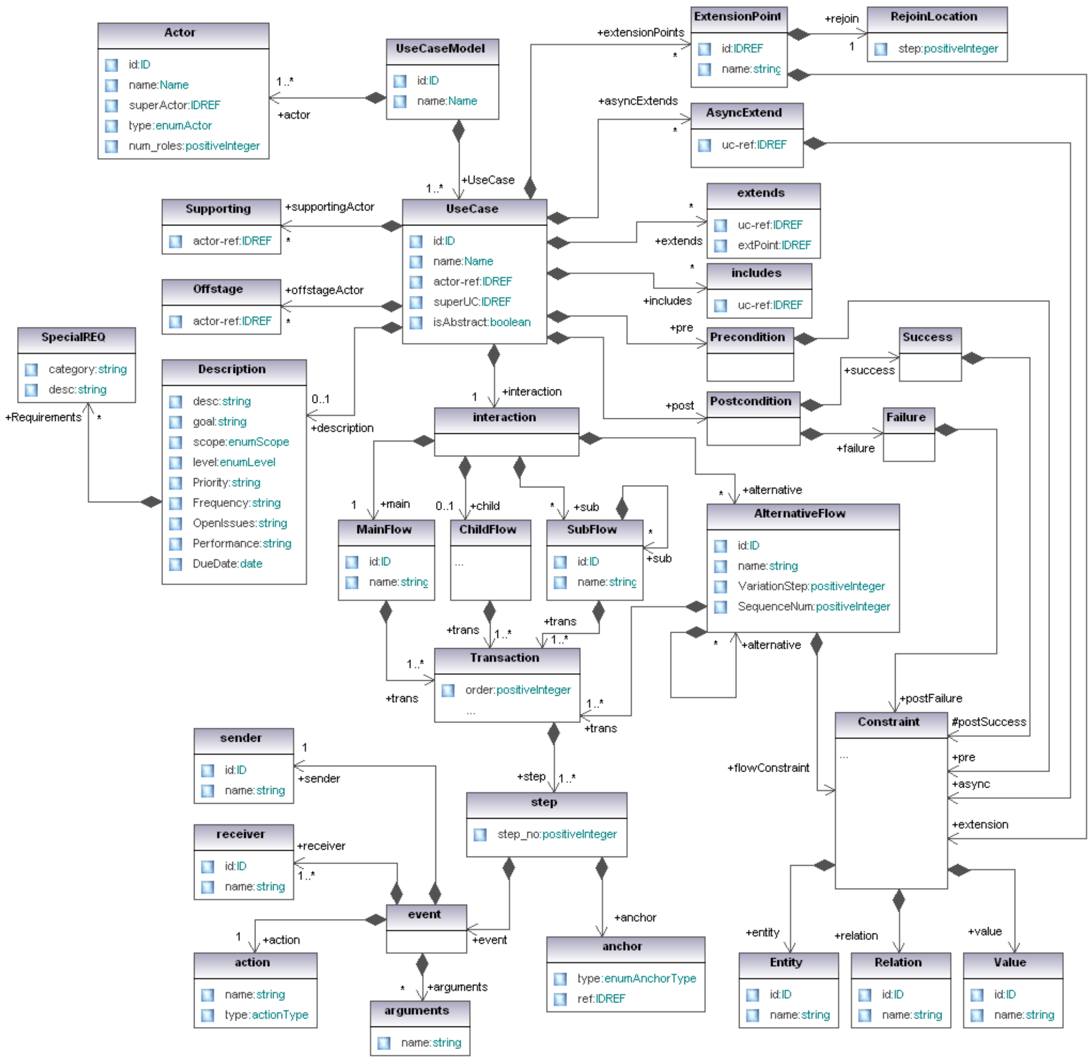
```
1  declare function local:NOIC($x as node(), $iModel as node())
2  {
3            for $y in $iModel//IntegratedModel/UseCase
4                      return if (is-value-in-sequence(data($x/@name),local:lifelines($y))
5                                then (count(distinct-values(local:particpating-ends($y,$x))))
6                      else ()
7  } ;
```

Listing 34: XQuery function to calculate the NOIC metric

```
1  declare function local:particpating-ends($a as node(), $x as node())
2  {
3        let $sequence := for $y in $a/Interaction/Message
4                      return if (data($y/@sender) = data($x/@name) or data($y/@reciever) =
5                                                                      data($x/@name))
6                      then data($y/@name)
7        return $sequence
8  } ;
```

Listing 35: XQuery function to identify the interacting lifelines

Appendix 5: XMI Schema for Extended Use Case Metamodel




```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
                              attributeFormDefault="unqualified">
     <xs:element name="UseCaseModel">
          <xs:annotation>
               <xs:documentation>Schema for Extended Use Case Metamodel
               </xs:documentation>
          </xs:annotation>
          <xs:complexType>
               <xs:sequence>
                    <xs:element name="UseCase" type="UseCase"
                                        maxOccurs="unbounded"/>
```

```
                        <xs:element name="Actor" type="Actor"
                                                maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="name" type="xs:Name"/>
                <xs:attribute name="id" type="xs:ID" use="required"/>
        </xs:complexType>
</xs:element>
<xs:complexType name="Actor">
        <xs:attribute name="id" type="xs:ID" use="required"/>
        <xs:attribute name="name" type="xs:Name" use="required"/>
        <xs:attribute name="superActor" type="xs:IDREF" use="optional"/>
        <xs:attribute name="type" type="enumActor" use="required"/>
        <xs:attribute name="num_roles" type="xs:positiveInteger" use="optional"/>
</xs:complexType>
<xs:complexType name="UseCase">
        <xs:sequence>
                <xs:element name="Supporting" minOccurs="0"/>
                <xs:element name="Offstage" minOccurs="0"/>
                <xs:element name="Description" minOccurs="0"/>
                <xs:element name="interaction"/>
                <xs:element name="includes" type="includes" minOccurs="0"
                                                maxOccurs="unbounded"/>
                <xs:element name="extends" type="extends" minOccurs="0"
                                                maxOccurs="unbounded"/>
                <xs:element name="AsyncExtend" minOccurs="0"
                                                maxOccurs="unbounded"/>
                <xs:element name="ExtensionPoint" minOccurs="0"
                                        maxOccurs="unbounded"/>
                <xs:element name="Precondition"/>
                <xs:element name="Postcondition"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:ID" use="required"/>
        <xs:attribute name="name" type="xs:Name" use="optional"/>
        <xs:attribute name="actor-ref" type="xs:IDREF" use="required"/>
        <xs:attribute name="superUC" type="xs:IDREF" use="optional"/>
        <xs:attribute name="isAbstract" type="xs:boolean" use="optional"/>
</xs:complexType>
<xs:complexType name="Precondition">
        <xs:sequence>
                <xs:element name="Constraint" maxOccurs="unbounded"/>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="Postcondition">
        <xs:sequence>
                <xs:element name="Success">
                        <xs:complexType>
                                <xs:sequence>
                                        <xs:element name="Constraint"
                                        maxOccurs="unbounded"/>
                                </xs:sequence>
                        </xs:complexType>
                </xs:element>
                <xs:element name="Failure" maxOccurs="unbounded">
                        <xs:complexType>
                                <xs:sequence>
                                        <xs:element name="Constraint"
```

```
                                        maxOccurs="unbounded"/>
                                </xs:sequence>
                            </xs:complexType>
                        </xs:element>
                    </xs:sequence>
                </xs:complexType>
                <xs:complexType name="ExtensionPoint">
                    <xs:sequence>
                        <xs:element name="Constraint"/>
                        <xs:element name="RejoinLocation"/>
                    </xs:sequence>
                    <xs:attribute name="id" type="xs:ID" use="required"/>
                    <xs:attribute name="name" type="xs:string" use="required"/>
                    <xs:attribute name="upper" type="xs:integer" use="optional"/>
                    <xs:attribute name="lower" type="xs:integer" use="optional"/>
                </xs:complexType>
                <xs:complexType name="includes">
                    <xs:attribute name="uc-ref" type="xs:IDREF" use="required"/>
                </xs:complexType>
                <xs:complexType name="extends">
                    <xs:attribute name="uc-ref" type="xs:IDREF" use="required"/>
                    <xs:attribute name="extPoint" type="xs:IDREF" use="required"/>
                </xs:complexType>
                <xs:complexType name="interaction">
                    <xs:choice>
                        <xs:element name="MainFlow"/>
                        <xs:element name="ChildFlow" minOccurs="0"/>
                        <xs:element name="SubFlow" minOccurs="0"
                                    maxOccurs="unbounded">
                            <xs:complexType/>
                        </xs:element>
                        <xs:element name="AlternativeFlow" minOccurs="0"
                                            maxOccurs="unbounded">
                            <xs:complexType/>
                        </xs:element>
                    </xs:choice>
                </xs:complexType>
                <xs:complexType name="MainFlow">
                    <xs:sequence>
                        <xs:element name="Transaction" maxOccurs="unbounded"/>
                    </xs:sequence>
                    <xs:attribute name="id" type="xs:ID" use="required"/>
                    <xs:attribute name="name" type="xs:string"/>
                </xs:complexType>
                <xs:complexType name="SubFlow">
                    <xs:sequence>
                        <xs:element name="Transaction" maxOccurs="unbounded"/>
                        <xs:element name="SubFlow" minOccurs="0"
                                            maxOccurs="unbounded"/>
                        <xs:element name="AlternateFlow" minOccurs="0"
                                            maxOccurs="unbounded"/>
                    </xs:sequence>
                    <xs:attribute name="id" type="xs:ID" use="required"/>
                    <xs:attribute name="name" type="xs:string" use="optional"/>
                </xs:complexType>
                <xs:complexType name="AlternateFlow">
```

```
        <xs:sequence>
            <xs:element name="Constraint"/>
            <xs:element name="Transaction" maxOccurs="unbounded"/>
            <xs:element name="SubFlow" minOccurs="0"
                                            maxOccurs="unbounded"/>
            <xs:element name="AlternateFlow" minOccurs="0"
                                            maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:ID" use="required"/>
        <xs:attribute name="name" type="xs:string" use="optional"/>
        <xs:attribute name="VariationStep" type="xs:positiveInteger" use="required"/>
        <xs:attribute name="SequenceNum" type="xs:positiveInteger" use="required"/>
    </xs:complexType>
    <xs:complexType name="Transaction">
        <xs:sequence>
            <xs:element name="Step" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="order" type="xs:positiveInteger" use="required"/>
    </xs:complexType>
    <xs:complexType name="Step">
        <xs:choice>
            <xs:element name="event" type="event"/>
            <xs:element name="anchor" type="anchor"/>
        </xs:choice>
        <xs:attribute name="step_no" type="xs:positiveInteger" use="required"/>
    </xs:complexType>
    <xs:complexType name="event">
        <xs:sequence>
            <xs:element name="sender" type="sender"/>
            <xs:element name="receiver" type="receiver"
                                            maxOccurs="unbounded"/>
            <xs:element name="arguments" type="arguments" minOccurs="0"
                                            maxOccurs="unbounded"/>
            <xs:element name="action" type="action"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="sender">
        <xs:attribute name="id" type="xs:ID" use="required"/>
        <xs:attribute name="name" type="xs:string" use="required"/>
    </xs:complexType>
    <xs:complexType name="receiver">
        <xs:attribute name="id" type="xs:ID" use="required"/>
        <xs:attribute name="name" type="xs:string" use="required"/>
    </xs:complexType>
    <xs:complexType name="action">
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="type" type="actionType" use="required"/>
    </xs:complexType>
    <xs:complexType name="arguments">
        <xs:attribute name="name" type="xs:string" use="required"/>
    </xs:complexType>
    <xs:complexType name="anchor">
        <xs:choice>
            <xs:element name="ExternalInclusion"/>
            <xs:element name="ExternalVariation"/>
            <xs:element name="InternalInclusion"/>
```

```
		</xs:choice>
	</xs:complexType>
	<xs:complexType name="ExternalInclusion">
		<xs:attribute name="uc-ref" type="xs:IDREF" use="required"/>
	</xs:complexType>
	<xs:complexType name="InternalInclusion">
		<xs:attribute name="ref" type="xs:IDREF" use="required"/>
	</xs:complexType>
	<xs:complexType name="ExternalVariation">
		<xs:attribute name="extPoint" type="xs:IDREF" use="required"/>
	</xs:complexType>
	<xs:complexType name="Constraint">
		<xs:sequence>
			<xs:element name="Entity"/>
			<xs:element name="Relation"/>
			<xs:element name="Value"/>
		</xs:sequence>
	</xs:complexType>
	<xs:complexType name="Entity">
		<xs:attribute name="id" type="xs:ID" use="required"/>
		<xs:attribute name="name" type="xs:string" use="required"/>
	</xs:complexType>
	<xs:complexType name="Relation">
		<xs:attribute name="id" type="xs:ID" use="required"/>
		<xs:attribute name="name" type="xs:string" use="required"/>
	</xs:complexType>
	<xs:complexType name="Value">
		<xs:attribute name="id" type="xs:ID" use="required"/>
		<xs:attribute name="name" type="xs:string" use="required"/>
	</xs:complexType>
	<xs:complexType name="RejoinLocation">
		<xs:attribute name="step" type="xs:positiveInteger" use="required"/>
	</xs:complexType>
	<xs:complexType name="Supporting">
		<xs:attribute name="actor-ref" type="xs:IDREF" use="required"/>
	</xs:complexType>
	<xs:complexType name="Offstage">
		<xs:attribute name="actor-ref" type="xs:IDREF" use="required"/>
	</xs:complexType>
	<xs:complexType name="Description">
		<xs:sequence>
			<xs:element name="SpecialREQ" minOccurs="0"
								maxOccurs="unbounded"/>
		</xs:sequence>
		<xs:attribute name="desc" type="xs:string" use="optional"/>
		<xs:attribute name="goal" type="xs:string" use="optional"/>
		<xs:attribute name="scope" type="enumScope" use="optional"/>
		<xs:attribute name="level" type="enumLevel" use="optional"/>
		<xs:attribute name="Priority" type="xs:string" use="optional"/>
		<xs:attribute name="Frequency" type="xs:string" use="optional"/>
		<xs:attribute name="OpenIssues" type="xs:string" use="optional"/>
		<xs:attribute name="Performance" type="xs:string" use="optional"/>
		<xs:attribute name="DueDate" type="xs:date" use="optional"/>
	</xs:complexType>
	<xs:complexType name="SpecialREQ">
		<xs:attribute name="category" type="xs:string"/>
```

```
        <xs:attribute name="desc" type="xs:string"/>
    </xs:complexType>
    <xs:complexType name="AsyncExtend">
        <xs:sequence>
            <xs:element name="Constraint"/>
        </xs:sequence>
        <xs:attribute name="uc-ref" type="xs:IDREF" use="required"/>
    </xs:complexType>
    <xs:simpleType name="actionType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="Request"/>
            <xs:enumeration value="DataValidate"/>
            <xs:enumeration value="Expletive"/>
            <xs:enumeration value="Response"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="enumActor">
        <xs:restriction base="xs:string">
            <xs:enumeration value="System"/>
            <xs:enumeration value="NetworkSystem"/>
            <xs:enumeration value="Human"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="enumScope">
        <xs:restriction base="xs:string">
            <xs:enumeration value="Organization"/>
            <xs:enumeration value="System"/>
            <xs:enumeration value="Component"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="enumLevel">
        <xs:restriction base="xs:string">
            <xs:enumeration value="Summary"/>
            <xs:enumeration value="PrimaryTask"/>
            <xs:enumeration value="Subfunction"/>
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Appendix 6: UCDesc User Manual

In this section, we present the user manual for the prototype tool UCDesc. The user manual describes how to author use case descriptions using the UCDesc tool.

A6.1 Creating a New Project

To create a new project, click File -> New Project. A dialog box requesting the name of the project followed by its destination location appears. A project is then created at the destination location. All UCDesc project files are saved with a *.ucdesc* extension. Figure A - 7 shows the New Project dialog. On successful creation of the project, the user can add, edit and delete actors and use cases to the system.

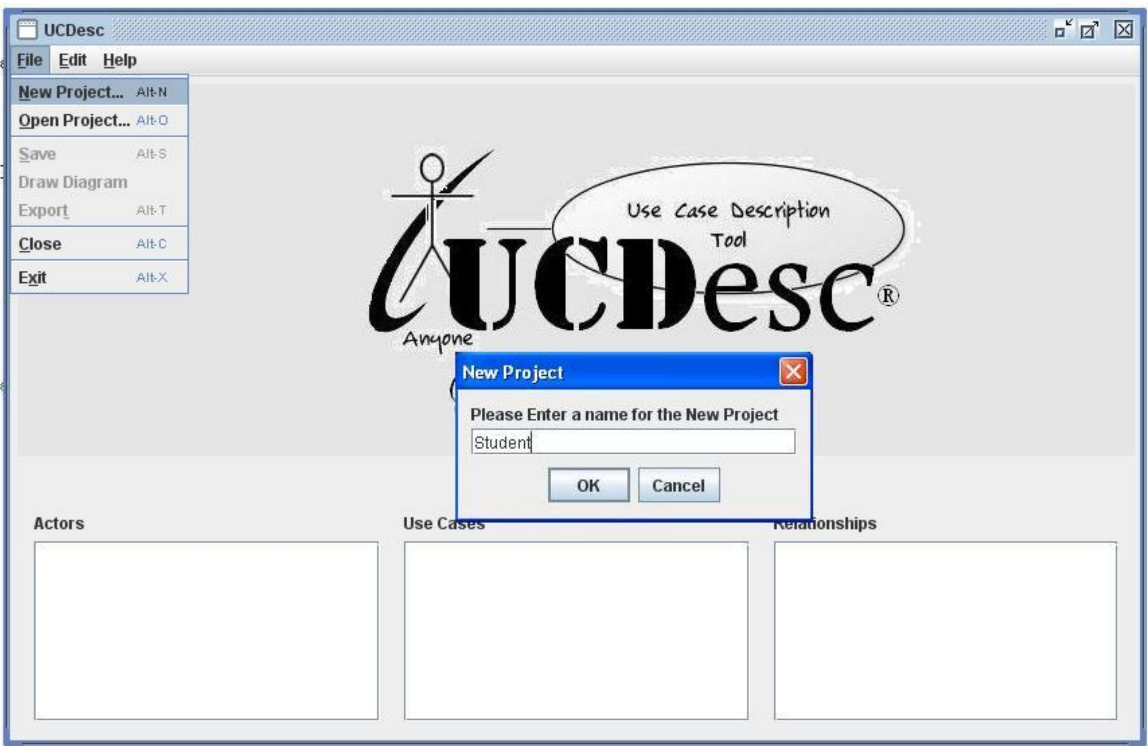


Figure A - 7 UCDesc: Create New Project

A6.2 Adding an Actor

Using the Edit Menu, users can add, edit and delete actors. Figure A - 8 shows the Add Actor dialog. In order to use an actor in a use case, it needs to be added before the adding the use case. The actor-id is automatically assigned by the system. If the actor is a specialization of an existing actor, the Parent Actor Drop-down list can be used to set the parent actor. If the parent actor is not yet added, the actor information can be edited later to set the parent actor.

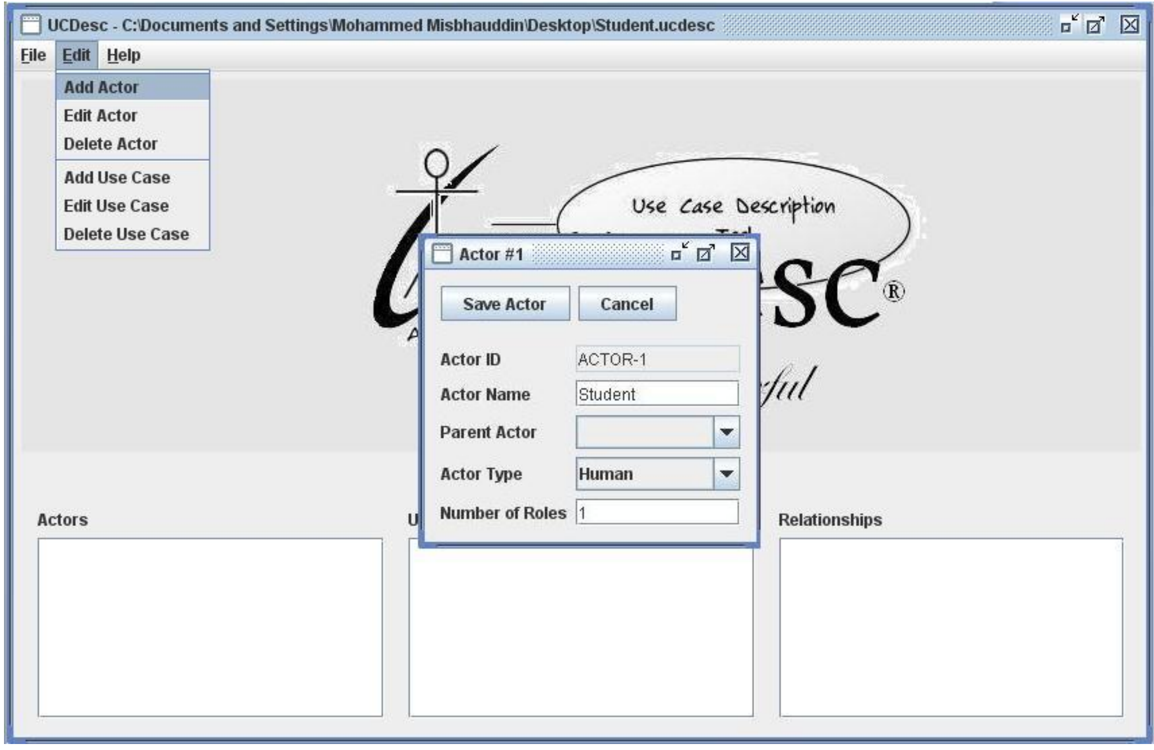


Figure A - 8 UCDesc: Add New Actor

Successfully added actor appears in the Actors panel in the UCDesc UI. Actor information can be edited by selecting the desired actor from the Actors panel and using the Edit Menu to either edit or delete the actor from the system.

A6.3 Adding a Use Case

Authoring a use case flows is the primary functionality of the UCDesc tool. Similar to the process of adding actors, users can add, edit and delete a use case from the Edit Menu. Figure A - 9 shows the Add Use Case dialog.

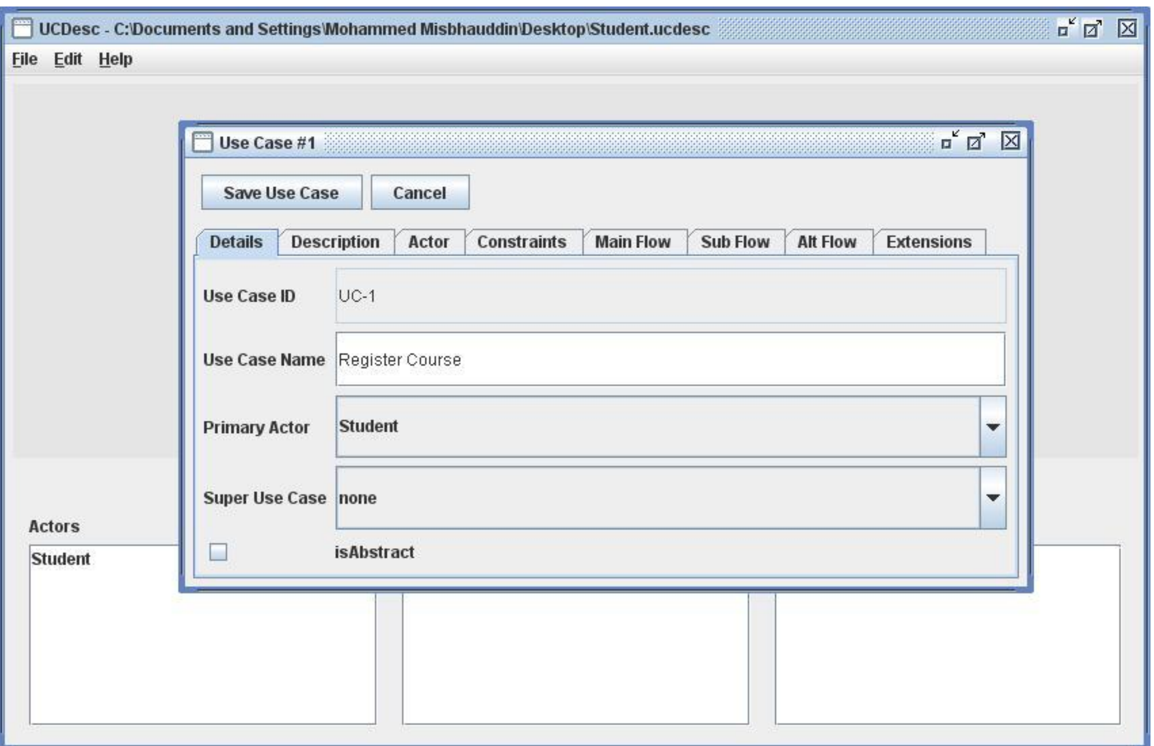


Figure A - 9 UCDesc: Add Use Case

Selecting the Add Use Case option from the Edit Menu opens the use case description editor. The editor is a tab-based input dialog. It includes eight tabs for adding the details for the use case, main flow description, sub flows, alternative flows and extension points. Use case ID is auto generated upon use case addition. The details required for each use is based on the extended use case metamodel proposed in this work.

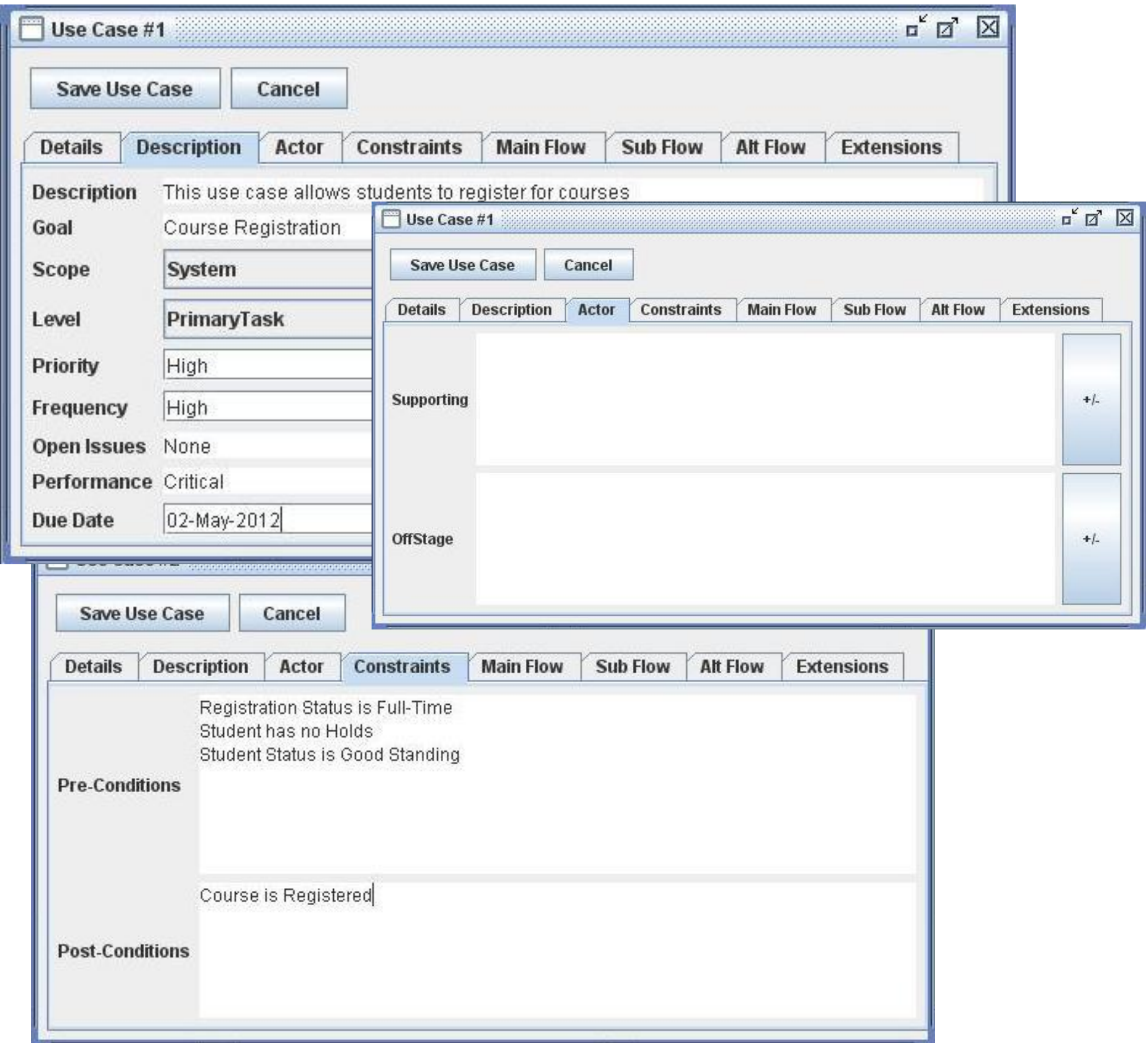


Figure A - 10 UCDesc: Use Case Description

The information required from the user for use case description, actor information and constraints are shown in Figure A - 10.

A6.4 Adding Flow of Steps

The Main Flow tab allows us to enter the steps required to fulfill the use case. Figure A - 11 shows the main flow editor tab. At any step, we can include an anchor for include and extend using the buttons provided in the Insert panel at the bottom of the dialog box.

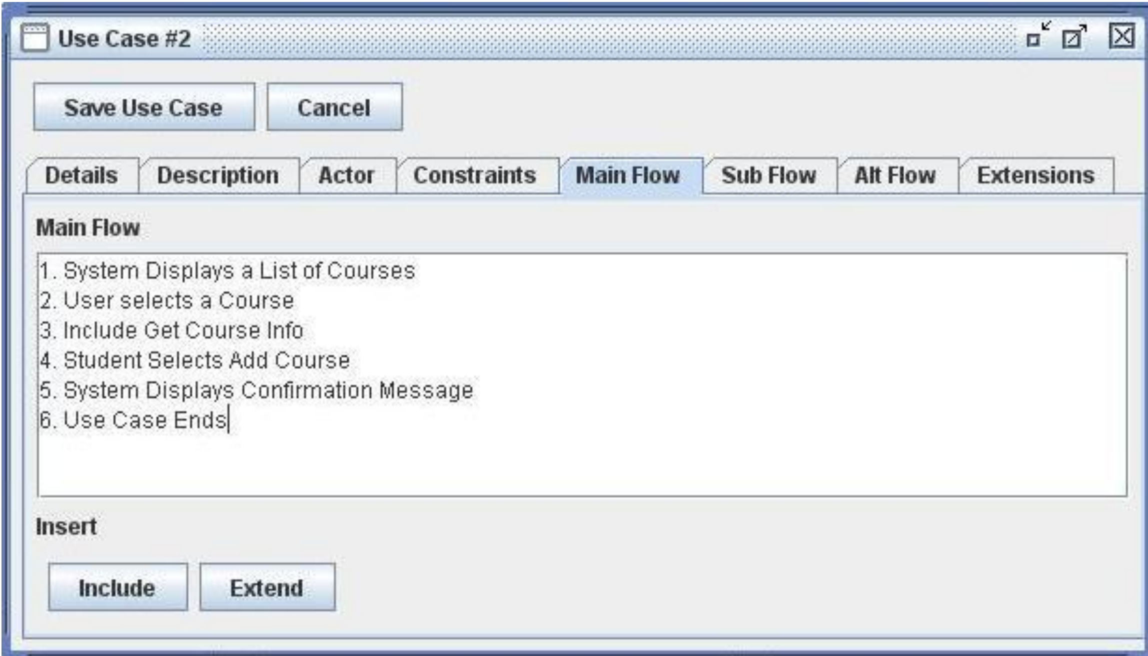


Figure A - 11 UCDesc: Main Flow Editor

If a user wants to include or extend a use case, the use case must added before the base use case is authored. Another constraint for extending a use case is to add an extension point before adding it to the main flow. Figure A - 12 shows information required for adding an extension point.

Adding Sub Flows and Alternative Flows to the Main Flow requires the user to indicate the step number in the flow description. If more than one sub or alternative flow is added to the same step, lower case alphabets are used to distinguish between them. Since the Sub Flow and Alternative Flow tab has the same basic architecture, we show only the Alternative Flow tab and the steps required to add an Alternative Flow in Figure A - 13.

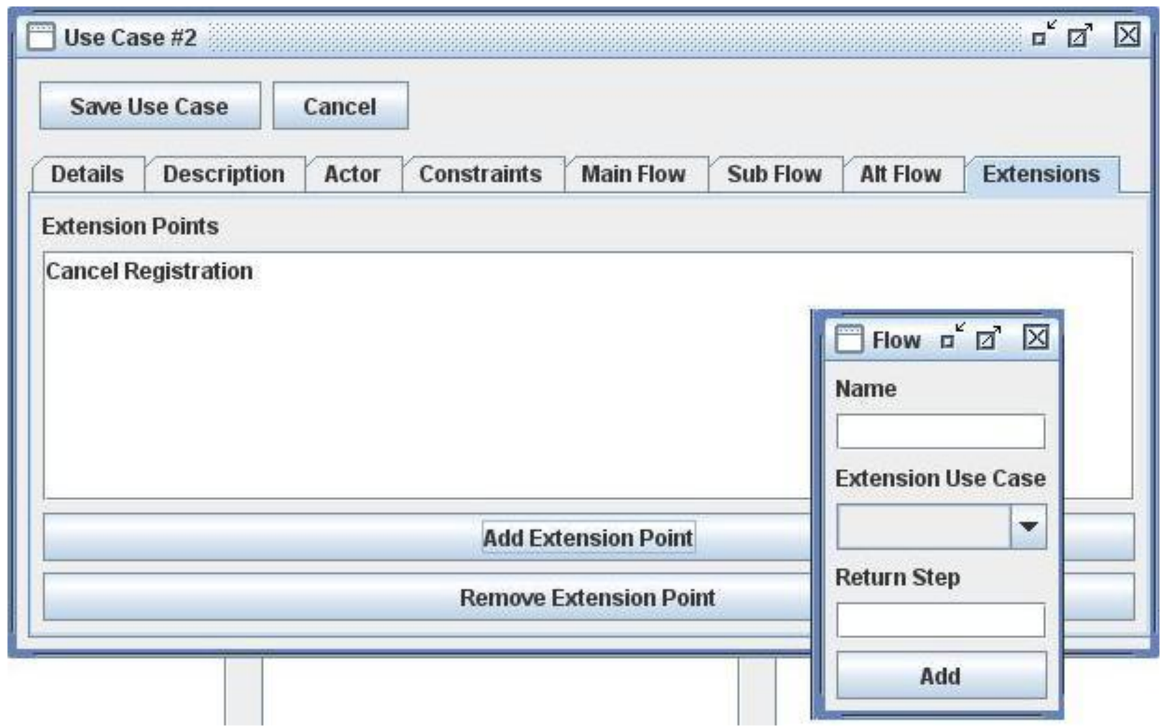


Figure A - 12 UCDesc: Extension Point

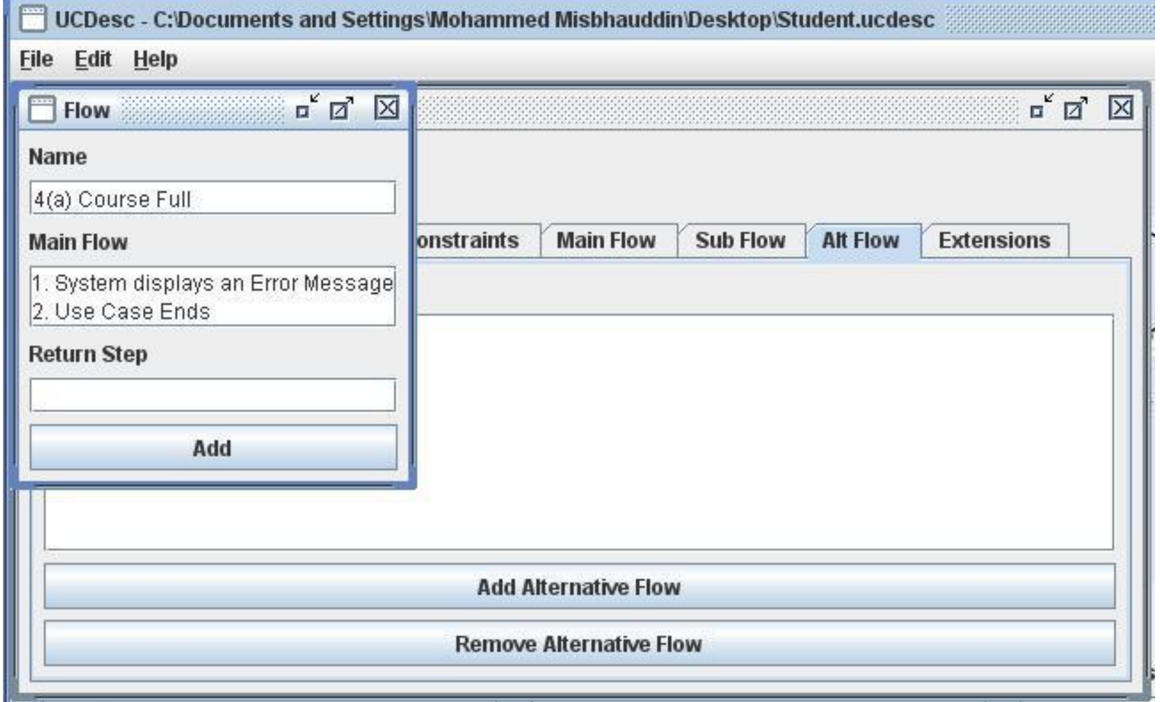


Figure A - 13 UCDesc: Alternative Flow

As use case relationships are added within the use case description (inclusion and extension), these are added simultaneously to the Relationship panel in the main UI as shown in Figure A - 14.

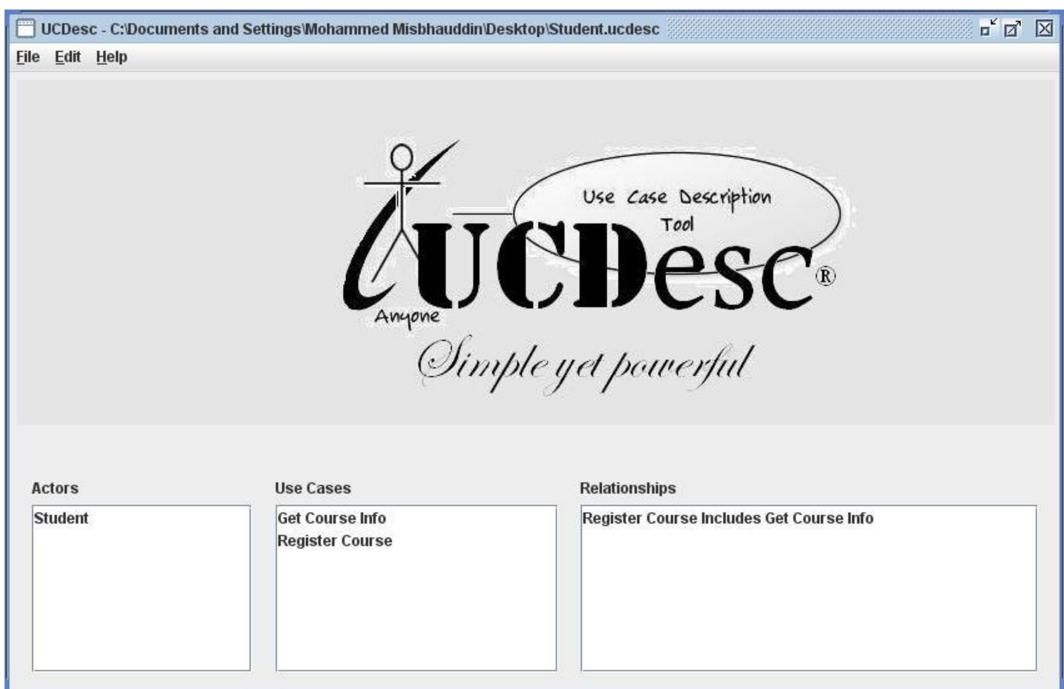


Figure A - 14 UCDesc: Main UI after use case addition

Once the project is completed, it can be exported to an XMI file using File -> Export option. A structural view of the use case diagram created can be viewed using File -> Draw Diagram option. Since UCDesc uses a web-based diagram generator, the resultant diagram opens in a web browser. Figure A - 15 shows the rendered use case diagram for the above-created project.

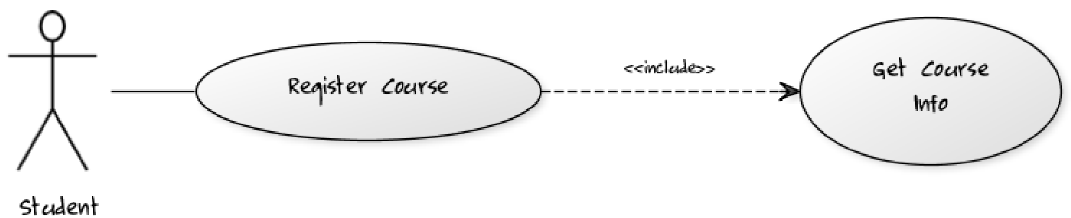


Figure A - 15 UCDesc: Use Case diagram rendered from yUML webserver

Appendix 7: XMI Schema for Integrated Metamodel

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2009 (http://www.altova.com) by KING FAHD UNIVERSITY OF
PETROLEUM & MINERALS (KING FAHD UNIVERSITY OF PETROLEUM & MINERALS) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
                                          attributeFormDefault="unqualified">
      <xs:element name="IntegratedModel">
         <xs:annotation>
            <xs:documentation>Schema for the Integrated Metamodel</xs:documentation>
         </xs:annotation>
         <xs:complexType>
            <xs:sequence>
               <xs:element name="UseCase" type="UseCase" maxOccurs="unbounded"/>
                  <xs:element name="Actor" type="Actor" maxOccurs="unbounded"/>
                  <xs:element name="Class" maxOccurs="unbounded">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element name="SuperClass" minOccurs="0"
                                             maxOccurs="unbounded">
                              <xs:complexType>
                                 <xs:attribute name="id" type="xs:IDREF"
                                                   use="required"/>
                                 <xs:attribute name="name" type="xs:string"
                                                   use="required"/>
                              </xs:complexType>
                           </xs:element>
                           <xs:element name="Message" minOccurs="0"
                                             maxOccurs="unbounded">
                           <xs:complexType>
                        <xs:sequence>
                        <xs:element name="Arguments">
                           <xs:complexType>
                              <xs:attribute name="id" type="xs:ID"
                                             use="required"/>
                              <xs:attribute name="name" type="xs:string"
                                             use="required"/>
                              <xs:attribute name="type" type="xs:string"
                                             use="optional"/>
                              <xs:attribute name="direction"
                                 type="enumDirectionKind" use="optional"/>
                              <xs:attribute name="default" type="xs:string"
                                             use="optional"/>
                           </xs:complexType>
                        </xs:element>
                     </xs:sequence>
                     <xs:attribute name="id" type="xs:ID" use="required"/>
                     <xs:attribute name="name" type="xs:string" use="required"/>
                     <xs:attribute name="retAttr" type="xs:string" use="optional"/>
                     <xs:attribute name="visibility" type="enumVisibilityKind"
                                                   use="optional"/>
```

```
            <xs:attribute name="isQuery" type="xs:boolean"
                                          use="optional"/>
                <xs:attribute name="isAbstract" type="xs:boolean"
                                          use="optional"/>
                <xs:attribute name="isStatic" type="xs:boolean"
                                          use="optional"/>
                <xs:attribute name="concurrency"
                                  type="enumConcurrencyKind"/>
              </xs:complexType>
             </xs:element>
            <xs:element name="Association" maxOccurs="unbounded">
             <xs:complexType>
              <xs:complexContent>
                <xs:extension base="Association">
                  <xs:sequence>
                    <xs:element name="AssociationEnd"
                                minOccurs="2" maxOccurs="2">
                    <xs:complexType>
                      <xs:attribute name="name" type="xs:string"
                                          use="required"/>
                       <xs:attribute name="type" type="xs:IDREF"
                                          use="required"/>
                       <xs:attribute name="lower" type="xs:string"
                                          use="optional"/>
                       <xs:attribute name="upper" type="xs:string"
                                          use="optional"/>
                       <xs:attribute name="isOwner"
                         type="xs:boolean"  use="optional"/>
                       <xs:attribute name="isNavigable"
                         type="xs:boolean" use="optional"/>
                    </xs:complexType>
                   </xs:element>
                   <xs:element name="AssociationClass">
                    <xs:complexType>
                      <xs:attribute name="id" type="xs:ID"
                                          use="required"/>
                      <xs:attribute name="idref" type="xs:IDREF"
                                          use="required"/>
                    </xs:complexType>
                   </xs:element>
                  </xs:sequence>
                </xs:extension>
              </xs:complexContent>
             </xs:complexType>
            </xs:element>
            <xs:element name="Property" type="Property"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:ID" use="required"/>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="isFinalSpecialization" type="xs:boolean"
                                          use="optional"/>
        <xs:attribute name="isAbstract" type="xs:boolean" use="optional"/>
        <xs:attribute name="isLeaf" type="xs:boolean" use="optional"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="Constraint" type="Constraint"/>
```

```
        </xs:sequence>
        <xs:attribute name="name" type="xs:Name"/>
        <xs:attribute name="id" type="xs:ID" use="required"/>
        </xs:complexType>
    </xs:element>
    <xs:complexType name="UseCase">
        <xs:sequence>
            <xs:element name="Supporting" minOccurs="0"/>
            <xs:element name="Offstage" minOccurs="0"/>
            <xs:element name="Include" type="Include" minOccurs="0"
                                                        maxOccurs="unbounded"/>
            <xs:element name="Extend" type="Extend" minOccurs="0"
                                                        maxOccurs="unbounded"/>
            <xs:element name="AsyncExtend" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="ExtensionPoint" minOccurs="0" maxOccurs="unbounded">
             <xs:complexType>
                  <xs:complexContent>
                      <xs:extension base="ExtPoint">
                          <xs:sequence>
                              <xs:element name="RejoinLocation"/>
                              <xs:element name="Constraint" type="Constraint"/>
                          </xs:sequence>
                      </xs:extension>
                  </xs:complexContent>
             </xs:complexType>
            </xs:element>
           <xs:element name="Precondition">
            <xs:complexType>
                  <xs:sequence>
                     <xs:element name="Constraint" type="Constraint" maxOccurs="unbounded"/>
                  </xs:sequence>
             </xs:complexType>
           </xs:element>
           <xs:element name="Postcondition">
              <xs:complexType>
                  <xs:sequence>
                        <xs:element name="Success">
                           <xs:complexType>
                                <xs:sequence>
                                    <xs:element name="Constraint" type="Constraint"
                                                            maxOccurs="unbounded"/>
                                </xs:sequence>
                           </xs:complexType>
                        </xs:element>
                        <xs:element name="Failure">
                           <xs:complexType>
                                <xs:sequence>
                                    <xs:element name="Constraint" type="Constraint"
                                                            maxOccurs="unbounded"/>
                                </xs:sequence>
                           </xs:complexType>
                         </xs:element>
                  </xs:sequence>
              </xs:complexType>
           </xs:element>
          <xs:element name="Interaction" type="Interaction"/>
```

```
 </xs:sequence>
 <xs:attribute name="id" type="xs:ID" use="required"/>
<xs:attribute name="name" type="xs:Name"/>
      <xs:attribute name="actor-ref" type="xs:IDREF" use="required"/>
      <xs:attribute name="superUC" type="xs:IDREF"/>
      <xs:attribute name="isAbstract" type="xs:boolean" use="optional"/>
  </xs:complexType>
  <xs:complexType name="ExtPoint">
      <xs:attribute name="id" type="xs:IDREF" use="required"/>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="lower" type="xs:integer" use="optional"/>
       <xs:attribute name="upper" type="xs:integer" use="optional"/>
  </xs:complexType>
  <xs:complexType name="Include">
      <xs:attribute name="uc-ref" type="xs:IDREF" use="required"/>
  </xs:complexType>
  <xs:complexType name="Extend">
       <xs:attribute name="uc-ref" type="xs:IDREF" use="required"/>
       <xs:attribute name="extPoint" type="xs:integer" use="required"/>
  </xs:complexType>
  <xs:complexType name="Actor">
       <xs:attribute name="id" type="xs:ID" use="required"/>
       <xs:attribute name="name" type="xs:Name" use="required"/>
       <xs:attribute name="type" type="enumType" use="required"/>
       <xs:attribute name="num_roles" type="xs:positiveInteger" use="optional"/>
       <xs:attribute name="superActor" type="xs:IDREF" use="optional"/>
  </xs:complexType>
  <xs:complexType name="Interaction">
       <xs:sequence>
           <xs:element name="Lifeline" type="Lifeline"/>
           <xs:element name="Message" type="Message"/>
           <xs:element name="Gate" minOccurs="0" maxOccurs="unbounded"/>
           <xs:element name="Fragment" type="Fragment" minOccurs="0"
                                                 maxOccurs="unbounded"/>
         </xs:sequence>
         <xs:attribute name="id" type="xs:ID" use="required"/>
         <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="Lifeline">
       <xs:sequence>
           <xs:element name="End" maxOccurs="unbounded">
             <xs:complexType>
               <xs:attribute name="endType" type="enumEvent" use="required"/>
               <xs:attribute name="event-ref" type="xs:IDREF" use="required"/>
            </xs:complexType>
           </xs:element>
           <xs:element name="StateInvariant" type="Constraint"/>
         </xs:sequence>
         <xs:attribute name="id" type="xs:IDREF" use="required"/>
         <xs:attribute name="name" type="xs:string" use="required"/>
         <xs:attribute name="class-ref" type="xs:IDREF" use="required"/>
  </xs:complexType>
  <xs:complexType name="Property">
         <xs:attribute name="id" type="xs:ID" use="required"/>
         <xs:attribute name="name" type="xs:string" use="required"/>
         <xs:attribute name="type" type="xs:string" use="required"/>
```

```
      <xs:attribute name="lower" type="xs:integer"/>
      <xs:attribute name="upper" type="xs:string"/>
      <xs:attribute name="default" type="xs:string"/>
    <xs:attribute name="isReadOnly" type="xs:boolean"/>
    <xs:attribute name="isDerived" type="xs:boolean"/>
    <xs:attribute name="visibility" type="enumVisibilityKind"/>
</xs:complexType>
<xs:complexType name="Association">
    <xs:attribute name="id" type="xs:ID" use="required"/>
    <xs:attribute name="name" type="xs:Name"/>
    <xs:attribute name="AggregationKind" type="enumAggregationKind"/>
</xs:complexType>
<xs:complexType name="MessageEnd">
     <xs:attribute name="name" type="xs:string"/>
     <xs:attribute name="id" type="xs:IDREF" use="required"/>
     <xs:attribute name="isGate" type="xs:boolean"/>
</xs:complexType>
<xs:complexType name="DataType">
     <xs:attribute name="id" type="xs:ID" use="required"/>
     <xs:attribute name="name" type="xs:string"/>
</xs:complexType>
<xs:complexType name="Message">
      <xs:attribute name="id" type="xs:ID" use="required"/>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="messageKind" type="enumMessageKind"/>
      <xs:attribute name="messageSort" type="enumMessageSort"/>
</xs:complexType>
<xs:complexType name="Fragment">
      <xs:choice>
         <xs:element name="SingleOperand">
              <xs:complexType>
                  <xs:sequence>
                          <xs:element name="InteractionOperand"
                                              type="InteractionOperand"/>
                               <xs:choice>
                                  <xs:element name="Opt"/>
                                  <xs:element name="Loop">
                                      <xs:complexType>
                                          <xs:attribute name="maxint" type="xs:integer"/>
                                          <xs:attribute name="minint" type="xs:integer"/>
                                      </xs:complexType>
                                  </xs:element>
                                  <xs:element name="Break"/>
                                  <xs:element name="Neg"/>
                                </xs:choice>
                  </xs:sequence>
               </xs:complexType>
           </xs:element>
           <xs:element name="MultiOperand">
                  <xs:complexType>
                       <xs:sequence>
                          <xs:element name="InteractionOperand"
                                  type="InteractionOperand" maxOccurs="unbounded"/>
                              <xs:choice>
                                  <xs:element name="Par"/>
                                  <xs:element name="Alt"/>
```

```
                                <xs:element name="Assert"/>
                                <xs:element name="Strict"/>
                                <xs:element name="Seq"/>
                            </xs:choice>
                        </xs:sequence>
                        <xs:attribute name="isStrict" type="xs:boolean" use="optional"/>
                    </xs:complexType>
                </xs:element>
                <xs:element name="InteractionUse">
                    <xs:complexType>
                        <xs:attribute name="interactionName" type="xs:IDREF"
                                                                    use="required"/>
                    </xs:complexType>
                </xs:element>
            </xs:choice>
            <xs:attribute name="id" type="xs:ID" use="required"/>
            <xs:attribute name="name" type="xs:string" use="required"/>
        </xs:complexType>
        <xs:complexType name="InteractionOperand">
            <xs:sequence>
                <xs:element name="InteractionFragment" type="Fragment"/>
                <xs:element name="Guard" type="Constraint" minOccurs="0"/>
            </xs:sequence>
        </xs:complexType>
        <xs:complexType name="Arguments">
            <xs:attribute name="id" type="xs:ID"/>
            <xs:attribute name="name" type="xs:Name"/>
            <xs:attribute name="direction" type="enumDirectionKind"/>
            <xs:attribute name="type" type="xs:IDREF"/>
            <xs:attribute name="default" type="xs:string"/>
        </xs:complexType>
        <xs:simpleType name="enumMessageSort">
            <xs:restriction base="xs:string">
                <xs:enumeration value="syncCall"/>
                <xs:enumeration value="asyncCall"/>
                <xs:enumeration value="createMessage"/>
                <xs:enumeration value="deleteMessage"/>
                <xs:enumeration value="reply"/>
            </xs:restriction>
        </xs:simpleType>
        <xs:simpleType name="enumMessageKind">
            <xs:restriction base="xs:string">
                <xs:enumeration value="complete"/>
                <xs:enumeration value="lost"/>
                <xs:enumeration value="found"/>
                <xs:enumeration value="unknown"/>
            </xs:restriction>
        </xs:simpleType>
        <xs:simpleType name="enumDirectionKind">
            <xs:restriction base="xs:string">
                <xs:enumeration value="in"/>
                <xs:enumeration value="out"/>
                <xs:enumeration value="inout"/>
                <xs:enumeration value="return"/>
            </xs:restriction>
        </xs:simpleType>
```

```xml
<xs:simpleType name="enumAggregationKind">
    <xs:restriction base="xs:string">
        <xs:enumeration value="none"/>
        <xs:enumeration value="shared"/>
        <xs:enumeration value="composite"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="enumVisibilityKind">
    <xs:restriction base="xs:string">
        <xs:enumeration value="public"/>
        <xs:enumeration value="private"/>
        <xs:enumeration value="protected"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="enumType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="System"/>
        <xs:enumeration value="NetworkSystem"/>
        <xs:enumeration value="Human"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="enumEvent">
    <xs:restriction base="xs:string">
        <xs:enumeration value="sendEvent"/>
        <xs:enumeration value="recieveEvent"/>
        <xs:enumeration value="fragment"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="enumConcurrencyKind">
    <xs:restriction base="xs:string">
        <xs:enumeration value="sequential"/>
        <xs:enumeration value="guarded"/>
        <xs:enumeration value="concurrent"/>
    </xs:restriction>
</xs:simpleType>
<xs:complexType name="Constraint">
    <xs:sequence>
        <xs:element name="Expression" maxOccurs="unbounded">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="Operator"
                                maxOccurs="unbounded"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>
</xs:schema>
```

Appendix 8: IntegraUML User Manual

In this section, we present the user manual for the prototype tool IntegraUML. The user manual describes how to user can create a project, import XMI files for the UML diagrams into the project, create and refactor the integrated model.

To create a new project, click File -> New Project. A dialog box requesting the name of the project followed by its destination location appears. When you click the 'Ok' button, a project folder is created at the Project Location with the Name of the project. New Project dialog box is shown in Figure A - 16.

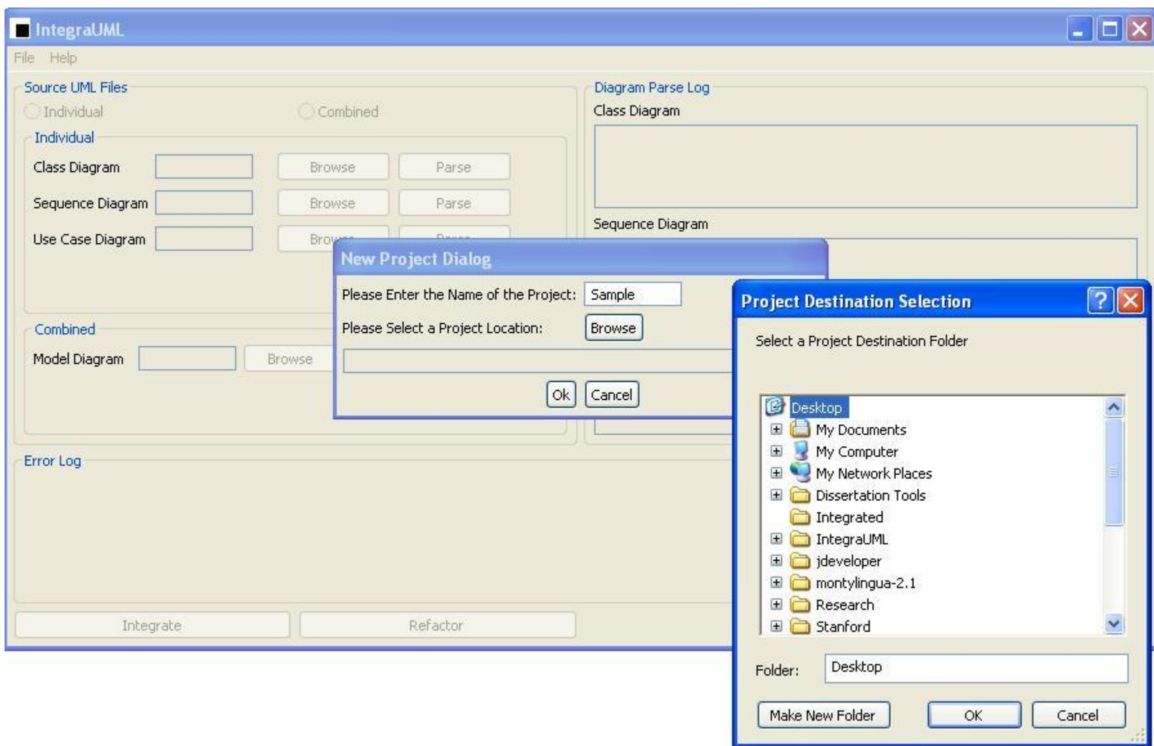


Figure A - 16 IntegrUML: New Project Dialog Box

If the selected project location already contains a folder of the same name, you will be prompted to rename the project or change the source location. To open an existing

project, choose the ‘File -> Open Project’ option from the menu. This will launch a project folder selector that will allow you to select and open a project. IntegraUML looks for a project file *(Data.IntegraUML)* within the project folder. This file identifies that the folder is an IntegraUML workspace. Failure to find this file in the selected folder will result in an error message. If the file is found, clicking the ‘Ok’ button or pressing the ‘Enter’ key opens the selected project.

When a project is created or opened, the user can browse and upload the model files. This can be done through the **Source UML Files** panel. IntegraUML provides users with two options when uploading source model files. Users can upload XMI files for the class diagram; sequence diagrams and use case diagram as individual files or combined as one XMI file. The browsing options are enabled based on the selection of an appropriate radio button at the top of the panel. Selecting one disables the other group as shown in Figure A - 17.

Figure A - 17 IntegraUML: Source UML File Selection

Once the selection is made, the users can upload respective model XMI files using the *Browse* button. Upon successful selection of the model files, the parse buttons beside the browse buttons is enabled. All model files must be parsed before integration. Parsing

starts the DOM API and generates a DOM tree for each UML model. The results of the parsing process is displayed in the **Diagram Parse Log** panel. Typical parse log information includes diagram version, tool exported from and statistical information like number of classes, number of interactions and so on as shown in Figure A - 18.

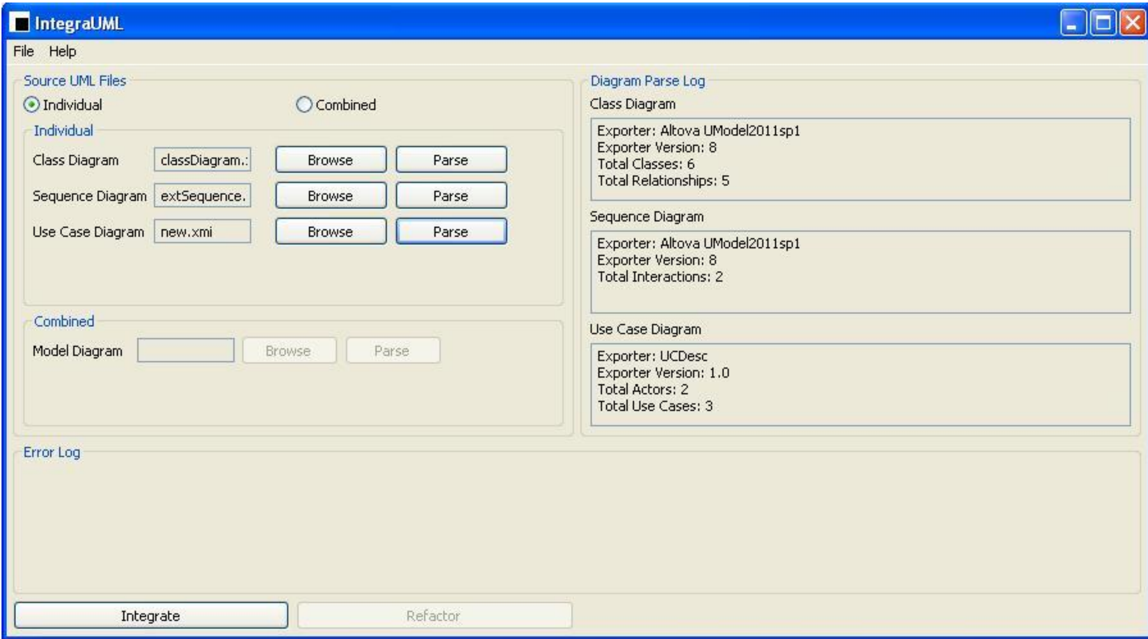


Figure A - 18 IntegraUML: UML Model Parsing Results

The 'Integrate' button at the bottom of the UI is activated once all uploaded model files are parsed successfully. The model integration process is started once the user presses the 'Integrate' button. Results of unsuccessful model integration are displayed in the Error Log as shown in Figure A - 19. The user is required to resolve these errors before proceeding with model integration.

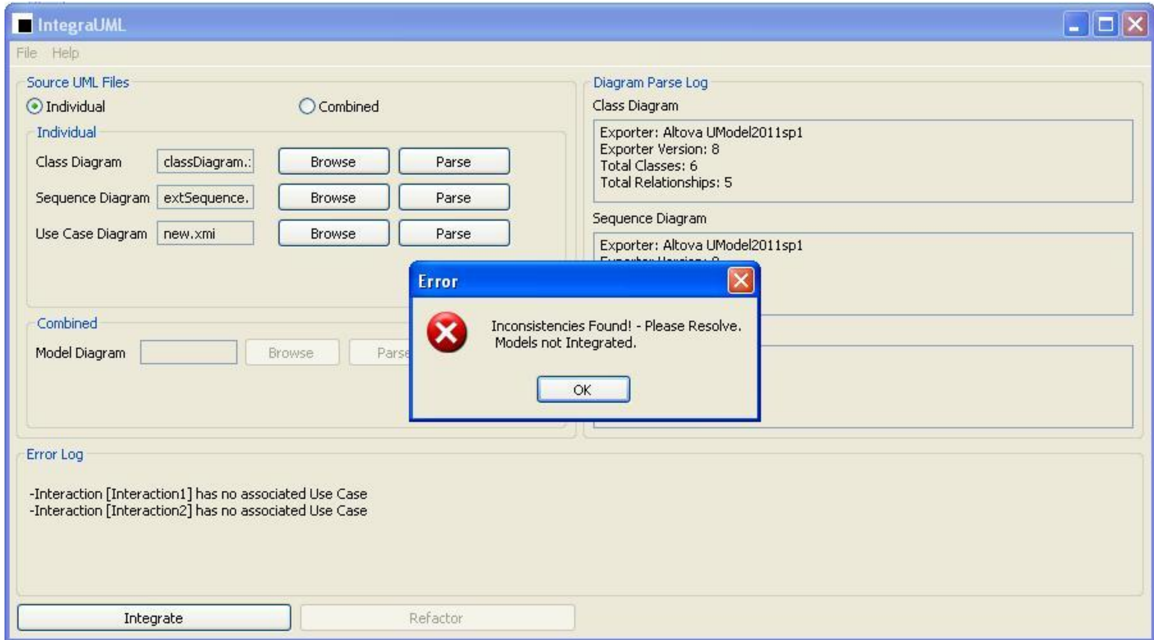


Figure A - 19 IntegraUML: Model Integration Error

An integrated model XMI file is created in the project's source location upon successful model integration and an appropriate feedback message dialog is show to the user. IntegraUML allows users to select and upload model files from any location in the user's computer. However, the user is required to save the project in order to copy these files to the project location. To save an open project, choose the 'File -> Save Project' option from the menu. This will copy and rename all the model files as shown in Figure A - 20.

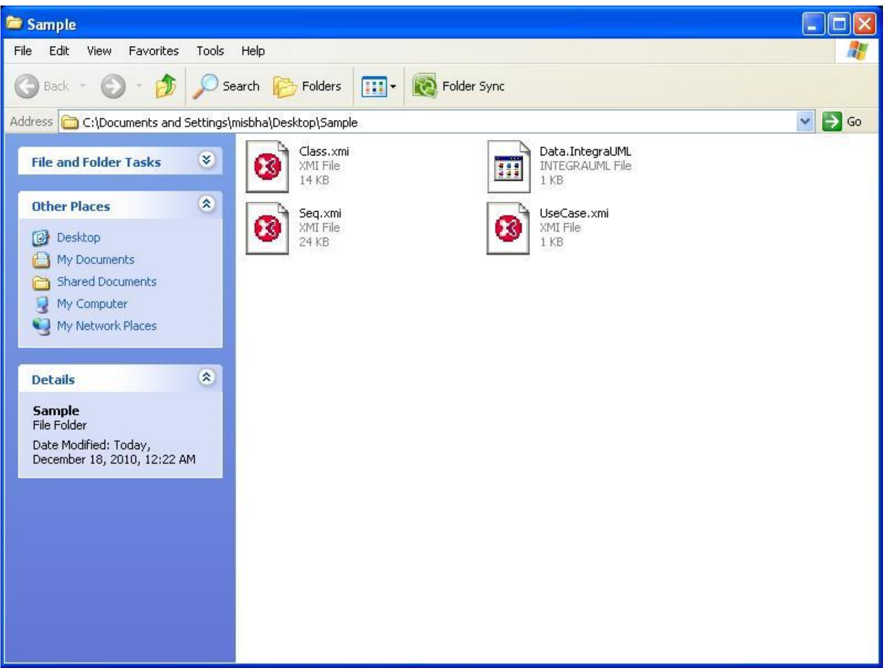


Figure A - 20 IntegraUML: File Structure

Appendix 9: UML Model Metrics

A9.1 Class Diagram Metrics

1. **DEPTH OF INHERITANCE TREE (DIT)**

   Level: Class-Level

   Description: This metric is useful for measuring the vertical hierarchy of an inheritance tree. The higher the value of DIT, the greater the chance of reuse becomes. However, a high value of DIT can cause program comprehension problem.

   Reference: **[62]**

2. **NUMBER OF CHILDREN (NOC)**

   Level: Class-Level

   Description: This metric counts the number of direct children of a class.

   Reference: **[62]**

3. **FAN-IN**

   Level: Class-Level

   Description: This metric counts the number of incoming association relations of a class. It measures the extent to which other classes use the class' provided services.

   Reference: **[290]**

4. FAN-OUT

Level: Class-Level

Description: This metric counts the number of outgoing association relations of a class. It measures the extent to which the class uses services provided by other classes.

Reference: **[290]**

5. NUMBER OF THE ASSOCIATIONS (NASM)

Level: Model-Level

Description: An association is a connection, or a link, between classes. This metric counts the number of associations in a class model. This metric is useful for estimating the scale of relationships between classes.

Reference: **[287]**

6. NUMBER OF THE AGGREGATIONS (NAGM)

Level: Model-Level

Description: An aggregation is a special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part. This metric counts the number of aggregations in a class model.

Reference: **[287]**

7. NUMBER OF THE ASSOCIATIONS LINKED TO A CLASS (NASC)

Level: Class-Level

Description: The number of associations including aggregations is counted. This metric is useful for estimating the static relationships between classes.

Reference: **[287]**

8. NUMBER OF THE ATTRIBUTES IN A CLASS UNWEIGHTED (NATC1)

Level: Class-Level

Description: This metric counts the number of attributes in a class. It does not apply a weighting scheme, meaning public, private and protected attributes are treated equal.

Reference: **[287]**

9. NUMBER OF THE ATTRIBUTES IN A CLASS WEIGHTED (NATC2)

Level: Class-Level

Description: This metric is a weighted version of NATC1. That is, it applies different weights to each metric depending on their visibility, i.e. 1.0 for public, 0.5 for protected and 0.0 for private attributes. This is more correct in a sense that the concept of encapsulation is more properly reflected in this weighting scheme.

Reference: **[287]**

10. NUMBER OF THE OPERATIONS IN A CLASS UNWEIGHTED (NOPC1)

Level: Class-Level

Description: This is an un-weighted metric that counts the number of operations in a class. Inheriting Operations in case of generalization relationship with other classes is also included in this measure.

Reference: **[287]**

11. NUMBER OF THE OPERATIONS IN A CLASS WEIGHTED (NOPC2)

Level: Class-Level

Description: This metric is same as NOPC1 except different weights are applied. The weights are similar to the one used in NATC2.

Reference: **[287]**

12. NUMBER OF THE CLASSES (NCM)

Level: Model-Level

Description: This metric counts the number of classes in a model. This metric is comparable to the traditional LOC (lines of code) or a more advanced McCabe's cyclomatic complexity (MVG) metric for estimating the size of a system [204]. Thus, in OOP this metric can be used to compare sizes of systems.

Reference: **[287]**

13. NUMBER OF THE INHERITANCE RELATIONS (NIM)

Level: Model-Level

Description: This metric counts the number of generalization relationships between classes existing in a model.

Reference: **[287]**

14. NUMBER OF THE SUPER CLASSES OF A CLASS (NSUPC)

Level: Class-Level

Description: This counts the direct parents of a class. In a single inheritance implementation like Java, the value of this metric is either 0 or 1, whereas under multiple inheritance schemes it is greater than or equal to 0.

Reference: **[287]**

15. NUMBER OF THE ELEMENTS IN THE TRANSITIVE CLOSURE OF THE SUBCLASSES OF A CLASS (NSUBC*)

Level: Class-Level

Description: This counts the transitive closure of the subclasses of a class, and it is potentially useful for predicting the classes who might be affected if changes occur in this class.

Reference: **[287]**

16. **NUMBER OF THE ELEMENTS IN THE TRANSITIVE CLOSURE OF THE SUPERCLASSES OF A CLASS (NSUPC*)**

Level: Class-Level

Description: This metric counts the transitive closure of the super classes of a class, and it is potentially useful for predicting the classes whose changes might affect this class.

Reference: **[287]**

A9.2 Sequence Diagram Metrics

1. **NUMBER OF MESSAGES SENT BY THE INSTANTIATED OBJECTS OF A CLASS (NMSC)**

Level: Lifeline-Level

Description: This metric count the number of messages sent by the objects instantiated from the class. It can be used for finding out which classes are actively involved in interactions within a system.

Reference: **[287]**

2. **NUMBER OF MESSAGES RECEIVED BY THE INSTANTIATED OBJECTS OF A CLASS (NMRC)**

Level: Lifeline-Level

Description: This metric is similar to the RFC metric of C&K metric suite. It counts the number of messages received by the objects instantiated from the class. It can be used for finding out which classes are actively involved in interactions within a system.

Reference: **[287]**

3. NUMBER OF MESSAGES (NMM)

Level: Model-Level

Description: Messages are exchanged between objects manifesting various interactions. This metric counts the number of messages within a sequence model.

Reference: **[287]**

4. NUMBER OF THE DIRECTLY DISPATCHED MESSAGES OF A MESSAGE (NDM)

Level: Message-Level

Description: According to the UML semantics, a message can be an activator of other messages. This metric counts the number of messages directly dispatched as a result of this message invocation.

Reference: **[287]**

5. **NUMBER OF THE ELEMENTS IN THE TRANSITIVE CLOSURE OF THE DIRECTLY DISPATCHED MESSAGES OF A MESSAGE (NDM*)**

Level: Message-Level

Description: This metric counts the transitive closure of all the messages activated as a result of this message being dispatched. It is potentially useful for predicting the lifelines and messages that might be affected if this message is modified or removed.

Reference: **[287]**

6. **NUMBER OF LIFELINES (LIFELINES)**

Level: Model-Level

Description: The metric counts the number of lifelines on the sequence model.

Reference: **[466]**

A9.3 Use Case Diagram Metrics

1. **NUMBER OF THE USE CASES (NUM)**

Level: Model-Level

Description: This metric counts the number of use cases in a use case model. The rationale behind the inclusion of this metric is

that a use case represents a coherent unit of functionality provided by a system, a subsystem, or a class.

Reference: **[287]**

2. NUMBER OF THE ACTORS (NAM)

Level: Model-Level

Description: This metric computes the number of actors in a use case model.

Reference: **[287]**

3. NUMBER OF THE ACTORS ASSOCIATED WITH A USE CASE (NACU)

Level: Use Case-Level

Description: This metric computes the number of actors that are associated with a use case, and it is useful to measure the importance of the requirement expressed by the use case. The reason for this argument is that the requirements that many actors concern are likely to be important for the system to function properly as a whole.

Reference: **[287]**

4. NUMBER OF USE CASES WHICH THIS EXTENDS (EXTENDING)

Level: Use Case-Level

Description: This metric counts the number of use cases extended by this use case.

Reference: **[466]**

5. NUMBER OF USE CASES WHICH EXTEND THIS USE CASE (EXTENDED)

Level: Use Case-Level

Description: The metric counts the number of use cases which extend this use case.

Reference: **[466]**

6. NUMBER OF USE CASES WHICH THIS INCLUDES (INCLUDING)

Level: Use Case-Level

Description: The metric counts the number of use cases which this use case includes.

Reference: **[466]**

7. NUMBER OF USE CASES THAT INCLUDES THIS USE CASE (INCLUDED)

Level: Use Case-Level

Description: The metric counts the number of use cases, which include this use case.

Reference: **[466]**

8. NUMBER OF EXTENSION POINTS OF THE USE CASE (ExtPts)

Level: Use Case-Level

Description: The metric counts the number of extension points in the use case. An extension point in a use case is a useful concept, but when too many are provided, it is a sign that perhaps the use case should be split up or modeled in a different way to improve readability.

Reference: **[466]**

9. DEPTH OF <<INCLUDE>> RELATIONSHIP (DOIR)

Level: Use Case-Level

Description: A series of nested <<include>> relationships in Use Case modeling is a sign of functional decomposition and makes for difficult reading. This metric computes the depth of the include relationship.

Reference: **[289]**

10. DEPTH OF <<EXTEND>> RELATIONSHIP (DOER)

Level: Use Case-Level

Description: An <<extend>> relationship is itself commonly misunderstood. A number of nested <<extend>> relationships can be difficult to understand and should be

discouraged. This metric computes the depth of the extend relationship.

Reference: **[289]**

Appendix 10: UML Model Smells

Models smells are defined as elements within the model that are potential candidates for improvements and refactoring. Either model smells could be symptoms of design defects or bad alternatives to recurring design problems in OO design also known as anti-patterns. Based on our detailed systematic literature review [18], we identified that 17 published studies proposed model smells over UML models. Models smells used in our work for validation and comparison were selected based on the following criteria:

1. Smells defined only over UML Class, Sequence and Use Case models. This is based on the scope of the work.
2. Smells defined are either metric-based or rule-based (heuristics). Design pattern based model smells are not considered in line of the scope of the work.
3. Smells defined should be measurable with threshold values clearly specified in the study.
4. Studies proposing model smells should associate refactoring strategy (solutions) to the identified refactoring opportunities.

A10.1 Class Model Smells

In this section, we describe smells for class models. Thirteen smells related to class diagram have been proposed in the literature. Of these, only seven satisfy our selection criteria. However, some of these smells either use information from more than one model in the detection and resolution strategy. Since we are concerned with smells that can be

detected and resolved only over the class diagram, a table of these seven smells along with the study is presented in Table A - 2 to aid in the selection. Grayed out cells indicate that the study did not address the particular smell in their proposal.

Table A - 2 Class Model Smells and their Information Dependence

Studies		GC	FD	LZC	DUP	Coup	RB	LF
Ruhroth et al. [32]	**Strategy**			CD Z		CD	CD	
	Solution			CD Z		CD	CD	
Fourati et al. [19]	**Strategy**	CD SEQ	CD					CD
	Solution	CD SEQ	CD					CD
Stolc and Polasek [236]	**Strategy**				CD			
	Solution				CD			
Enckevort [21]	**Strategy**	CD						
	Solution	Man						

(**GC:** God Class, **FD:** Functional Decomposition, **LZC:** Lazy Class, **DUP:** Duplication, **Coup:** Strong Coupling, **RB:** Refused Bequest, **LF:** Lava Flow, **CD:** Class Diagram, **SEQ:** Sequence Model, **Man:** Manual Refactoring by Expert, **Z:** Method description in the form of Z Language notation.)

Based on the information presented in Table A - 2, five class model smells were selected. These smells are described in this section.

1. **Functional Decomposition**

   a. **Smell Description:** Functional Decomposition anti-pattern described by Brown et al. [17] is a "main" routine that accesses numerous subroutines. Moha et al. [24] described functional decomposition as a smell that consists of a main class in which Inheritance and Polymorphism are rarely

used i.e. associated with small classes, has private attributes and implements a few methods.

b. **Smell Detection:** Fourati et al. [19] provided a rule based detection of this model smell. Moreover, an important constraint in their description is detecting whether the name of the class is functional or not (Hence, it is a semi-automated detection strategy). According to them, a functionally decomposed class has all private attributes (High Cohesion) and a single function. The detection rule proposed is shown below:

(IsFunctional (C) = true) & (NPrAtt = high) & (NOM = low) & (DIT = 0) & (NOC = 0)

IsFunctional: It tests if a class has a name as a function i.e. a verb or a noun action like 'Creation', 'Making' and so on.
NPrAtt: The number of Private Attributes

NOM: Number of Methods of a class including the constructor

DIT: It is the depth in the inheritance tree. Maximum length is considered in case of multiple inheritance.
NOC: It is the number of immediate subclasses subordinated to a class in the class hierarchy.

The *low* and *high* thresholds delimiting each metric in the above rule are based on the maximum value of each used metric. Due to the lack of information, we used max and min values suggested by Gronback [288].

c. **Smell Resolution:** Classes associated with the functionally decomposed class are merged together if the class is coupled with only one class (i.e. the functionally decomposed class accessing it).

2. **Class Duplication**

    a. **Smell Description:** This smell is derived from Fowler et al.'s Duplicate Code smell. Since a class model does not contain any code, its attributes, association ends and operations care used to identify duplication.

    b. **Smell Detection:** Stolc and Polasek [236] detects duplication in classes that are subclasses of a common superclass. Attribute *body* of each attribute in a subclass is compared with attribute bodies of other siblings to identify redundancy. If an attribute with the same body exists in all subclasses, then refactoring is recommended.

    c. **Smell Resolution:** *Pull up Property* Refactoring is used on all sub-classes to remove the attribute from all subclasses and added to the common superclass.

3. **Too Strong Coupling**

    a. **Smell Description:** Coupling in a class diagram measures the degree of dependency between classes. Two classes are coupled if at least one of them depends upon the other [62]. Strong Coupling is an indicator that classes in the model are too interdependent on others and is a sign of sensibility to changes and hence difficult to maintain.

    b. **Smell Detection:** Ruhroth et al. [32] used a combination of the Direct Class Coupling (DCC) metric [467] and MaxUsedPL metric to identify the smell. The detection rule proposed is shown below:

$$(DCC > UP_{DCC}) \; \& \; (MaxUsedPL > UP_{MaxUsedPL})$$

DCC: It counts the different number of classes that a class is directly related to. It includes classes that are directly related by attribute declarations and message passing (parameters) in methods.
MaxUsedPL: Maximum length of all parameter lists used by a class.

UP indicates the upper limit threshold for the metric. As indicated in their work $UP_{DCC} = 9$ and $UP_{MaxUsedPL} = 5$.

c. **Smell Resolution:** *Move Property* Refactoring is used to move attributes to classes where they are used more often. In case moving attributes cannot be performed, *Merge Class* Refactoring is used to join strongly coupled classes.

4. **Refused Bequest**

a. **Smell Description:** Refused Bequest is also one of the code smells proposed by Fowler et al. Subclasses get to inherit methods and data of their parents. This smell arises when subclasses use either some or none of the features inherited from their superclass.

b. **Smell Detection:** Ruhroth et al. [32] redefined the detection strategy for this traditional smell to detect it over class models. They used a combination of the OIF (Operations Inheritance Factor) metric and DIT to detect refused bequest model smell. The detection rule proposed is shown below:

$$(OIF < LOW_{OIF})\ \&\ (DIT >= 1)$$

OIF: It is a quotient between the number of inherited operations and the number of available operations (locally defined and inherited).

LOW indicates the lower limit threshold for the metric. As indicated in their work $LOW_{OIF} = 0.2$.

c. **Smell Resolution:** In order to resolve this smell, a new sibling class is created and *Push Down Operation* and *Push Down Property* Refactoring is used to push all the unused methods to the sibling. Hence, the parent holds only what is common.

5. **Lava Flow**

a. **Smell Description:** Lava Flow is a software development anti-pattern described by Brown et al. as dead code in an isolated class which makes it uncoupled from other classes.

b. **Smell Detection:** Fourati et al. [19] provided a rule based detection of this model smell. According to them, a class demonstrating lava flow model smell has large number of attributes and is complex and has no interactions. The detection rule proposed is shown below:

(NAtt = high) & (NAss = low) & (NOM = high) & (DIT = 0) & (NOC = 0)

NAss: The number of Associations (association link, aggregation, composition, dependency link) of a class.
NAtt: The number of Attributes of a class

The *low* and *high* thresholds delimiting each metric in the above rule are based on the maximum value of each used metric. Similar to what we adopted for the Functional Decomposition Smell, we used max and min values suggested by Gronback [288].

c. **Smell Resolution:** A semi-automatic solution is proposed for resolving this model smell. A class, if detected to contain lava flow smell, is presented to the user. If acceptable, the class is removed from the model.

A10.2 Sequence Model Smells

In this section, we describe smells for sequence models. This is the least studied topic in the field of model-driven refactoring. Although, Astels [251] was the first author to propose the use of sequence diagrams to detect the existence of *Middle Man* model smell, his approach was naïve and lacked a proper detection and resolution strategy. Only two studies discuss the identification of smells related to the UML sequence diagram. Liu et al. [263] proposed the duplication model smell over sequence diagrams using suffix trees as model representation. Since their approach is heavily based on a different representation formalism and tool support for detection and resolution is no longer available, we do not consider this model smell. The only other model smell Middle Man which is described below.

1. **Middle Man**

a. **Smell Description:** A Middle Man is a lifeline (or a class/object) that sits between two others and forwards method calls between them. A middle

man is apparent in the sequence diagram by the pattern of messages being simply delegated to another lifeline.

b. **Smell Detection:** Dobrzanski and Kuzniarz [262] provided a rule based detection of this model smell. According to them, a class is considered a middle man if it has an attribute with at least two *Simple Delegating Operations* (SDO). A special metamodel (TAU) is used to describe the body of an operation. Nevertheless, based on the description provided by Wake [16] and included in their detection approach, “most methods of a class call the same or similar method on another object.” We use this definition to define a detection strategy that evaluates the incoming and outgoing messages to a class. If similar names are used, the methods is classified as a delegating operation.

c. **Smell Resolution:** *Remove Middle Man* Refactoring is used to remove the middle man lifeline from the sequence diagram. It works for a pair of classes A and B, where A is a middle man of a delegate B. If the middle man delegates for more than one lifeline, the refactoring process is repeated for any pair consisting of A and a delegate class.

A10.3 Use Case Model Smells

Although quite a few studies proposed refactoring operations over Use Case (UC) diagrams, only one study by El-Attar and Miller [264] is available in the literature that

identified anti-patterns and discussed their resolution with refactoring solution. They proposed a total of 21 anti-patterns. We classified these into three categories as follows:

1. **OMG:** Anti-patterns that can be detected over the use case model conforming to the standard UML metamodel proposed by OMG (i.e. only the structural part of the use case model).
2. **Behavior:** Anti-patterns that require information about the flow of steps within each use case model. This information is not available in the standard UML metamodel and requires alternate description formalisms such as text, template and other UML diagrams. Hence, anti-patterns in this category are not considered in this work.
3. **Consistency:** Anti-patterns that describe violations to use case diagram well-formedness rules. Since each diagram undergoes consistency (syntactic and semantic) before refactoring, these smells are also excluded from this work.

Out of the all the anti-patterns presented in Table A - 3, only those conforming to the UML metamodel (1-8) were considered for our work. Based on our initial selection criteria, any smell without a quantitative threshold value is excluded from the work. Hence, anti-patterns 7 and 8 were also excluded as an upper limit threshold for *"too many"* was not provided and lack of detection strategy for classifying an actor as a device. Selected anti-patterns are described in detail in this section.

Table A - 3 Classification of Use Case Anti-patterns [263]

Use Case Anti-patterns					
OMG		**Consistency**		**Behavior**	
1	Using extension/inclusion UCs to implement an abstract UC	9	Two actors with the same name	14	Functional decomposition of UCs: Using the include relationship
2	Multiple generalizations of a UC	10	An actor inside the system boundary	15	Functional decomposition by using an extension UC to extend multiple UCs
3	An actor associated with an unimplemented abstract UC	11	An unassociated UC	16	Using instances for actors instead of roles
4	An actor associated with a generalized concrete UC	12	An association between two actors	17	An actor associated with an extension UC
5	Duplicating functionalities for the generalized and specializing UCs	13	An association between UCs	18	Functional decomposition by creating a call sequence between UCs using pre and post-conditions
6	A UC that is used as an extension and inclusion UC			19	Very large alternative flows
7	Too many UCs			20	UC initiated by two actors
8	Representing devices as actors			21	Using the term “actor” in textual descriptions

1. **Using extension/inclusion UC’s to implement an abstract UC**

   a. **Smell Description:** An actor is directly associated with an abstract UC. The implementation of this abstract UC is not done through a specializing UC but through extension or inclusion UCs instead.

   b. **Smell Detection:** The steps involved in the detection of this smell are as follows:

      i. Search for any abstract UC.

ii. If the (UC is associated with an actor) AND (is extended by or includes other UCs) AND (has no child UC).

c. **Smell Resolution:** The authors proposed a semi-automatic resolution strategy for this anti-pattern. If the inclusion or extension is justified, the UC is changed to *concrete*. If not, it is left to the modeler to add either specializing use case in the future or change relationship to inheritance instead of inclusion/extension.

2. **Multiple Generalizations of a UC**

   a. **Smell Description:** A single UC specializes two or more UCs. Multiple generalization leads to violation of behavioral semantics of the use case model.

   b. **Smell Detection:** The steps involved in the detection of this smell are as follows:

      i. Search for a child UC.

      ii. If the UC is specializing more than one UC.

   c. **Smell Resolution:** The specialization relationship is replaced by an include relationship. The include relationship is considered more appropriate since the shared UC contains common behavior not specializing behavior.

3. **An actor associated with an unimplemented abstract UC**

   a. Smell Description: An actor is directly associated with an abstract UC that is not implemented by specializing UC(s).

b. **Smell Detection:** The steps involved in the detection of this smell are as follows:

   i. Search for an abstract UC.

   ii. If the (UC is associated with an actor) AND (not specialized by at least one UC).

c. **Smell Resolution:** The authors propose a semi-automatic resolution strategy for this anti-pattern. The UC is changed to *concrete* in order to allow initiation by an actor.

4. An actor associated with a generalized concrete UC

a. **Smell Description:** Often generalized UCs only contain fragments of general behavior that is used by its specializing UCs. Therefore, generalized UCs are often incomplete. Such incomplete generalized UCs contain *"blanks"* that are intended to be "filled" by special behavior contained in the specializing UCs.

b. **Smell Detection:** The steps involved in the detection of this smell are as follows:

   i. Search for any generalized UC.

   ii. If the (UC is concrete) AND (associated with an actor).

c. **Smell Resolution:** Explicit associations between the actor and the specializing UCs is created in place of the association between the actor and the generalized UC. The explicit associations with the specializing UCs will enforce the service request to be performed through one of the specializing UCs.

5. **Duplicating functionalities for the generalized and specializing UCs**

   a. **Smell Description:** This anti-pattern detects duplication in the use case diagram. The relationships that a generalized UC has with other UCs are duplicated for the specializing UC. Hence, this leads to creation of duplicated or redundant code in the implementation phase.

   b. **Smell Detection:** The steps involved in the detection of this smell are as follows:

      i. Search for a generalization relationship between two UCs.

      ii. If both the generalized and specializing UCs have similar relationships with other UCs.

   c. **Smell Resolution:** The authors proposed a semi-automatic resolution strategy for this anti-pattern. Modelers determine whether a given included or extending UC is applicable to all of the specializing UCs or only a subset of them. Based on this if it is applicable to all other children, redundant path is removed. Else, irrelevant associations with other use cases is removed from the diagram.

6. **A UC that is used as an extension and inclusion UC**

   a. **Smell Description:** The reuse of a preexisting UC is achieved by making it both an extension UC and an inclusion UC. Object-oriented modeling and design strongly promotes the concept of reuse. However, when applying the concept of reuse, the include and the extend relationships can be misused leading to the creation of UCs containing both common and exception-handling behavior.

b. **Smell Detection:** The steps involved in the detection of this smell are as follows:

    i. Search for any included UCs.

    ii. If inclusion UC is extending other UC's.

c. **Smell Resolution:** The authors proposed a semi-automatic resolution strategy for this anti-pattern. Resolution is based on the type of situation encountered and is described below:

    i. If the shared UC contains functionality suitable for only the base UC that includes it, the extend relationship is removed. A new extension UC is created to handle the exceptional situation generated by the other base UC.

    ii. If the shared UC contains functionality suitable only for the base UC that it extends, the include relationship is removed. A new UC is created and included by the other base UC.

    iii. If the shared UC does contain both common and exception behavior, the shared UC is split into two separate UCs. Each of the newly created UCs should only contain functionality appropriate to the base UC.

References

[1] L. L. Constantine and A. D. Lockwood Lucy, "Structure and Style in Use Cases for User Interface Design," in *Object Modeling and User Interface Design*, M. van Harmelen: Addison-Wesley, 2001.

[2] N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *Journal of Software Maintenance,* vol. 13, pp. 3-30, 2001.

[3] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: a roadmap," *Proceedings of the 22nd International Conference on Software Engineering*, pp. 73-87, Limerick, Ireland, 2000.

[4] S. Cook, H. Ji, and R. Harrison, "Software Evolution and Software Evolvability," Working Paper, University of Reading, UK, 2000.

[5] E. J. Chikofsky and J. H. Cross, Reverse engineering and design recovery: a taxonomy," *IEEE Software*," vol. 7, pp. 13-17, IEEE, 1990.

[6] W. Opdyke, "Refactoring Object-Oriented Frameworks," PhD thesis, University of Illinois at Urbana Champaign, 1992.

[7] F. Fondement and R. Silaghi, "Defining Model Driven Engineering Processes," in *Third International Workshop in Software Model Engineering (WiSME)*, 2004.

[8] R. B. France and J. M. Bieman, "Multi-View Software Evolution: A UML-based Framework for Evolving Object-Oriented Software," in *17th IEEE International Conference on Software Maintenance (ICSM'01)*, p. 386, 2001.

[9] J. Miller and J. Mukerji, "MDA Guide Version 1.0.1," 2003.

[10] OMG, "Unified Modeling Language: Superstructure," Version: 2.4.1, formal/2011-08-06, Object Management Group, 2011.

[11] T. Mens, S. Demeyer, B. Du Bois, H. Stenten, and P. Van Gorp, "Refactoring: Current Research and Future Trends," *Electronic Notes in Theoretical Computer Science,* vol. 82, 2003.

[12] T. Mens, G. Taentzer, and D. Müller, "Challenges in Model Refactoring," *International Workshop on Object-Oriented Reengineering*, Berlin, Germany, 2007.

[13] T. Mens, G. Taentzer, and D. Müller, "Model-Driven Software Refactoring," in *Model-Driven Software Development: Integrating Quality Assurance*: IDEA Group Publishing, 2008.

[14] R. Van Der Straeten, T. Mens, and S. Van Baelen, "Challenges in Model-Driven Software Engineering," in *Models in Software Engineering*. vol. 5421, M. Chaudron, Berlin / Heidelberg: Springer, pp. 35-47, 2009.

[15] M. Fowler, K. Beck, J. Brant, and W. Opdyke, *Refactoring: Improving the Design of Existing Code*: Addison-Wesley, 1999.

[16] W. C. Wake, *Refactoring Workbook*: Addison-Wesley, 2003.

[17] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *AntiPatterns: Refactoring Software Architectures and Projects in Crisis*, 1st Edition ed.: John Wiley and Sons, 1998.

[18] M. Misbhauddin and M. Alshayeb, "Model-driven software refactoring: A systematic review," Manuscript submitted for publication, King Fahd University (KFUPM), Saudi Arabia.

[19] R. Fourati, N. Bouassida, and H. Abdallah, "A Metric-Based Approach for Anti-pattern Detection in UML Designs," in *Computer and Information Science*. vol. 364, R. Lee, Berlin / Heidelberg: Springer, pp. 17-33, 2011.

[20] M. Mohamed, M. Romdhani, and K. Ghedira, "M-REFACTOR: A New Approach and Tool for Model Refactoring " *ARPN Journal of Systems and Software,* vol. 1, pp. 117-122, 2011.

[21] T. v. Enckevort, "Refactoring UML models: using openarchitectureware to measure uml model quality and perform pattern matching on UML models with OCL queries," *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pp. 635-646, Orlando, Florida, USA, 2009.

[22] A. Ghannem, M. Kessentini, and G. El Boussaidi, "Detecting model refactoring opportunities using heuristic search," *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pp. 175-187, Toronto, Ontario, Canada, 2011.

[23] M. T. Llano and R. Pooley, "UML Specification and Correction of Object-Oriented Anti-patterns," in *Fourth International Conference on Software Engineering Advances*, pp. 39-44, 2009.

[24] N. Moha, Y. G. Gueheneuc, L. Duchien, and A. F. Le Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Transactions on Software Engineering,* vol. 36, pp. 20-36, 2010.

[25] M. v. Kempen, M. Chaudron, D. Kourie, and A. Boake, "Towards proving preservation of behaviour of refactoring of UML models," *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, pp. 252-259, White River, South Africa, 2005.

[26] J. Muskens, R. J. Bril, and M. R. V. Chaudron, "Generalizing Consistency Checking between Software Views," in *5th Working IEEE/IFIP Conference on Software Architecture (WICSA)* pp. 169-180, 2005.

[27] A. Egyed and N. Medvidovic, "Extending architectural representation in UML with view integration," *Proceedings of the 2nd international conference on The unified modeling language: beyond the standard*, pp. 2-16, Fort Collins, CO, USA, 1999.

[28] A. Egyed, "UML/Analyzer: A Tool for the Instant Consistency Checking of UML Models," *Proceedings of the 29th international conference on Software Engineering*, pp. 793-796, 2007.

[29] J. Muskens, M. Chaudron, and C. Lange, "Investigations in Applying Metrics to Multi-View Architecture Models," in *Proceedings of the 30th EUROMICRO Conference*, pp. 372-379, 2004.

[30] M. Ahmed, "Towards the Development of Integrated Reuse Environments for UML Artifacts," in *The Sixth International Conference on Software Engineering Advances*, 2011.

[31] J. Derrick and H. Wehrheim, "Model transformations across views," *Science of Computer Programming,* vol. 75, pp. 192-210, 2010.

[32] T. Ruhroth, H. Voigt, and H. Wehrheim, "Measure, diagnose, refactor: a formal quality cycle for software models," *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications,* pp. 360-367, 2009.

[33] J. Iivari, "Object-orientation as structural, functional and behavioural modelling: a comparison of six methods for object-oriented analysis," *Information and Software Technology,* vol. 37, pp. 155-163, 1995.

[34] OMG, "UML 2.0 OCL Specification," Object Management Group, 2003.

[35] OMG, "XML Metadata Interchange Specification 2.1.1," formal/2007-12-01, Object Management Group, 2007.

[36] E. Seidewitz, "What models mean," *IEEE Software,* vol. 20, pp. 26-32, 2003.

[37] G. Booch, *Object-Oriented Analysis and Design with Applications* vol. Second: Benjamin-Cummings Publishing Co., 1993.

[38] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*: Prentice Hall, 1991.

[39] I. Jacobson, *Object Oriented Software Engineering: A Use Case Driven Approach*: Addison-Wesley, 1992.

[40] B. Arief and N. A. Speirs, "A UML tool for an automatic generation of simulation programs," in *Proceedings of the 2nd international workshop on Software and performance*, pp. 71-76, NY, USA, 2000.

[41] B. Meyer, UML: the positive spin," *American Programmer*," 1997.

[42] P. Kruchten, The 4+1 View Model of Architecture," *IEEE Software*," vol. 12, pp. 42-50, IEEE Computer Society, Available: http://doi.ieeecomputersociety.org/10.1109/52.469759, 1995.

[43] S. Shlaer and S. J. Mellor, *Object Oriented Systems Analysis: Modeling the World in Data*: Prentice Hall, 1988.

[44] S. J. Mellor and S. Shlaer, *Object Life Cycles: Modeling the World In States*: Prentice Hall, 1991.

[45] OMG, "Meta Object Facility (MOF)," formal/2006-01-01, Object Management Group, 2006.

[46] S. Sendall and W. Kozaczynski, Model Transformation: The Heart and Soul of Model-Driven Software Development," *IEEE Software*," vol. 20, pp. 42-45, 2003.

[47] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal,* vol. 45, pp. 621-646, 2006.

[48] K. Czarnecki and S. Helsen, "Classification of Model Transformation Approaches," in *Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[49] N. Prakash, S. Srivastava, and S. Sabharwal, "The Classification Framework for Model Transformation," *Journal of Computer Science,* vol. 2, pp. 166-170, 2006.

[50] T. Mens and P. Van Gorp, "A Taxonomy of Model Transformation," *Electronic Notes in Theoretical Computer Science,* vol. 152, pp. 125-142, 2006.

[51] K. T. Kalleberg, "Abstractions for Language-Independent Program Transformations," Doctoral thesis, Department of Informatics, University of Bergen, Bergen, Norway, 2007.

[52] A. Habel, R. Heckel, and G. Taentzer, "Graph grammars with negative application conditions," *Fundam. Inf.,* vol. 26, pp. 287-313, 1996.

[53] D. Jackson, I. Shlyakhter, and M. Sridharan, "A micromodularity mechanism," *SIGSOFT Software Enggineering Notes,* vol. 26, pp. 62-73, 2001.

[54] D. Jackson, *Software abstractions: logic, language and analysis*: MIT Press, 2006.

[55] J. M. Spivey, *The Z notation: a reference manual*. Hertfordshire, UK: Prentice Hall International, 1992.

[56] G. Smith, *The Object-Z Specification Language*: Kluwer Academic Publisher, 2000.

[57] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, *The Description Logic Handbook: Theory, Implementation and Applications.*: Cambridge University Press, 2003.

[58] OMG, "Meta object faacility (MOF) 2.0 query/view/transformation specification," ad/04-10-11, 2004.

[59] openArchitectureware.org. (March 2012). *openArchitectureWare (oAW)*. Available: http://www.openarchitectureware.org/

[60] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, "ATL: a QVT-like transformation language," *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pp. 719-720, Portland, Oregon, USA, 2006.

[61] J. M. Bieman and B.-K. Kang, "Cohesion and reuse in an object-oriented system," *SIGSOFT Software Enggineering Notes,* vol. 20, pp. 259-262, 1995.

[62] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Transactions on SoftwareEngineering,* vol. 20, 1994.

[63] M. Genero, M. Piattini, and C. Calero, "Empirical validation of class diagram metrics," in *Proceedings. 2002 International Symposium in Empirical Software Engineering*, pp. 195-203, 2002.

[64] R. Marinescu, "Detecting Design Flaws via Metrics in Object-Oriented Systems," in *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS '39)*, Washington, DC, USA, 2001.

[65] P. F. Mihancea and R. Marinescu, "Towards the Optimization of Automatic Detection of Design Flaws in Object-Oriented Software Systems," in *Ninth European Conference on Software Maintenance and Reengineering*, pp. 92-101, 2005.

[66] N. Tsantalis and A. Chatzigeorgiou, "Ranking Refactoring Suggestions Based on Historical Volatility," in *15th European Conference on Software Maintenance and Reengineering*, pp. 25-34, 2011.

[67] R. B. France, S. Ghosh, E. Song, and D.-K. Kim, A metamodeling approach to pattern-based model refactoring," *IEEE Software*," vol. 20, pp. 52-58, 2003.

[68] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison Wesley, 1994.

[69] C. F. J. Lange and M. R. V. Chaudron, "Managing Model Quality in UML-Based Software Development," in *13th IEEE International Workshop on Software Technology and Engineering Practice*, pp. 7-16, 2005.

[70] A. Egyed, "Instant consistency checking for the UML," *Proceedings of the 28th international conference on Software engineering*, pp. 381-390, Shanghai, China, 2006.

[71] G. Spanoudakis and A. Zisman, "Management of inconsistencies in software engineering: a survey of the state of the art," in *Handbook of Software Engineering and Knowledge Engineering*. vol. 1: World Scientific Publishing Co., pp. 329-380, 2001.

[72] Z. Huzar, L. Kuzniarz, G. Reggio, and J. Sourrouille, "Consistency Problems in UML-Based Software Development," in *UML Modeling Languages and Applications*. vol. 3297, N. Jardim Nunes, B. Selic, A. Rodrigues da Silva, and A. Toval Alvarez, Berlin / Heidelberg: Springer pp. 1-12, 2005.

[73] T. Mens and T. Tourwe, "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering,* vol. 30, pp. 126-139, 2004.

[74] T. Mens and A. Van Deursen, "Refactoring: Emerging Trends and Open Problems," in *Proceedings First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE)*, 2003.

[75] B. Du Bois, P. Van Gorp, A. Amsel, N. Van Eetvelde, H. Stenten, S. Demeyer, and T. Mens, "A discussion of refactoring in research and practice," University of Antwerp, Belgium,Available: http://win.ua.ac.be/~lore/refactoringProject/publications/ADiscussionOfRefactoringInResearchAndPractice.pdf, 2004.

[76] J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," in *Proc. of ACM-SIGMOD Int'l Conference on Management of Data*, pp. 311-322, 1987.

[77] D. B. Roberts, "Practical Analysis for Refactoring," University of Illinois at Urbana-Champaign, 1999.

[78] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *Proceedings of the International Conference on Software Maintenance*, Washington, DC, USA, 1998.

[79] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Measuring Clone Based Reengineering Opportunities," in *Proceedings of the 6th International Symposium on Software Metrics*, Washington, DC, USA, 1999.

[80] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, "Advanced Clone-Analysis to Support Object-Oriented System Refactoring," in *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, Washington, DC, USA, 2000.

[81] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," in *Static Analysis*. vol. 2126 Berlin / Heidelberg: Springer, pp. 40-56, 2001.

[82] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," in *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, Washington, DC, USA, 2001.

[83] N. Tsantalis and A. Chatzigeorgiou, "Identification of Extract Method Refactoring Opportunities," *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, pp. 119-128, 2009.

[84] N. Juillerat and B. Hirsbrunner, "FOOD: An Intermediate Model for Automated Refactoring," *Proceedings of the Conference on New Trends in Software Methodologies, Tools and Techniques*, pp. 452-461, 2006.

[85] T. Mens, N. Eetvelde, S. Demeyer, and D. Janssens, "Formalizing refactorings with graph transformations," *Software Maintenance and Evolution : Research and Practice,* vol. 17, pp. 247-276, July/August 2005 2005.

[86] T. Tourwe and T. Mens, "Identifying Refactoring Opportunities Using Logic Meta Programming," in *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, Washington, DC, USA, 2003.

[87] P. Meananeatra, S. Rongviriyapanish, and T. Apiwattanapong, "Identifying refactoring through formal model based on data flow graph," *5th Malaysian Conference in Software Engineering (MySEC),* pp. 113-118118, 2011 2011.

[88] N. Tsantalis and A. Chatzigeorgiou, "Identification of Move Method Refactoring Opportunities," *IEEE Transactions on Software Enggineering,* vol. 35, pp. 347-367, 2009.

[89] R. Oliveto, M. Gethers, G. Bavota, D. Poshyvanyk, and A. De Lucia, "Identifying method friendships to remove the feature envy bad smell (NIER track)," *Proceedings of the 33rd International Conference on Software Engineering*, pp. 820-823, Waikiki, Honolulu, HI, USA, 2011.

[90] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," in *Proceedings of the IEEE International Conference on Software Maintenance*, Washington, DC, USA, 1999.

[91] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering,* vol. 28, pp. 654-670, 2002.

[92] R. Wettel and R. Marinescu, "Archeology of Code Duplication: Recovering Duplication Chains from Small Duplication Fragments," in *Proceedings of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Washington, DC, USA, 2005.

[93] F. Van Rysselberghe and S. Demeyer, "Evaluating Clone Detection Techniques from a Refactoring Perspective," *Proceedings of the 19th IEEE international conference on Automated software engineering*, pp. 336-339, 2004.

[94] R. Tairas and J. Gray, "An information retrieval process to aid in the analysis of code clones," *Empirical Software Enggineering,* vol. 14, pp. 33-56, 2009.

[95] A. Marcus and J. I. Maletic, "Identification of High-Level Concept Clones in Source Code," *Proceedings of the 16th IEEE international conference on Automated software engineering*, p. 107, 2001.

[96] G. G. Koni, "A Scenario Based Approach for Refactoring Duplicated Code in Object Oriented Systems," Institute of Computer Science, University of Bern, 2001.

[97] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Refactoring Support Based on Code Clone Analysis," in *Product Focused Software Process Improvement*. vol. 3009;3009 Berlin / Heidelberg: Springer, pp. 220-233, 2004.

[98] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "On Refactoring Support Based on Code Clone Dependency Relation," in *Proceedings of the 11th IEEE International Software Metrics Symposium*, Washington, DC, USA, 2005.

[99] M. Rieger, S. Ducasse, and M. Lanza, "Insights into System-Wide Code Duplication," *Proceedings of the 11th Working Conference on Reverse Engineering*, pp. 100-109, 2004.

[100] H. A. Basit and S. Jarzabek, "Detecting higher-level similarity patterns in programs," *SIGSOFT Software Enggineering Notes,* vol. 30, pp. 156-165, 2005.

[101] E. Merlo, G. Antoniol, M. Di Penta, and V. F. Rollo, "Linear Complexity Object-Oriented Similarity for Clone Detection and Software Evolution Analyses," *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pp. 412-416, 2004.

[102] C. Kapser and M. W. Godfrey, "Aiding Comprehension of Cloning Through Categorization," *Proceedings of the Principles of Software Evolution, 7th International Workshop*, pp. 85-94, 2004.

[103] C. Kapser and M. W. Godfrey, ""Cloning Considered Harmful" Considered Harmful," *Proceedings of the 13th Working Conference on Reverse Engineering*, pp. 19-28, 2006.

[104] A. Monden, D. Nakae, T. Kamiya, S.-i. Sato, and K.-i. Matsumoto, "Software Quality Analysis by Code Clones in Industrial Legacy Software," *Proceedings of the 8th International Symposium on Software Metrics*, p. 87, 2002.

[105] R. Hill and J. Rideout, "Automatic Method Completion," *Proceedings of the 19th IEEE international conference on Automated software engineering*, pp. 228-235, 2004.

[106] F. Simon, F. Steinbrückner , and C. Lewerentz, "Metrics Based Refactoring," in *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, Washington, DC, USA, 2001.

[107] P. Joshi and R. K. Joshi, "Concept Analysis for Class Cohesion," *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, pp. 237-240, 2009.

[108] L. Zhao and J. H. Hayes, "Predicting Classes in Need of Refactoring: An Application of Static Metrics," in *Workshop on Predictive Models of Software Engineering (PROMISE), associated with ICSM 2006*, 2006.

[109] D. C. Atkinson and T. King, "Lightweight Detection of Program Refactorings," *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pp. 663-670, 2005.

[110] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring-Improving Coupling and Cohesion of Existing Code," in *11th Working Conference on Reverse Engineering*, pp. 144-151, 2004.

[111] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures," *Journal of Systems and Software,* vol. 84, pp. 397-414, 2011.

[112] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "A two-step technique for extract class refactoring," *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 151-154, Antwerp, Belgium, 2010.

[113] D. Boshnakoska and A. Mišev, "Correlation between Object-Oriented Metrics and Refactoring," in *ICT Innovations 2010*. vol. 83, M. Gusev and P. Mitrevski: Springer Berlin Heidelberg, pp. 226-235, 2011.

[114] A. De Lucia, R. Oliveto, and L. Vorraro, "Using structural and semantic metrics to improve class cohesion," in *IEEE International Conference on Software Maintenance*, pp. 27-36, 2008.

[115] R. Marinescu, "Measurement and quality in object-oriented design," in *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 701-704, 2005.

[116] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," in *Proceedings of 20th IEEE International Conference Software Maintenance*, pp. 350-359, 2004.

[117] M. J. Munro, "Product Metrics for Automatic Identification of 'Bad Smell' Design Problems in Java Source-Code," in *Proceedings of the 11th IEEE International Software Metrics Symposium*, Washington, DC, USA, 2005.

[118] N. Moha, Y.-G. Gueheneuc, and P. Leduc, "Automatic Generation of Detection Algorithms for Design Defects," *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pp. 297-300, 2006.

[119] M. Zhang, N. Baddoo, P. Wernick, and T. Hall, "Improving the Precision of Fowler's Definitions of Bad Smells," *Proceedings of the 2008 32nd Annual IEEE Software Engineering Workshop*, pp. 161-166, 2008.

[120] S. Singh and K. S. Kahlon, "Effectiveness of encapsulation and object-oriented metrics to refactor code and identify error prone classes using bad smells," *SIGSOFT Software Enggineering Notes,* vol. 36, pp. 1-10, 2011.

[121] E. Piveta, M. Pimenta, J. Araújo, A. Moreira, P. Guerreiro, and R. T. Price, "Representing refactoring opportunities," *Proceedings of the 2009 ACM symposium on Applied Computing*, pp. 1867-1872, Honolulu, Hawaii, 2009.

[122] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," *SIGPLAN Notes,* vol. 35, pp. 166-177, 2000.

[123] M. Salehie, S. Li, and L. Tahvildari, "A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws," *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pp. 159-168, 2006.

[124] A. Trifu and U. Reupke, "Towards Automated Restructuring of Object Oriented Systems," in *11th European Conference on Software Maintenance and Reengineering*, pp. 39-48, 2007.

[125] L. Tahvildari and K. Kontogiannis, "A Metric-Based Approach to Enhance Design Quality through Meta-pattern Transformations," *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, p. 183, 2003.

[126] T. Dudziak and J. Wloka, "Tool-Supported Discovery and Refactoring of Structural Weaknesses in Code," Technical University of Berlin, 2002.

[127] Y. Kosker, B. Turhan, and A. Bener, "An expert system for determining candidate software classes for refactoring," *Expert Systems with Applications,* vol. 36, pp. 10000-10003, 2009.

[128] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A Bayesian Approach for the Detection of Code and Design Smells," *Proceedings of the 2009 Ninth International Conference on Quality Software*, pp. 305-314, 2009.

[129] Y. Köşker, B. Turhan, and A. Bener, "Refactoring prediction using class complexity metrics," *International Conference on Software Paradigm Trends (ICSOFT)*, pp. 289-292, Porto, Portugal, 2008.

[130] H. A. Sahraoui, R. Godin, and T. Miceli, "Can Metrics Help to Bridge the Gap Between the Improvement of OO Design Quality and Its Automation?," *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, p. 154, 2000.

[131] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, and J. Sander, "Decomposing object-oriented class modules using an agglomerative clustering technique," in *25th IEEE International Conference on Software Maintenance*, pp. 93-101, Alberta, Canada, 2009.

[132] N. Anquetil, C. Fourrier, and T. C. Lethbridge, "Experiments with Clustering as a Software Remodularization Method," *Proceedings of the Sixth Working Conference on Reverse Engineering*, p. 235, 1999.

[133] I. G. Czibula and G. Şerban, "Improving Systems Design Using a Clustering Approach," *International Journal of Computer Science and Network Security,* vol. 6, 2006.

[134] G. Serban and I. G. Czibula, "Restructuring software systems using clustering," in *22nd international symposium on Computer and information sciences*, pp. 1-6, 2007.

[135] A. Alkhalid, M. Alshayeb, and S. Mahmoud, "Software Refactoring at the Function Level Using New Adaptive K-Nearest Neighbor Algorithm," *Advances in Engineering Software,* vol. 41, pp. 1160-1178, 2010.

[136] A. Alkhalid, M. Alshayeb, and S. Mahmoud, "Software Refactoring at the Package Level Using Clustering Techniques," *IET Software,* vol. 5, pp. 276-284, 2011.

[137] P. Lerthathairat and N. Prompoon, "An Approach for Source Code Classification Using Software Metrics and Fuzzy Logic to Improve Code Quality with Refactoring Techniques," in *Software Engineering and Computer Systems*. vol. 181, J. M. Zain, W. M. b. Wan Mohd, and E. El-Qawasmeh: Springer Berlin Heidelberg, pp. 478-492, 2011.

[138] J. Kerievsky, *Refactoring to Patterns*: Addison-Wesley, 2005.

[139] Y.-G. Guéhéneuc and H. Albin-Amiot, "Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-Class Design Defects," *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, p. 296, 2001.

[140] J. H. Jahnke and A. Zündorf, "Rewriting Poor Design Patterns by Good Design Patterns," in *Proceedings of ESEC/FSE '97 Workshop on Object-Oriented Reengineering*, 1997.

[141] S.-U. Jeon, J.-S. Lee, and D.-H. Bae, "An Automated Refactoring Approach to Design Pattern-Based Program Transformations in Java Programs," *Proceedings of the Ninth Asia-Pacific Software Engineering Conference*, p. 337, 2002.

[142] C. Jebelean, "Automatic Detection of Missing Abstract Factory Design Pattern in Object-Oriented Code," *Proceedings of the International Conference on Technical Informatics*, Politehnica University in Timisoara, 2004.

[143] C. Jebelean, C.-B. Chirila, and V. Cretu, "A logic based approach to locate composite refactoring opportunities in object-oriented code," *Proceedings of the 2010 IEEE International Conference on Automation, Quality and Testing, Robotics*, pp. 1-6, 2010.

[144] T. Shimomura, K. Ikeda, and M. Takahashi, "An Approach to GA-Driven Automatic Refactoring Based on Design Patterns," in *Fifth International Conference on Software Engineering Advances*, pp. 213-218, 2010.

[145] A. Chatzigeorgiou and A. Manakos, "Investigating the Evolution of Bad Smells in Object-Oriented Code," in *Seventh International Conference on the Quality of Information and Communications Technology*, pp. 106-115, 2010.

[146] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pp. 390-400, 2009.

[147] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An Exploratory Study of the Impact of Code Smells on Software Change-proneness," *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, pp. 75-84, 2009.

[148] S. Olbrich, D. S. Cruzes, and D. I. K. Sjoberg, "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems," *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, pp. 1-10, 2010.

[149] J. Pérez and Y. Crespo, "Perspectives on automated correction of bad smells," *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pp. 99-108, Amsterdam, The Netherlands, 2009.

[150] M. Zhang, T. Hall, and N. Baddoo, "Code Bad Smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: Research and Practice,* vol. 23, pp. 179-202, 2011.

[151] R. Heckel, "Algebraic Graph Transformations with Application Conditions," TU Berlin, 1995.

[152] J. U. Pipka, "Refactoring in a "Test First"-World," *XP-2002-Proceedings of the 3rd International Conference on eXtreme Programming and Flexible Proceses in Software Engineering*, 2002.

[153] A. van Deursen and L. Moonen, "The Video Store Revisited—Thoughts on Refactoring and Testing," *XP 2002 - Proceedings of the 3rd International Conference on eXtreme Programming and Flexible Proceses in Software Engineering*, 2002.

[154] N. Tsantalis and A. Chatzigeorgiou, "Identification of refactoring opportunities introducing polymorphism," *J. Syst. Softw.,* vol. 83, pp. 391-404, 2010.

[155] H. Liu, L. Yang, Z. Niu, Z. Ma, and W. Shao, "Facilitating software refactoring with appropriate resolution order of bad smells," *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 265-268, Amsterdam, The Netherlands, 2009.

[156] H. Liu, Z. Ma, W. Shao, and Z. Niu, "Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort," *IEEE Transactions on Software Engineering,* vol. 38, pp. 220-235, 2012.

[157] M. Zhang, N. Baddoo, P. Wernick, and T. Hall, "Prioritising Refactoring Using Code Bad Smells," *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 458-464, 2011.

[158] N. Tsantalis and A. Chatzigeorgiou, "Ranking Refactoring Suggestions Based on Historical Volatility," in *15th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 25-34, 2011.

[159] Y.-P. Cheng and J.-R. Liao, "An ontology-based taxonomy of bad code smells," *Proceedings of the third conference on IASTED International Conference:*

Advances in Computer Science and Technology, pp. 437-442, Phuket, Thailand, 2007.

[160] T. Mens, G. Taentzer, and O. Runge, "Analysing refactoring dependencies using graph transformation," *Software and Systems Modeling,* vol. 6, pp. 269-285, 2007.

[161] F. Qayum and R. Heckel, "Analysing refactoring dependencies using unfolding of graph transformation systems," *Proceedings of the 7th International Conference on Frontiers of Information Technology*, pp. 1-5, Abbottabad, Pakistan, 2009.

[162] F. Qayum and R. Heckel, "Search-Based Refactoring based on Unfolding of Graph Transformation Systems," *Proceedings of the Fifth International Conference on Graph Transformation - Doctoral Symposium*, 2010.

[163] E. Piveta, J. Araújo, M. Pimenta, A. Moreira, P. Guerreiro, and R. T. Price, "Searching for Opportunities of Refactoring Sequences: Reducing the Search Space," *Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference*, pp. 319-326, 2008.

[164] M. F. Zibran and C. K. Roy, "A Constraint Programming Approach to Conflict-Aware Optimal Scheduling of Prioritized Code Clone Refactoring," *Proceedings of the 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, pp. 105-114, 2011.

[165] M. F. Zibran and C. K. Roy, "Conflict-Aware Optimal Scheduling of Code Clone Refactoring: A Constraint Programming Approach," in *IEEE 19th International Conference on Program Comprehension (ICPC)*, pp. 266-269, 2011.

[166] H. Liu, G. Li, Z. Y. Ma, and W. Z. Shao, "Conflict-aware schedule of software refactorings," *IET Software,* vol. 2, pp. 446-460, 2008.

[167] H. Liu, G. Li, Z. Ma, and W. Shao, "Scheduling of conflicting refactorings to promote quality improvement," *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 489-492, Atlanta, Georgia, USA, 2007.

[168] L.-j. Zhang and X.-f. Xie, "Software refactoring scheme optimization model based on set pair analysis," *Application Research of Computers,* vol. 27, pp. 4175-41774177, Nov. 2010.

[169] L.-j. Zhang and X.-f. Xie, "Application of Identical Degree of Set Pair Analysis on Software Refactoring," in *International Conference on Computational Intelligence and Software Engineering (CiSE)*, pp. 1-4, 2010.

[170] S. Lee, G. Bae, H. S. Chae, D.-H. Bae, and Y. R. Kwon, "Automated scheduling for clone-based refactoring using a competent GA," *Software: Practice and Experience,* vol. 41, pp. 521-550, 2011.

[171] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pp. 1909-1916, Seattle, Washington, USA, 2006.

[172] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pp. 1106-1113, London, England, 2007.

[173] M. O'Keeffe and M. Ó Cinnéide, "Search-based refactoring for software maintenance," *Journal of Systems and Software,* vol. 81, pp. 502-516, 2008.

[174] R. Arcoverde, A. Garcia, and E. Figueiredo, "Understanding the longevity of code smells: preliminary results of an explanatory survey," *Proceedings of the 4th Workshop on Refactoring Tools*, pp. 33-36, Waikiki, Honolulu, HI, USA, 2011.

[175] M. O'Keeffe and M. Ó Cinnéide, "Getting the most from search-based refactoring," *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pp. 1114-1120, London, England, 2007.

[176] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Automated Software Engineering,* pp. 1-33, 2012.

[177] H. Kilic, E. Koc, and I. Cereci, "Search-based parallel refactoring using population-based direct approaches," *Proceedings of the Third international conference on Search based software engineering*, pp. 271-272, Szeged, Hungary, 2011.

[178] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Proceedings of the 9th Working Conference on Reverse Engineering*, pp. 97-107, 2002.

[179] A. Trifu and R. Marinescu, "Diagnosing design problems in object oriented systems," in *12th Working Conference on Reverse Engineering*, p. 10 pp., 2005.

[180] E. Duala-Ekoko and M. P. Robillard, "Clonetracker: tool support for code clone management," *Proceedings of the 30th international conference on Software engineering*, pp. 843-846, Leipzig, Germany, 2008.

[181] D. B. Roberts, J. Brant, and R. Johnson, "A refactoring tool for Smalltalk," *Theory and Practice of Object Systems,* vol. 3, pp. 253-263, 1997.

[182] E. Foundation. (2012). Eclipse IDE. Online: http://www.eclipse.org/downloads/

[183] JetBrains. (2012). IntelliJ IDEA.

[184] C. Zannier, "Tool support for refactoring to design patterns," *Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 122-123, Seattle, Washington, 2002.

[185] M. O'Keeffe and M. O. Cinneide, "Automated Design Improvement by Example," *Proceedings of the 2007 conference on New Trends in Software Methodologies, Tools and Techniques*, pp. 315-329, 2007.

[186] N. Sudhakar and J. Gyani, "TECDP: a tool for extracting creational design patterns," *Proceedings of the International Conference and Workshop on Emerging Trends in Technology*, pp. 735-736, Mumbai, Maharashtra, India, 2010.

[187] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "JDeodorant: identification and application of extract class refactorings," *Proceedings of the 33rd International Conference on Software Engineering*, pp. 1037-1039, Waikiki, Honolulu, HI, USA, 2011.

[188] S. Erb, "A Survey of Software Refactoring Tools," *Course of Applied Computer Science,* Baden-Württemberg Cooperative State University, Karlsruhe,Available: http://stephanerb.eu/files/erb2010b_Survey_of_Software_Refactoring_Tools.pdf, 2010.

[189] B. Du Bois and T. Mens, "Describing the impact of refactoring on internal program quality," in *Proceedings of the workshop on Evolution of Large-Scale Industrial Software Applications (ELISA)*, pp. 37-48, Netherlands, 2003.

[190] K. Stroggylos and D. Spinellis, "Refactoring--Does It Improve Software Quality?," in *International Conference on Software Engineering Workshops*, pp. 10-16, 2007.

[191] S. Bryton and F. B. e. Abreu, "Strengthening Refactoring: Towards Software Evolution with Quantitative and Experimental Grounds," *Proceedings of the 2009 Fourth International Conference on Software Engineering Advances*, pp. 570-575, 2009.

[192] B. Du Bois, S. Demeyer, and J. Verelst, "Refactor to Understand" Reverse Engineering Pattern Improve Program Comprehension?," in *European Conference on Software Maintenance and Reengineering*, pp. 334-343, 2005.

[193] M. Alshayeb, "Empirical investigation of refactoring effect on software quality," *Information and Software Technology,* vol. 51, pp. 1319-1326, 2009.

[194] M. Alshayeb, "The Impact of Refactoring to Patterns on Software Quality Attributes," *Arabian Journal for Science and Engineering,* vol. 36, pp. 1241-1251, 2011.

[195] B. Geppert, A. Mockus, and F. Rossler, "Refactoring for Changeability: A way to go?," in *11th IEEE International Software Metrics Symposium*, 2005.

[196] D. Wilking, U. F. Khan, and S. Kowalewski, "An Empirical Evaluation of Refactoring," *e-Informatica Software Engineering Journal,* vol. 1, pp. 27-42, 2007.

[197] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A Quantitative Evaluation of Maintainability Enhancement by Refactoring," in *Proceedings of the International Conference on Software Maintenance*, pp. 576-585, Washington, DC, 2002.

[198] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi, "Does refactoring improve reusability?," in *9th International Conference on Software Reuse*, pp. 287-297, Berlin, Heidelberg, 2006.

[199] J. Ratzinger, T. Sigmund, and H. C. Gall, "On the relation of refactorings and software defect prediction," *Proceedings of the 2008 international working conference on Mining software repositories*, pp. 35-38, Leipzig, Germany, 2008.

[200] A. Ampatzoglou and A. Chatzigeorgiou, "Evaluation of object-oriented design patterns in game development," *Information and Software Technology,* vol. 49, pp. 445-454, 2007.

[201] B. Huston, "The effects of design pattern application on metric scores," *Journal of Systems and Software,* vol. 58, pp. 261-269, 2001.

[202] N.-L. Hsueh, P.-H. Chu, and W. Chu, "A quantitative approach for evaluating the quality of design patterns," *Journal of Systems and Software,* vol. 81, pp. 1430-1439, 2008.

[203] F. Khomh and Y.-G. Gueheneuce, "Do Design Patterns Impact Software Quality Positively?," *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, pp. 274-278, 2008.

[204] A. Ampatzoglou, G. Frantzeskou, and I. Stamelos, "A methodology to assess the impact of design patterns on software quality," *Information and Software Technology,* vol. 54, pp. 331-346, 2012.

[205] J. M. Bieman, G. Straw, H. Wang, P. W. Munger, and R. T. Alexander, "Design Patterns and Change Proneness: An Examination of Five Evolving Systems," *Proceedings of the 9th International Symposium on Software Metrics*, p. 40, 2003.

[206] D. Jain and H. J. Yang, "OO Design Patterns, Design Structure, and Program Changes: An Industrial Case Study," *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, p. 580, 2001.

[207] M. Di Penta, L. Cerulo, Y. G. Gueheneuc, and G. Antoniol, "An empirical study of the relationships between design pattern roles and class change proneness," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pp. 217-226, 2008.

[208] D. Posnett, C. Bird, and P. Dévanbu, "An empirical study on the influence of pattern roles on change-proneness," *Empirical Software Enggineering,* vol. 16, pp. 396-423, 2011.

[209] K. O. Elish and M. Alshayeb, "A Classification of Refactoring Methods Based on Software Quality Attributes," *Arabian Journal for Science and Engineering,* vol. 36, pp. 1253-1267, 2011.

[210] M. Mohamed, M. Romdhani, and K. Ghedira, "Classification of model refactoring approaches," *Journal of Object Technology,* vol. 8, pp. 121-126, 2009.

[211] M. Katić and K. Fertalj, "Challenges and Discussion of Software Redesign," in *The 4th International Conference on Information Technology*, pp. 1-7, Amman, Jordan, 2009.

[212] A. Corradini, U. Montanari, and F. Rossi, "Graph processes," *Fundam. Inf.,* vol. 26, pp. 241-265, 1996.

[213] T. Mens, "On the Use of Graph Transformations for Model Refactoring," in *Generative and Transformational Techniques in Software Engineering*. vol. 4143, R. Lämmel, J. Saraiva, and J. Visser, Berlin / Heidelberg: Springer, pp. 219-257, 2006.

[214] L. Grunske, L. Geiger, A. Zündorf, N. Eetvelde, P. Gorp, and D. Varró, "Using Graph Transformation for Practical Model-Driven Software Engineering," in *Model-Driven Software Development*, S. Beydeda, M. Book, and V. Gruhn, Berlin Heidelberg: Springer, pp. 91-117, 2005.

[215] T. Baar and S. Marković, "A Graphical Approach to Prove the Semantic Preservation of UML/OCL Refactoring Rules," in *Perspectives of Systems Informatics*. vol. 4378, I. Virbitskaite and A. Voronkov, Berlin / Heidelberg: Springer, pp. 70-83, 2007.

[216] A. Folli and T. Mens, "Refactoring of UML models using AGG," *Proceedings of the Third International ERCIM Symposium on Software Evolution*, 2007.

[217] A. Moeini, V. Rafe, and F. Mahdian, "An approach to refactoring legacy systems," in *3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)*, pp. 5-8, 2010.

[218] H. Kazato, M. Takaishi, T. Kobayashi, and M. Saeki, "Formalizing refactoring by using graph transformation," *IEICE Transactions on Information and Systems,* vol. E87-D, pp. 855-867, 2004.

[219] M. Saeki and H. Kaiya, "Model Metrics and Metrics of Model Transformation," in *ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems*, 2006.

[220] C. Ermel, H. Ehrig, and K. Ehrig, "Refactoring of Model Transformations," *Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques*, 2009.

[221] R. Gheyi, T. Massoni, and P. Borba, "A Rigorous Approach for Proving Model Refactorings," *20th IEEE/ACM International Conference on Automated Software Engineering*, 2005.

[222] T. Massoni, R. Gheyi, and P. Borba, "Formal Refactoring for UML Class Diagrams," in *19th Brazilian Symposium on Software Engineering (SBES)*, pp. 152-167, 2005.

[223] T. Massoni, R. Gheyi, and P. Borba, "Formal Model-Driven Program Refactoring," in *Fundamental Approaches to Software Engineering*. vol. 4961, J. Fiadeiro and P. Inverardi, Berlin / Heidelberg: Springer, pp. 362-376, 2008.

[224] T. Massoni, R. Gheyi, and P. Borba, "Synchronizing Model and Program Refactoring," in *Formal Methods: Foundations and Applications*. vol. 6527, J. Davies, L. Silva, and A. Simao, Berlin / Heidelberg: Springer, pp. 96-111, 2011.

[225] H. C. Estler and H. Wehrheim, "Alloy as a Refactoring Checker?," *Electronic Notes in Theoretical Computer Science,* vol. 214, pp. 331-357, 2008.

[226] H. C. Estler, T. Ruhroth, and H. Wehrheim, "Modelchecking Correctness of Refactorings - Some Experiments," *Electronic Notes in Theoretical Computer Science,* vol. 187, pp. 3-17, 2007.

[227] J. Derrick and H. Wehrheim, "Model Transformations Incorporating Multiple Views," in *Algebraic Methodology and Software Technology*. vol. 4019, M. Johnson and V. Vene, Berlin / Heidelberg: Springer, pp. 111-126, 2006.

[228] G. Spanoudakis and A. Zisman, "Inconsistency Management in Software Engineering: Survey and Open Research Issues," in *Handbook of Software Engineering and Knowledge Engineering*. vol. 1, S. K. Chang: World Scientific Publishing Co., pp. 329-380, 2001.

[229] R. Van Der Straeten, V. Jonckers, and T. Mens, "Supporting Model Refactorings Through Behaviour Inheritance Consistencies," in *<<UML>> 2004 - The Unified Modeling Language. Modelling Languages and Applications*. vol. 3273, T. Baar,

A. Strohmeier, A. Moreira, and S. Mellor, Berlin / Heidelberg: Springer, pp. 305-319, 2004.

[230] R. Van Der Straeten and M. D'Hondt, "Model refactorings through rule-based inconsistency resolution," *Proceedings of the 2006 ACM symposium on Applied computing*, pp. 1210-1217, Dijon, France, 2006.

[231] R. Van Der Straeten, V. Jonckers, and T. Mens, "A formal approach to model refactoring and model refinement," *Software and Systems Modeling,* vol. 6, pp. 139-162, 2007.

[232] E. Saadeh, D. Kourie, and A. Boake, "Fine-grain transformations to refactor UML models," *Proceedings of the Warm Up Workshop for ACM/IEEE ICSE 2010*, pp. 45-51, Cape Town, South Africa, 2009.

[233] G. Sunyé, D. Pollet, Y. Le Traon, and J.-M. Jézéquel, "Refactoring UML Models," in *«UML» 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. vol. 2185 Berlin: Springer-Verlag, pp. 134-148, 2001.

[234] T. Millan, L. Sabatier, T.-T. Le Thi, P. Bazex, and C. Percebois, "An OCL extension for checking and transforming UML models," *Proceedings of the 8th WSEAS International Conference on Software engineering, parallel and distributed systems*, pp. 144-149, Cambridge, UK, 2009.

[235] S. Markovic and T. Baar, "Refactoring OCL Annotated UML Class Diagrams," *Software and Systems Modeling,* vol. 7, pp. 25-47, 2008.

[236] M. Stolc and I. Polasek, "A visual based framework for the model refactoring techniques," in *IEEE 8th International Symposium on Applied Machine Intelligence and Informatics*, pp. 72-82, 2010.

[237] A. Correa and C. Werner, "Applying Refactoring Techniques to UML/OCL Models," in *The Unified Modeling Language. Modelling Languages and*

Applications. vol. 3273, T. Baar, A. Strohmeier, A. Moreira, and S. Mellor, Berlin / Heidelberg: Springer, pp. 173-187, 2004.

[238] A. Correa and C. Werner, "Refactoring object constraint language specifications," *Software and Systems Modeling,* vol. 6, pp. 113-138, 2007.

[239] A. Correa, C. Werner, and M. Barros, "An Empirical Study of the Impact of OCL Smells and Refactorings on the Understandability of OCL Specifications," in *Model Driven Engineering Languages and Systems*. vol. 4735, G. Engels, B. Opdyke, D. Schmidt, and F. Weil, Berlin / Heidelberg: Springer, pp. 76-90, 2007.

[240] A. Correa, C. Werner, and M. Barros, "Refactoring to improve the understandability of specifications written in object constraint language," *IET Software,* vol. 3, pp. 69-90, 2009.

[241] R. Gheyi, T. Massoni, P. Borba, and A. Sampaio, "A Complete Set of Object Modeling Laws for Alloy," in *Formal Methods: Foundations and Applications*. vol. 5902, M. Oliveira and J. Woodcock, Berlin / Heidelberg: Springer, pp. 204-219, 2009.

[242] A. Christoph, "Describing Horizontal Model Transformations with Graph Rewriting Rules," in *Model Driven Architecture*. vol. 3599, U. Aßmann, M. Aksit, and A. Rensink, Berlin / Heidelberg: Springer, pp. 901-901, 2005.

[243] C. Junbing, W. Zhijian, C. Bo, and Q. Si, "Towards A Model Refactoring Conflict Resolution Algorithm," in *1st International Conference on Information Science and Engineering (ICISE2009)*, pp. 5439-5442, 2009.

[244] P. Bottoni, F. Presicce, and G. Taentzer, "Specifying Integrated Refactoring with Distributed Graph Transformations," in *Applications of Graph Transformations with Industrial Relevance*. vol. 3062, J. Pfaltz, M. Nagl, and B. Böhlen, Berlin / Heidelberg: Springer, pp. 220-235, 2004.

[245] G. Rangel, L. Lambers, B. König, H. Ehrig, and P. Baldan, "Behavior Preservation in Model Refactoring Using DPO Transformations with Borrowed

Contexts," in *Graph Transformations*. vol. 5214, H. Ehrig, R. Heckel, G. Rozenberg, and G. Taentzer, Berlin / Heidelberg: Springer, pp. 242-256, 2008.

[246] C. Amelunxen and A. Schürr, "Formalising model transformation rules for UML/MOF 2," *Software, IET,* vol. 2, pp. 204-222, 2008.

[247] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz, "A meta-model for language-independent refactoring," in *Proceedings. International Symposium on Principles of Software Evolution*, pp. 154-164, 2000.

[248] P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer, "Towards Automating Source-Consistent UML Refactorings," in *«UML» 2003 - The Unified Modeling Language. Modeling Languages and Applications*. vol. 2863, P. Stevens, J. Whittle, and G. Booch, Berlin / Heidelberg: Springer, pp. 144-158, 2003.

[249] N. Moha, A. Rouane Hacene, P. Valtchev, and Y.-G. Guéhéneuc, "Refactorings of Design Defects Using Relational Concept Analysis," in *Formal Concept Analysis*. vol. 4933, R. Medina and S. Obiedkov, Berlin / Heidelberg: Springer, pp. 289-304, 2008.

[250] N. Maneerat and P. Muenchaisri, "Bad-smell prediction from software design model using machine learning techniques," in *Eighth International Joint Conference on Computer Science and Software Engineering*, pp. 331-336, 2011.

[251] D. Astels, "Refactoring with UML," in *Proceedings of International Conference eXtreme Programming and Flexible Processes in Software Engineering*, pp. 67-70, 2002.

[252] D.-K. Kim, R. France, S. Ghosh, and S. Eunjee, "A role-based metamodeling approach to specifying design patterns," in *Proceedings. 27th Annual International Computer Software and Applications Conference*, pp. 452-457, 2003.

[253] D.-K. Kim, "Software Quality Improvement via Pattern-Based Model Refactoring," in *11th IEEE High Assurance Systems Engineering Symposium*, pp. 293-302, 2008.

[254] D.-K. Kim and C. El Khawand, "An approach to precisely specifying the problem domain of design patterns," *Journal of Visual Languages and Computing,* vol. 18, pp. 560-591, 2007.

[255] D. Ballis, A. Baruzzo, and M. Comini, "A Rule-based Method to Match Software Patterns Against UML Models," *Electronic Notes in Theoretical Computer Science,* vol. 219, pp. 51-66, 2008.

[256] D. Ballis, A. Baruzzo, and M. Comini, "A Minimalist Visual Notation for Design Patterns and Antipatterns," in *Fifth International Conference on Information Technology: New Generations*, pp. 51-56, 2008.

[257] G. El-Boussaidi and H. Mili, "Detecting Patterns of Poor Design Solutions Using Constraint Propagation," in *Model Driven Engineering Languages and Systems*. vol. 5301, K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, Berlin / Heidelberg: Springer, pp. 189-203, 2008.

[258] M. El-Sharqwi, H. Mahdi, and I. El-Madah, "Pattern-based model refactoring," *International Conference on Computer Engineering and Systems (ICCES)*, pp. 301-306, 2010.

[259] C. Bouhours, H. Leblanc, and C. Percebois, "Bad smells in design and design patterns," *Journal of Object Technology,* vol. 8, pp. 43-63, 2009.

[260] M. Akiyama, S. Hayashi, T. Kobayashi, and M. Saeki, "Supporting Design Model Refactoring for Improving Class Responsibility Assignment," in *Model Driven Engineering Languages and Systems*. vol. 6981, J. Whittle, T. Clark, and T. Kühne, Berlin / Heidelberg: Springer, pp. 455-469, 2011.

[261] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd ed. Englewood Cliffs: Prentice Hall, 2005.

[262] Ł. Dobrzański and L. Kuźniarz, "An approach to refactoring of executable UML models," *Proceedings of the 2006 ACM symposium on Applied computing*, pp. 1273-1279, Dijon, France, 2006.

[263] H. Liu, Z. Ma, L. Zhang, and W. Shao, "Detecting Duplications in Sequence Diagrams Based on Suffix Trees," in *13th Asia Pacific Software Engineering Conference*, pp. 269-276, Kanpur, India, 2006.

[264] M. El-Attar and J. Miller, "Improving the quality of use case models using antipatterns," *Software and Systems Modeling,* vol. 9, pp. 141-160, 2010.

[265] L. Favre and C. Pereira, "Formalizing MDA-Based Refactorings," *19th Australian Conference on Software Engineering*, pp. 377-386, 2008.

[266] K. Lano and D. Clark, "Model Transformation Specification and Verification," in *The Eighth International Conference on Quality Software*, pp. 45-54, 2008.

[267] I. Porres, "Model Refactorings as Rule-Based Update Transformations," in *«UML» 2003 - The Unified Modeling Language. Modeling Languages and Applications*. vol. 2863, P. Stevens, J. Whittle, and G. Booch, Berlin / Heidelberg: Springer, pp. 159-174, 2003.

[268] I. Porres, "Rule-based update transformations and their application to model refactorings," *Software and Systems Modeling,* vol. 4, pp. 368-385, 2005.

[269] J. Zhang, Y. Lin, and J. Gray, "Generic and Domain-Specific Model Refactoring Using a Model Transformation Engine," in *Model-Driven Software Development*, S. Beydeda, M. Book, and V. Gruhn, Berlin Heidelberg: Springer, pp. 199-217, 2005.

[270] W. Yu, J. Li, and G. Butler, "Refactoring Use Case Models on Episodes," in *19th IEEE International Conference on Automated Software Engineering (ASE'04)*, pp. 328-331, 2004.

[271] Y. Kim and K.-G. Doh, "The Service Modeling Process Based on Use Case Refactoring," in *Business Information Systems*. vol. 4439, W. Abramowicz, Berlin / Heidelberg: Springer, pp. 108-120, 2007.

[272] P. Bottoni, F. Parisi-Presicce, and G. Taentzer, "Specifying Coherent Refactoring of Software Artefacts with Distributed Graph Transformations," in *Transformation of Knowledge, Information and Data: Theory and Applications*, P. van Bommel, Hershey, PA: Information Science Publishing, 2005.

[273] S. Hosseini and M. A. Azgomi, "UML Model Refactoring with Emphasis on Behavior Preservation," *Proceedings of the 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*, pp. 125-128, 2008.

[274] C. A. R. Hoare, *Communicating Sequential Processes*: Prentice Hall, 1985.

[275] A. W. Roscoe, *The Theory and Practice of Concurrency*: Prentice Hall, 1997.

[276] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "SAL 2," in *Computer Aided Verification*. vol. 3114, R. Alur and D. Peled: Springer Berlin / Heidelberg, pp. 251-254, 2004.

[277] L. S. Barbosa and M. Sun, "UML Model Refactoring as Refinement: A Coalgebraic Perspective," *10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pp. 340-347, 2008.

[278] R. Gheyi, T. Massoni, and P. Borba, "Type-safe Refactorings for Alloy," *Proceedings of the 8th Brazilian Symposium on Formal Methods*, pp. 174-190, Porto Alegre, Brazil, 2005.

[279] M. Boger, T. Sturm, and P. Fragemann, "Refactoring Browser for UML," in *Objects, Components, Architectures, Services, and Applications for a Networked*

World. vol. 2591, M. Aksit, M. Mezini, and R. Unland, Berlin / Heidelberg: Springer, pp. 366-377, 2003.

[280] J. Philipps and B. Rumpe, "Roots of Refactoring," in *Tenth OOPSLA Workshop on Behavioral Semantics*, pp. 187-199, 2001.

[281] F. L. Bauer and C. Language Group, *The Munich Project CIP, Vol. 1: The Wide Spectrum Language CIP-L* vol. 2: Springer - Verlag, 1985.

[282] K. Rui and G. Butler, "Refactoring Use Case Models: The Metamodel," in *Proceedings of the 25th Australasian Computer Society Conference*, pp. 301-308, 2003.

[283] B. Regnell, "Requirements Engineering with Use Cases - a Basis for Software Development," Lund University, 1999.

[284] S. Ren, G. Butler, K. Rui, J. Xu, W. Yu, and R. Luo, "A prototype tool for use case refactoring," in *Proceedings of the 6th International Conference on Enterprise Information Systems*, pp. 173–178 2004.

[285] J. Xu, W. Yu, K. Rui, and G. Butler, "Use Case Refactoring: A Tool and a Case Study," in *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, pp. 484-491, Washington, DC, USA, 2004.

[286] K. Rui, "Refactoring use case models," PhD Thesis, Concordia University, 2007.

[287] H. Kim and C. Boldyreff, "Developing Software Metrics Applicable to UML Models," in *6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2002.

[288] R. Gronback. (2004, May 2012). *Model Validation: Applying Audits and Metrics to UML Models*. Available: http://conferences.embarcadero.com/jp/article/32089

[289] S. W. Ambler, *The Elements of UML 2.0 Style*. New York: Cambridge University Press, 2005.

[290] S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering,* vol. 7, pp. 510-518, 1981.

[291] C. Lange, "Model Size Matters," in *Models in Software Engineering*. vol. 4364, T. Kühne, Berlin / Heidelberg: Springer, pp. 211-216, 2007.

[292] I. O. f. S. I. I. E. C. (IEC), "Software Engineering-Product Quality," ISO/IEC Standard No. 9126, 2001-2004.

[293] J. McCall, P. Richards, and G. Walters, "Factors in Software Quality," US Rome Air Development Center, RADC TR-77-369,Available: http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA049014, 1977.

[294] A. A. Jalbani, J. Grabowski, H. Neukirchen, and B. Zeiss, "Towards an integrated quality assessment and improvement approach for UML models," *Proceedings of the 14th international SDL conference on Design for motes and mobiles*, pp. 63-81, Bochum, Germany, 2009.

[295] H. Voigt and T. Ruhroth, "A Quality Circle Tool for Software Models," in *Conceptual Modeling - ER 2008*. vol. 5231, Q. Li, S. Spaccapietra, E. Yu, and A. Olivé, Berlin / Heidelberg: Springer, pp. 526-527, 2008.

[296] AGG. (2011). The Attributed Graph Grammar System. Online: http://user.cs.tu-berlin.de/~gragra/agg/

[297] Fujaba. (2011). Fujaba Tool Suite. Online: http://www.fujaba.de/

[298] L. Geiger and A. Zündorf, "Statechart Modeling with Fujaba," *Electronic Notes in Theoretical Computer Science,* vol. 127, pp. 37-49, 2005.

[299] W. M. Ho, J. M. Jezequel, A. Le Guennec, and F. Pennaneac'h, "UMLAUT: an extendible UML transformation framework," in *14th IEEE International Conference on Automated Software Engineering*, pp. 275-278, Florida, USA, 1999.

[300] J. Oldevik. (2005). UMT-QVT Tool. Online: http://umt-qvt.sourceforge.net/

[301] M. Peltier, J. Bézivin, and G. Guillaume, "MTRANS: A general framework based on XSLT for model transformations," in *Workshop on Transformations in UML (WTUML01)*, Genova, Italy, 2001.

[302] D. Li, X. Li, and V. Stolz, "QVT-based model transformation using XSLT," *SIGSOFT Software Enggineering Notes,* vol. 36, pp. 1-8, 2011.

[303] F. J. Lucas, F. Molina, and A. Toval, "A systematic review of UML model consistency management," *Information and Software Technology,* vol. 51, pp. 1631-1645, 2009.

[304] D. Chiorean, M. Paşca, A. Cârcu, C. Botiza, and S. Moldovan, "Ensuring UML Models Consistency Using the OCL Environment," *Electronic Notes in Theoretical Computer Science,* vol. 102, pp. 99-110, 2004.

[305] G. Spanoudakis and H. Kim, "Diagnosis of the significance of inconsistencies in object-oriented designs: a framework and its experimental evaluation," *Journal of Systems and Software,* vol. 64, pp. 3-22, 2002.

[306] G. Spanoudakis, K. Kasis, and F. Dragazi, "Evidential diagnosis of inconsistencies in object-oriented designs," *International Journal of Software Engineering and Knowlege Enggineering,* vol. 14, pp. 141-178, 2004.

[307] R. F. Paige, D. S. Kolovos, and F. A. C. Polack, "Refinement via Consistency Checking in MDA," *Electronic Notes in Theoretical Computer Science,* vol. 137, pp. 151-161, 2005.

[308] P. G. Sapna and H. Mohanty, "Ensuring Consistency in Relational Repository of UML Models," in *Proceedings of the 10th International Conference on Information Technology*, pp. 217-222, 2007.

[309] B. Graaf and A. van Deursen, "Model-Driven Consistency Checking of Behavioural Specifications," in *Fourth International Workshop on Model-Based*

Methodologies for Pervasive and Embedded Software (MOMPES '07) pp. 115-126, 2007.

[310] A. Egyed, "Consistent Adaptation and Evolution of Class Diagrams during Refinement," in *Fundamental Approaches to Software Engineering*. vol. 2984, M. Wermelinger and T. Margaria-Steffen, Berlin / Heidelberg: Springer pp. 37-53, 2004.

[311] A. Egyed, "Fixing Inconsistencies in UML Design Models," *Proceedings of the 29th international conference on Software Engineering*, pp. 292-301, 2007.

[312] T. Mens, R. Van Der Straeten, and M. D'Hondt, "Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis," in *Model Driven Engineering Languages and Systems*. vol. 4199, O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Berlin / Heidelberg: Springer pp. 200-214, 2006.

[313] J. M. Küster, "Towards Inconsistency Handling of Object-Oriented Behavioral Models," *Electronic Notes in Theoretical Computer Science,* vol. 109, pp. 57-69, 2004.

[314] L. Fryz and L. Kotulski, "Assurance of System Consistency During Independent Creation of UML Diagrams," *Proceedings of the 2nd International Conference on Dependability of Computer Systems*, pp. 51-58, 2007.

[315] J. H. Hausmann and R. Heckel, "Extended model relations with graphical consistency condition," *Proceedings UML 2002 Workshop on Consistency Problems in UML-based Software Development*, pp. 61-74, Blekinge Institute of Technology, 2002.

[316] R. Wagner, H. Giese, and U. A. Nickel, "A plug-in for flexible and incremental consistency management," *Proceedings of the International Conference on the Unified Modeling Language (Workshop 7: Consistency Problems in UML-based Software Development)*, San Francisco, USA, 2003.

[317] N. Amálio, S. Stepney, and F. Polack, "Formal Proof from UML Models," in *Formal Methods and Software Engineering*. vol. 3308, J. Davies, W. Schulte, and M. Barnett, Berlin / Heidelberg: Springer pp. 418-433, 2004.

[318] K. van Hee, N. Sidorova, L. Somers, and M. Voorhoeve, "Consistency in model integration," *Data and Knowledge Engineering,* vol. 56, pp. 4-22, 2006.

[319] M. Schrefl and M. Stumptner, "Behavior-consistent specialization of object life cycles," *ACM Transactions on Software Enggineering Methodology,* vol. 11, pp. 92-148, 2002.

[320] Y. Shinkawa, "Inter-Model Consistency in UML Based on CPN Formalism," *Proceedings of the XIII Asia Pacific Software Engineering Conference*, pp. 411-418, 2006.

[321] S. Yao and S. M. Shatz, "Consistency Checking of UML Dynamic Models Based on Petri Net Techniques," *Proceedings of the 15th International Conference on Computing*, pp. 289-297, 2006.

[322] D. Kholkar, G. M. Krishna, U. Shrotri, and R. Venkatesh, "Visual specification and analysis of use cases," *Proceedings of the 2005 ACM symposium on Software visualization*, pp. 77-85, St. Louis, Missouri, 2005.

[323] R. Laleau and F. Polack, "Using formal metamodels to check consistency of functional views in information systems specification," *Information and Software Technology,* vol. 50, pp. 797-814, 2008.

[324] D. Ossami, J.-P. Jacquot, and J. Souquières, "Consistency in UML and B Multi-view Specifications," in *Integrated Formal Methods*. vol. 3771, J. Romijn, G. Smith, and J. van de Pol, Berlin / Heidelberg: Springer pp. 386-405, 2005.

[325] W. L. Yeung, "Checking Consistency between UML Class and State Models Based on CSP and B," *Journal of Universal Computer Science,* vol. 10, pp. 1540-1559, 2004.

[326] V. Lam and J. Padget, "Consistency Checking of Sequence Diagrams and Statechart Diagrams Using the pi-Calculus," in *Integrated Formal Methods*, pp. 347-365, 2005.

[327] H. Malgouyres and G. Motet, "A UML model consistency verification approach based on meta-modeling formalization," *Proceedings of the 2006 ACM symposium on Applied computing*, pp. 1804-1809, Dijon, France, 2006.

[328] R. F. Paige, P. J. Brooke, and J. S. Ostroff, "Metamodel-based model conformance and multiview consistency checking," *ACM Transactions on Software Enggineering Methodology,* vol. 16, p. 11, 2007.

[329] R. F. Paige, L. Kaminskaya, J. S. Ostroff, and J. Lancaric, "BON-CASE: an extensible CASE tool for formal specification and reasoning," *Journal of Object Technology,* vol. 1, pp. 77-96, 2002.

[330] H. Rasch and H. Wehrheim, "Checking Consistency in UML Diagrams: Classes and State Machines," in *Formal Methods for Open Object-Based Distributed Systems*. vol. 2884, E. Najm, U. Nestmann, and P. Stevens, Berlin / Heidelberg: Springer pp. 229-243, 2003.

[331] H. Wang, T. Feng, J. Zhang, and K. Zhang, "Consistency check between behaviour models," in *IEEE International Symposium on Communications and Information Technology*, pp. 486-489, 2005.

[332] X. Zhao, Q. Long, and Z. Qiu, "Model checking dynamic UML consistency," *Proceedings of the 8th international conference on Formal Methods and Software Engineering*, pp. 440-459, Macao, China, 2006.

[333] R. Van Der Straeten, T. Mens, J. Simmonds, and V. Joncker, "Using Description Logic to Maintain Consistency between UML Models," in *"UML" 2003 - The Unified Modeling Language*. vol. 2863 Berlin / Heidelberg: Springer, pp. 326-340, 2004.

[334] P. Inverardi, H. Muccini, and P. Pelliccione, "Automated Check of Architectural Models Consistency Using SPIN," *Proceedings of the 16th IEEE international conference on Automated software engineering*, p. 346, 2001.

[335] F. J. Lucas and A. Toval, "A precise approach for the analysis of the UML models consistency," *1st International Workshop on Best Practices of UML part of 24th International Conference on Conceptual Modeling (ER 2005)*, Klagenfurt (Austria), 2005.

[336] T. Massoni, "Introducing Refactoring to Heavyweight Software Processes," Federal University of Pernambuco, Brazil, Technical Report,Available: http://www.cin.ufpe.br/~tlm/download/article-refactoring-RUP.pdf, 2003.

[337] M. Fontoura, W. Pree, and B. Rumpe, "UML-F: A Modeling Language for Object-Oriented Frameworks," in *ECOOP 2000 - Object-Oriented Programming*. vol. 1850 Berlin / Heidelberg: Springer, pp. 63-82, 2000.

[338] M. M. Kandé and A. Strohmeier, "Towards a UML Profile for Software Architecture Descriptions," in *Proceedings of the 3rd international conference on The unified modeling language: advancing the standard* pp. 513-527, 2000.

[339] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins, "Modeling Software Architectures in the Unified Modeling Language," *ACM Transactions on Software Engineering and Methodology,* vol. 11, pp. 2-57, 2002.

[340] M. H. Kacem, A. H. Kacem, M. Jmaiel, and K. Drira, "Describing dynamic software architectures using an extended UML model," in *Proceeding of the 2006 ACM Symposium on Applied Computing*, pp. 1245-1249, 2006.

[341] G. Wagner, "A UML Profile for Agent-Oriented Modeling," in *Proceedings of the 3rd International Workshop on Agent-Oriented Software Engineering*, 2002.

[342] C. Hahn and I. Slomic, "Agent-based Extensions for the UML Profile and Metamodel for Service-oriented Architectures," in *12th Enterprise Distributed Object Computing Conference Workshops*, pp. 309-316, 2008.

[343] V. T. da Silva and C. J. de Lucena, "From a Conceptual Framework for Agents and Objects to a Multi-Agent System Modeling Language," *Autonomous Agents and Multi-Agent Systems,* vol. 9, pp. 145-189, 2004.

[344] S. Clarke, "Extending the UML Metamodel for Design Composition," in *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, 2000.

[345] A. A. Zakaria, H. Hosny, and A. Zeid, "A UML Extension for Modeling Aspect-Oriented Systems," in *2nd Workshop on Aspect-Oriented Modeling with UML*, 2002.

[346] A. Przybylek, "Separation of Crosscutting Concerns at the Design Level: an Extension to the UML Metamodel," in *Proceedings of the International Multiconference on Computer Science and Information Technology*, pp. 551-557, 2008.

[347] S. Robak, B. Franczyk, and K. Politowicz, "Extending the UML for Modelling variability for System Families," *International Journal of Applied Math and Computer Science,* vol. 12, pp. 285-298, 2002.

[348] B. Korherr and B. List, "Extending the UML 2 Activity Diagram with Business Process Goals and Performance Measures and the Mapping to BPEL," in *Advances in Conceptual Modeling - Theory and Practice*. vol. 4231 Berlin / Heidelberg: Springer, pp. 7-18, 2006.

[349] B. Vela and E. Marcos, "Extending UML to represent XML Schemas," in *The 15th Conference On Advanced Information Systems Engineering*, pp. 97-100, 2003.

[350] J. Jürjens, "UMLsec: Extending UML for Secure Systems Development," in *Proceedings of the 5th International Conference on The Unified Modeling Language*, pp. 412-425, London, UK, 2002.

[351] L. Baresi, F. Garzotto, and P. Paolini, "Extending UML for Modeling Web Applications," in *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, p. 3055, Washington, DC, USA, 2001.

[352] V. T. da Silva and C. J. de Lucena, "Extending the UML Sequence Diagram to model the dynamic aspects of Multi-Agent Systems," PUC-Rio, Rio de Janeiro, Brazil, MCC 15/03,Available: ftp://ftp.inf.puc-rio.br/pub/docs/techreports/03_15_silva.pdf, 2003.

[353] G. Padilla, M. A. Serrano, and C. Montes de Oca, "A UML Sequence Diagram Extension to Handle Multiplicities," in *Fifth Mexican International Conference in Computer Science*, pp. 80-87, Los Alamitos, CA 2004.

[354] D. Harel and S. Maoz, "Assert and negate revisited: Modal semantics for UML sequence diagrams," *Software and Systems Modeling,* vol. 7, pp. 237-252, 2008.

[355] A. Refsdal and K. Stolen, "Extending UML Sequence Diagrams to Model Trust-dependent Behavior With the Aim to Support Risk Analysis," *Science of Computer Programming,* vol. 74, pp. 34-42, 2008.

[356] O. Haugen, K. E. Husa, R. K. Runde, and K. Stolen, "STAIRS towards formal design with sequence diagrams," *Software and Systems Modeling,* vol. 4, pp. 355-357, 2005.

[357] A. Durán, A. Ruiz-Cortés, R. Corchuelo, and M. Toro, "Supporting Requirements Verification Using XSLT," in *IEEE International Requirements Engineering Conference*, pp. 165-172, Essen, Germany, 2002.

[358] I. Díaz, F. Losavio, A. Matteo, and O. Pastor, "A specification pattern for use cases," *Journal of Information and Management,* vol. 41, pp. 961-975, 2004.

[359] P. Metz, J. O'Brien, and W. Weber, "Specifying Use Case Interaction: Types of Alternative Courses," *Journal of Object Technology,* vol. 2, pp. 111-131, 2003.

[360] A. Bragança and R. J. Machado, "Extending UML 2.0 Metamodel for Complementary Usages of the <<extend>> Relationship within Use Case Variability Specification," in *10th International Software Product Line Conference*, pp. 123-130, Baltimore, USA, 2006.

[361] V. Hoffmann, H. Lichter, A. Nyáen, and A. Walter, "Towards the Integration of UML and textual Use Case Modeling," *Journal of Object Technology,* vol. 8, pp. 85-100, 2009.

[362] L. u. Zelinka and V. Vrani´c, "A Configurable UML Based Use Case Modeling Metamodel," in *First IEEE Eastern European Conference on the Engineering of Computer Based Systems*, pp. 1-8, Washington, DC, 2009.

[363] S. S. Somé, "A Meta-Model for Textual Use Case Description," *Journal of Object Technology,* vol. 8, pp. 87-106, 2009.

[364] J. Repond, P. Dugerdil, and P. Descombes, "Use-Case and Scenario Metamodeling for Automated Processing in a Reverse Engineering Tool," in *4th India Software Engineering Conference*, pp. 135-144, New York, 2011.

[365] B. Henderson-Sellers, "Who needs an OO methodology anyway?," *Journal of Object Oriented Programming,* vol. 8, pp. 6-8, 1995.

[366] C. Gonzalez-Perez and B. Henderson-Sellers, *Metamodelling for Software Engineering*: Wiley Publishing, 2008.

[367] S. Meng and B. K. Aichernig, "Towards a Coalgebraic Semantics of UML: Class Diagrams and Use Cases," The United Nations University / International Institute for Software Technology, Technical Report 272,Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.2628, 2003.

[368] M. Misbhauddin and M. Alshayeb, "Extending the UML Metamodel for Sequence Diagram to Enhance Model Traceability," in *Proceedings of the 2010 Fifth International Conference on Software Engineering Advances* pp. 129-134, France, 2010.

[369] J. M. Almendros-Jiménez and L. Iribarne, "Describing Use-Case Relationships with Sequence Diagrams," *The Computer Journal,* vol. 50, pp. 116-128, 2007.

[370] L. Li, "Translating Use Cases to Sequence Diagrams," in *Proceedings of the 15th IEEE international conference on Automated software engineering*, pp. 293-296, Grenoble , France, 2000.

[371] T. Yue, L. C. Briand, and Y. Labiche, "Automatically Deriving UML Sequence Diagrams from Use Cases," *Simula Research Laboratory,* Carleton University, Canada, TR-SCE-10-03,Available: http://squall.sce.carleton.ca/pubs/tech_report/TR-SCE-10-03.pdf, 2010.

[372] J. Gutiérrez, C. Nebut, M. Escalona, M. Mejías, and I. Ramos, "Visualization of Use Cases through Automatically Generated Activity Diagrams," in *Model Driven Engineering Languages and Systems*. vol. 5301, K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, Berlin / Heidelberg: Springer, pp. 83-96, 2008.

[373] M. Lei and W. C. Jiang, "Research on Activity Based Use Case Meta-Model," in *International Conference on Advanced Computer Theory and Engineering*, pp. 843-846, 2008.

[374] T. Nakatani, T. Urai, S. Ohmura, and T. Tamai, "A requirements description metamodel for use cases," in *Proceedings of the Eighth Asia-Pacific on Software Engineering Conference*, p. 251, 2001.

[375] R.-J. Back, L. Petre, and I. Porres, "Formalising UML use cases in the refinement calculus," *TUCS Technical Reports,* Turku Centre for Computer Science, Turku, Finland, TUCS-TR-279,Available: http://tucs.fi/publications/view/?pub_id=tBaPePa99a, 1999.

[376] D. Dranidis, K. Tigka, and P. Kefalas, "Formal modelling of use cases with X-machines," in *Proceedings of the 1st South-East European Workshop on Formal Methods*, 2003.

[377] W. Grieskamp, M. Lepper, W. Schulte, and N. Tillmann, "Testable Use Cases in the Abstract State Machine Language," in *Proceedings of the Second Asia-Pacific Conference on Quality Software*, p. 167, Washington, D.C., 2001.

[378] W. Grieskamp and M. Lepper, "Using Use Cases in Executable Z," in *Proceedings of the 3rd IEEE International Conference on Formal Engineering Methods*, p. 111, Washington, DC, USA, 2000.

[379] A. Cockburn, *Writing Effective Use Cases*: Addison-Wesley, 2000.

[380] D. Kulak and E. Guiney, *Use Cases: Requirements in Context* vol. 2 nd Boston, MA, USA: Addison-Wesley Professional, 2003.

[381] P. Kruchten, *The Rational Unified Process: An Introduction* vol. 2nd: Addison-Wesley, 2000.

[382] J. C. Leite, J. Doorn, G. Hadad, J. H. Doorn, and G. Kaplan, "A Scenario Construction Process," *Requirements Engineering Journal,* vol. 5, pp. 38-61, 2000.

[383] A. D. Toro, B. B. Jiménez, A. R. Cortés, and M. T. Bonilla, "A Requirements Elicitation Approach Based in Templates and Patterns," in *Workshop em Engenharia de Requisitos*, pp. 17-29, 1999.

[384] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process* vol. 2nd. Upper Saddle River, NJ: Prentice Hall PTR, 2001.

[385] G. Schneider and J. P. Winters, *Applying Use Cases: A Practical Guide* vol. 2nd: Addison-Wesley, 2001.

[386] N. J. Nunes, "iUCP-Estimating Interaction Design Projects with Enhanced Use Case Points," in *Task Models and Diagrams for User Interface Design*. vol. 5963, D. England, P. Palanque, J. Vanderdonckt, and P. Wild, Berlin / Heidelberg: Springer, pp. 131-145, 2010.

[387] L. L. Constantine and A. D. Lockwood Lucy, *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*: Addison Wesley, Longman, 1999.

[388] S. Adolph, P. Bramble, A. Cockburn, and A. Pols, *Patterns for Effective Use Cases*: Addison-Wesley, 2003.

[389] F. Armour and G. Miller, *Advanced Use Case Modeling: Software Systems*: Addison-Wesley, 2001.

[390] J. Arlow and I. Neustadt, *UML and the Unified Process: practical object-oriented analysis and design*: Addison-Wesley, 2002.

[391] K. Bittner and I. Spence, *Use Case Modeling*: Addison-Wesley, 2003.

[392] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual* vol. 2nd: Addison-Wesley, 2005.

[393] M. Hilsbos, I.-Y. Song, and Y. Choi, "A Comparative Analysis of Use Case Relationships," in *Perspectives in Conceptual Modeling*. vol. 3770, J. Akoka, S. Liddle, I.-Y. Song, M. Bertolotto, I. Comyn-Wattiau, W.-J. van den Heuvel, M. Kolp, J. Trujillo, C. Kop, and H. Mayr, Berlin / Heidelberg: Springer, pp. 53-62, 2005.

[394] M. A. Laguna and J. M. Marqués, "On the Multiplicity Semantics of the Extend Relationship in Use Case Models," in *Software and Data Technologies*. vol. 47, J. Cordeiro, B. Shishkov, A. Ranchordas, and M. Helfert, Berlin Heidelberg: Springer, pp. 62-75, 2009.

[395] S. Diev, "Software estimation in the maintenance context," *SIGSOFT Software Engineering Notes,* vol. 31, pp. 1-8, 2006.

[396] W. M. Wilson, L. H. Rosenberg, and L. E. Hyatt, "Automated analysis of requirement specifications," in *19th International Conference on Software Engineering*, pp. 161-171, New York, 1997.

[397] A. Fantechi, S. Gnesi, G. Lami, and A. Maccari, "Application of Linguistic Techniques for Use Case Analysis," in *International Requirements Engineering Conference*, pp. 161-170, London Limited, 2003.

[398] V. Ambriola and V. Gervasi, "On the Systematic Analysis of Natural Language Requirements with CIRCE," *Automated Software Engineering,* vol. 13, pp. 107-167, 2006.

[399] S. Overmyer, B. Lavoie, and O. Rambow, "Conceptual modeling through linguistic analysis using LIDA," in *23rd International Conference on Software Engineering*, pp. 401-410, Washington, 2001.

[400] A. Sampaio, R. Chitchyan, A. Rashid, and P. Rayson, "EA-Miner: a tool for automating aspect-oriented requirements identification," in *International Conference on Automated Software Engineering*, pp. 352-355, New York, 2005.

[401] G. Fliedl, C. Kop, H. C. Mayr, A. Salbrechter, J. Vöhringer, G. Weber, and C. Winkler, "Deriving static and dynamic concepts from software requirements using sophisticated tagging," *Journal of Data & Knowledge Engineering,* vol. 61, pp. 433-448, 2007.

[402] C. Rolland and C. Ben Achour, "Guiding the construction of textual use case specifications," *Journal of Data & Knowledge Engineering,* vol. 25, pp. 125-160, 1998.

[403] A. Sinha, A. M. Paradkar, P. Kumanan, and B. Boguraev, "A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases," in *International Conference on Dependable Systems and Networks*, pp. 327-336, 2009.

[404] T. Yue, L. C. Briand, and Y. Labiche, "Automatically Deriving a UML Analysis Model from a Use Case Model," *Simula Research Laboratory,* Carleton University, Canada, TR SCE-09-09,Available: http://134.117.61.33/pubs/tech_report/TR-SCE-09-09.pdf, 2010.

[405] M. Ochodek and J. R. Nawrocki, "Automatic Transactions Identification in Use Cases," in *Second Central and East European Conference on Software Engineering Techniques*, pp. 55-68, 2011.

[406] S. S. Somé and D. K. Nair, "Use Case Based Requirements Verification - Verifying the consistency between use cases and assertions," in *9th International Conference on Enterprise Information Systems*, pp. 190-195, 2007.

[407] S. S. Somé, "Specifying Use Case Sequencing Constraints Using Description Elements," in *Sixth International Workshop on Scenarios and State Machines*, pp. 4-10, Washington, DC, 2007.

[408] S. S. Somé and X. Cheng, "An Approach for Supporting System-level Test Scenarios Generation from Textual Use Cases," in *ACM symposium on Applied computing*, pp. 724-729, New York, 2008.

[409] M. R. Braz and S. R. Vergilio, "Software Effort Estimation Based on Use Cases," in *30th Annual International Computer Software and Applications Conference*, pp. 221-228, Washington, DC, 2006.

[410] M. Misbhauddin and M. Alshayeb, "Extending the UML Use Case Metamodel with Behavioral Information to Facilitate Model Analysis and Interchange," Manuscript submitted for publication, King Fahd University (KFUPM), Saudi Arabia.

[411] F. Ramalho, J. Robin, and R. Barros, "XOCL - an XML Language for Specifying Logical Constraints in Object Oriented Models," *Journal of Universal Computer Science,* vol. 9, pp. 956-969, 2003.

[412] A. Tchertchago, "Formal Semantics for a UML fragment using UML/OCL metamodeling," in *Software Engineering and Applications* Cambridge, MA: ACTA Press, 2002.

[413] P. P. da Silva and N. W. Paton, "User Interface Modeling in UMLi," *IEEE Software,* vol. 20, pp. 62-69, 2003.

[414] A. Boronat, J. Á. Carsí, I. Ramos, and P. Letelier, "Formal Model Merging Applied to Class Diagram Integration," *Electronic Notes in Theoretical Computer Science,* vol. 166, pp. 5-26, 2007.

[415] P. Selonen and T. Systä, "Scenario-based Synthesis of Annotated Class Diagrams in UML," *Proceedings of OOPSLA 2000 Workshop: Scenario-based round-trip engineering*, pp. 26-31, 2000.

[416] R. B. Salem, R. Grangel, and J.-P. Bourey, "A comparison of model transformation tools: Application for Transforming GRAI Extended Actigrams into UML Activity Diagrams," *Computers in Industry,* vol. 59, pp. 682-693, 2008.

[417] W. Sun, E. Song, P. C. Grabow, and D. M. Simmonds, "Toward an Integrated Tool Environment for Static Analysis of UML Class and Sequence Models," *Journal of Universal Computer Science,* vol. 16, pp. 2435--2454, 2010.

[418] A. Staikopoulos and B. Bordbar, "A Comparative Study of Metamodel Integration and Interoperability in UML and Web Services," in *Model Driven Architecture – Foundations and Applications*. vol. 3748, A. Hartman and D. Kreische, Berlin / Heidelberg: Springer pp. 145-159, 2005.

[419] OMG, "Unified Modeling Language: Infrastructure," Version. 2.4.1, formal/2011-08-05, Object Management Group, 2011.

[420] C. Batini, M. Lenzerini, and S. B. Navathe, "A comparative analysis of methodologies for database schema integration," *ACM Computing Surveys,* vol. 18, pp. 323-364, 1986.

[421] R. A. Pottinger and P. A. Bernstein, "Merging models based on given correspondences," *Proceedings of the 29th international conference on Very large data bases*, pp. 862-873, Berlin, Germany, 2003.

[422] S. B. Chaouni, M. Fredj, and S. Mouline, "MDA based-approach for UML Models Complete Comparison," *International Journal of Computer Science Issues,* vol. 8, pp. 1-10, 2011.

[423] K. Lano, "Introduction to the Unified Modeling Language," in *UML 2 Semantics and Applications*, K. Lano, Hoboken, New Jersey: John Wiley and Sons Inc., 2009.

[424] M. Misbhauddin and M. Alshayeb, "Model-driven Refactoring Approaches —A Comparison Framework.," in *The African Conference on Software Engineering and Applied Computing*, Botswana, 2012.

[425] H. Liu and X. Jia, "Model Transformation Using a Simplified Metamodel," *Journal of Software Engineering and Applications,* vol. 3, pp. 653-660, 2010.

[426] G. Taentzer, K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, D. Varro, and S. Varro-Gyapay, "Model transformation by graph transformation: a comparative study," *International Workshop on Model Transformations in Practice*, 2005.

[427] T. Furche, F. Bry, S. Schaffert, R. Orsini, I. Horrocks, M. Kraus, and O. Bolzer, "Survey over Existing Query and Transformation Languages," *Reasoning on the Web with Rules and Semantics (REWERSE),* LudwigMaximiliansUniversität München, Munich, IST506779/Munich/I4D1/D/PU/a1,Available: http://rewerse.net/deliverables/m24/i4-d9a.pdf, 2004.

[428] G. Kniesel and H. Koch, "Static composition of refactorings," *Science of Computer Programming, Special Issue on Program Transformation,* vol. 52, pp. 9-51, 2004.

[429] M. Fowler, Use and Abuse Cases," *Distributed Computing*," 1998.

[430] M. Misbhauddin and M. Alshayeb, "Towards a Multi-view Approach to Model-based Refactoring," in *The African Conference on Software Engineering and Applied Computing*, Botswana, 2012.

[431] A. J. Riel, *Object-Oriented Design Heuristics*: Addison-Wesley, 1996.

[432] B. Anda, H. Dreiem, D. I. K. Sjøberg, and M. Jørgensen, "Estimating Software Development Effort Based on Use Cases - Experiences from Industry," in *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools* pp. 487-502, 2001.

[433] M. Laguna, J. Marqués, and Y. Crespo, "On the Semantics of the Extend Relationship in Use Case Models: Open-Closed Principle or Clairvoyance?," in *Advanced Information Systems Engineering*. vol. 6051, B. Pernici, Berlin / Heidelberg: Springer, pp. 409-423, 2010.

[434] S. Lilly, "Use Case Pitfalls: Top 10 problems from real projects using use cases," *Proceedings of Technology of Object-Oriented Languages and Systems*, 1999.

[435] A. Ciemniewska, J. Jurkiewicz, L. Olek, and J. Nawrocki, "Supporting Use-Case Reviews," in *Proceedings of the 10th international conference on Business information systems*, pp. 424-437, Berlin, Heidelberg, 2007.

[436] G. Kotonya and I. Sommerville, *Requirements Engineering: Processes and Techniques*: John Wiley and Sons Ltd, 1998.

[437] R. C. Martin, "Design Principles and Design Patterns," Objectmentor.com,Available: http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf, 2000.

[438] Altova. (2010). XMLSpy. Online: http://www.altova.com/xmlspy.html

[439] Altova. (2012). UModel. Online: http://www.altova.com/umodel.html

[440] Gentleware. (2010). Poseidon for UML. Online: www.gentleware.com

[441] Omondo. (2011). EclipseUML. Online: http://www.omondo.com/

[442] Objecteering. (2009). Objecteering. Online: http://www.objecteering.com/

[443] IBM. (2010). Rational Rose Modeler. Online: www.ibm.com/software/awdtools/developer/rose/modeler/

[444] S. Systems. (2011). Enterprise Architect. Online: http://www.sparxsystems.com.au/

[445] Collabnet. (2009). ArgoUML. Online: http://argouml.tigris.org/

[446] Serlio. (2011). CaseComplete. Online: http://www.casecomplete.com/

[447] TechnoSolutions. (2009). Visual Use Case. Online: http://www.visualusecase.com/

[448] V. Paradigm. (2011). Visual Paradigm for UML. Online: http://www.visual-paradigm.com/product/vpuml/

[449] K. Toutanova, D. Klein, C. Manning, and Y. Singer. (2003). Stanford POS Tagger. Online: http://nlp.stanford.edu/software/tagger.shtml

[450] T. Harris. (2010). yUML beta v0.18. Online: http://yuml.me/

[451] A. C. Jensen and B. H. C. Cheng, "On the use of genetic programming for automated refactoring and the introduction of design patterns," *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pp. 1341-1348, Portland, Oregon, USA, 2010.

[452] L. Briand, Y. Labiche, and L. O'Sullivan, "Impact Analysis and Change Management of UML Models," *Software Quality Engineering Laboratory,* Carleton University, Ontario, Canada, TR-SCE-03-01,Available: http://squall.sce.carleton.ca/pubs/tech_report/TR_SCE-03-01.pdf, 2001.

[453] L. Briand, Y. Labiche, L. O'Sullivan, and M. M. Sówka, "Automated impact analysis of UML models," *Journal of Systems and Software,* vol. 79, pp. 339-352, 2006.

[454] M. Chapman, M. Goodner, B. Lund, B. McKee, and R. Rekasius, "Supply Chain Management Sample Application Architecture," Web Services-Interoperability Organization,Available: http://www.ws-i.org/SampleApplications/SupplyChainManagement/2003-12/SCMArchitecture1.01.pdf, 2003.

[455] A. T. Pugibet, "CSTL: A Conceptual Schema Testing Language," Master Thesis, Conceptual Modeling of Information Systems Research Group, Universitat Politècnica de Catalunya (UPC), Spain, 2008.

[456] N. Koch, "Automotive Case Study: UML Specification of On Road Assistance Scenario," *Sensoria: Software Engineering for Service-Oriented Overlay Computers,* Information Society Technologies, Italy, FAST-TR_No.1,Available: http://rap.dsi.unifi.it/sensoria/files/FAST_report_1_2007_ACS_UML.pdf, 2007.

[457] D. Berndl and N. Koch, "Automotive Scenario: Illustrating Service Specification," *Sensoria: Software Engineering for Service-Oriented Overlay Computers,* Information Society Technologies, Italy, FAST-TR-No.2,Available: http://rap.dsi.unifi.it/sensoria/files/FAST_report_2_2007_ACS_Spec.pdf, 2007.

[458] R. Seidl and H. Sneed, Modeling Metrics for UML Diagrams," *Testing Experience*," 2010.

[459] D. E. Perry, A. A. Porter, and L. G. Votta, "Empirical studies of software engineering: a roadmap," *Proceedings of the Conference on The Future of Software Engineering*, pp. 345-355, Limerick, Ireland, 2000.

[460] A. Funes and C. George, "Formal Foundations in RSL for UML Class Diagrams," The United Nations University / International Institute for Software Technology, Technical Report 2532002.

[461] F. Mantz, "Syntactic Quality Assurance Techniques for Software Models," Diploma Thesis, Fachbereich Mathematik und Informatik, Philipps-Universität, Marburg, 2009.

[462] S. Meng and L. S. Barbosa, "A Coalgebraic Semantic Framework for Reasoning about UML Sequence Diagrams," in *The Eighth International Conference on Quality Software*, pp. 17-26, 2008.

[463] W3C. (2008, May 2012). *Extensible Markup Language (XML) Version 1.0*. Available: http://www.w3.org/TR/REC-xml/

[464] W3C. (2011, May 2012). *XQuery 1.0: An XML Query Language*. Available: http://www.w3.org/TR/xquery/

[465] W3C. (2007, May 2012). *XSL Transformations (XSLT) Version 2.0*. Available: http://www.w3.org/TR/xslt20/

[466] J. Wüst. (2011). SDMetrics. Online: http://www.sdmetrics.com

[467] J. Bansiya and C. G. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment," *IEEE Transactions on Software Engineering,* vol. 28, pp. 4-17, 2002.

Vitae

Name : Mohammed Misbhauddin

Nationality : Indian

Date of Birth :8/2/1981

Email : misbhauddin.mohammed@gmail.com

Address : Hyderabad, India

Academic Background : Mohammed Misbhauddin earned his Bachelors of Engineering degree in Computer Science and Engineering from Deccan College of Engineering and Technology (affiliated to Osmania University), Hyderabad, India in May 2003. He completed his Masters in Science from Illinois Institute of Technology, Chicago, USA in May 2005. His research interests include model-driven software development, software refactoring, meta-modeling, software metrics and quality, artificial intelligence and data mining.