

University of Alberta

**Improving the Quality of Use Case Models and their Utilization in
Software Development**

by

Mohamed El-Attar

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Software Engineering

Department of Electrical and Computer Engineering

Edmonton, Alberta

Fall 2009

Dedications

***to Mahmoud El-Attar
(Geddo)***

As ever, my parents Sherif El-Attar and Shahrzad Badawy are a constant source of support and encouragement. Their efforts do not go unnoticed, supporting me in every possible way and every step of the way, especially during the tougher latter stages of my degree, and are greatly appreciated. My parents have always been, and will always be, the rock of my life and without their constant prayers, never-ending love and support; I would not amount to anything. I am also very appreciative to the support given to me by other family members, especially my sister Iman El-Attar and brother Omar El-Attar, I miss you both and I am always thinking of you. I would also to make a very heartfelt dedication of this thesis to my grandparents, Mahmoud El-Attar (Geddo), Soraya El-Attar (Nona), Sameeha Badawy (Teita) and my late grandfather El-Sayed Badawy (Geddo El-Sayed). I love you all very dearly and miss you all very much.

Abstract

Use Case modeling has been constantly gaining popularity as the technique of choice for eliciting and documenting functional requirements. The deployment of Use Case models in industry has resulted in many positive experience reports being published. The inclusion of Use Case modeling into the Unified Modeling Language (UML) (OMG 2005) has aided its widespread use in industry, especially within the object-oriented community.

One of the most attractive aspects of Use Case modeling is its technical simplicity, allowing stakeholders with differing backgrounds, to have a common understanding of the requirements. This technical simplicity can be deceptive, as many modelers create models that are incorrect, inconsistent, and ambiguous and contain restrictive design decisions. In Use Case driven development processes, Use Case models are used to drive the design and test phases. While a number of techniques have been proposed to develop test cases from Use Case models, these techniques tend to suffer from two major shortcomings. The techniques are technically too complex to be effectively used by its potential users (business analysts and customers); and the inability to use these techniques in the early stages of development.

This thesis describes work tackling these deficiencies. Support for developing higher quality Use Case models is achieved by developing a modeling syntax that ensures consistency when constructing Use Case models. A controlled experiment was performed to empirically evaluate the effectiveness of using the modeling syntax. In addition, a technique based on utilizing antipatterns to detect

potentially defective areas in Use Case models was developed. The technique prompts modelers to revise and remedy poor design decisions, yielding superior quality models. Finally, a framework was developed, which utilizes Use Case models, to develop acceptance tests. The framework was designed to account for the technical abilities of its potential users.

Acknowledgements

I am indebted to my supervisor Dr. James Miller for his invaluable advice and criticism. Dr. James Miller has welcomed me to his research group with open arms since day one. Without him this thesis would genuinely not have been completed. I am very grateful for his support throughout my research. I am also very grateful for his essential advice towards my future academic and industrial careers. I regard Dr. James Miller not only as an excellent supervisor, but also as a best friend. I would like to thank Dr. Lionel Briand for accepting my enrollment into his Software Engineering undergraduate project back in the summer of 2002 and for inspiring to pursue a graduate degree in the field.

I would also like to thank my fellow graduate students at the department of Electrical and Computer Engineering, University of Alberta, who volunteered to take part in a controlled experiment performed during this research.

A number of other people that I would like to thank for taking time out of their busy schedules to answer my technical questions, namely Mr. Brian Berenbach, Mr. Scott Ambler, and Mr. Kurt Bittner.

The research contained within this thesis was supported by a number of Graduate Teaching Assistant position with the Department Electrical and Computer Engineering, University of Alberta. Travel to conferences was supported by the Department. Their support is gratefully acknowledged.

Publications

The research contained in this thesis is based on a number of publications. These are as follows:

Chapter 3:

- Producing Robust Use Case Diagrams via Reverse Engineering of Use Case Descriptions. M. El-Attar, J. Miller, Journal of Software and Systems Modeling, 2007.
- A User-Centered Approach to Modeling BPEL Business Processes Using SUCD Use Cases. M. El-Attar, J. Miller, Journal of Software Development and Theory, Practice and Experimentation - e-Informatica, 2007.
- A Subject-Based Empirical Evaluation of SSUCD's Performance in Reducing Inconsistencies in Use Case Models, M. El-Attar, J. Miller, Journal of Empirical Software Engineering, 2009.
- AGADUC: Towards a More Precise Presentation of Functional Requirements in Use Case Models, M. El-Attar, J. Miller, 4th ACIS International Conference on Software Engineering, Research, Management and Applications, 2006.

Chapter 4:

- Improving the Quality of Use Case Models Using Antipatterns, M. El-Attar, J. Miller, Journal of Software and Systems Modeling, 2009.
- Matching Antipatterns to Improve the Quality of Use Case Models, M. El-Attar, J. Miller, 14th IEEE International Requirements Engineering Conference, 2006.
- A Systematic Review and Formulation of Use Case Modeling Antipatterns, M. El-Attar, J. Miller, Journal of Information and Software Technology, (*under review*).

Chapter 5:

- Developing Comprehensive Acceptance Tests from Use Cases and Robustness Diagrams, M. El-Attar, J. Miller, Requirements Engineering Journal (*under review*).

Contents

CHAPTER 1 - NTRODUCTION.....	1
1.1. THE REQUIREMENTS ENGINEERING PROCESS	2
1.2. ISSUES WITH REQUIREMENTS ENGINEERING.....	5
1.3. THESIS CONTRIBUTION	7
1.4. THESIS OUTLINE.....	8
CHAPTER 2 - A REVIEW OF UC MODELING ISSUES AND POSSIBLE IMPROVEMENTS	11
2.1 UC MODELING	11
2.2 COMPONENTS OF A UC MODEL	12
2.3 SEMANTICS OF THE UC MODELING NOTATION.....	14
2.3.1 <i>Use Cases</i>	14
2.3.2 <i>Actors</i>	15
2.3.3 <i>Relationships</i>	15
2.3.4 <i>Association</i>	16
2.3.5 <i>Relationships between UCs Only</i>	17
2.3.6 <i>Actor Generalization</i>	19
2.4 IMPROVING UC MODELING.....	20
2.4.1 <i>Improving the Quality of UC Models</i>	20
2.4.2 <i>Improving the Requirements Engineering Process Using UC Models</i>	23
2.5 THESIS CONTRIBUTIONS TOWARDS IMPROVING UC MODELING.....	26
2.5.1 <i>Improving the Presentation of Functional Requirements in UC Models</i>	26
2.5.2 <i>Improving Consistency in UC Models</i>	27
2.5.3 <i>Consequencs of Inconsistencies in UC Models</i>	28
2.5.4 <i>Inconsistencies: A Closer Look</i>	30
2.5.5 <i>Reducing Inconsistencies in UC Models with SSUCD – An Empirical Evaluation</i>	33
2.5.6 <i>Using Antipatterns to Improve the Quality of UC Models</i>	35
2.5.7 <i>Towards Achieving the Benefits of Acceptance Testing for UC-Driven Development Processes</i> 35	
CHAPTER 3 - USING STRUCTURE TO IMPROVE THE QUALITY OF USE CASE MODELS.....	39
3.1 IMPROVING THE PRESENTATION OF FUNCTIONAL REQUIREMENTS IN UC MODELS	39
3.1.1 <i>Introduction</i>	39
3.1.2 <i>Related Work</i>	40
3.1.3 <i>The AGADUC process and the SUCD Structure</i>	41
3.1.4 <i>Mapping UC Descriptions to UCADs</i>	43
3.1.5 <i>The Library System Case Study</i>	49
3.1.5.1 <i>Displaying UCADs and Their Dependencies</i>	52
3.1.6 <i>AREUCD – Automating the ARADUC Process</i>	55
3.2 PRODUCING CONSISTENT USE CASE MODELS VIA REVERSE ENGINEERING OF USE CASE DESCRIPTIONS	57
3.2.1 <i>Introduction</i>	57
3.2.2 <i>Related Work in UC Authoring</i>	57
3.2.2.1 <i>Maintaining the Readability of UC Descriptions</i>	60
3.2.3 <i>Simple Structured UC Description (SSUCD)</i>	61
3.2.3.1 <i>A Brief Introduction to the Elements of SSUCD</i>	62
3.2.3.2 <i>Formalizing the SSUCD Structure Grammar</i>	69
3.2.4 <i>Consistency and Mapping Rules between UC Descriptions and Diagrams</i>	69
3.2.5 <i>Tool Support Using SARUECD</i>	74
3.2.6 <i>SSUCD Modeling Language Design</i>	77

3.2.7	<i>Online Hockey Team Store System Case Study</i>	84
3.3	A SUBJECT-BASED EMPIRICAL EVALUATION OF SSUCD'S PERFORMANCE IN REDUCING INCONSISTENCIES IN USE CASE MODELS	96
3.3.1	<i>Introduction</i>	96
3.3.2	<i>Experimental Planning</i>	96
3.3.2.1	Experiment Definition	96
3.3.2.2	Experiment Context.....	97
3.3.2.3	Hypotheses Formulation	98
3.3.2.4	Subject Selection	99
3.3.2.5	Experimental Design and Tasks	99
3.3.2.6	Time Allocation.....	101
3.3.2.7	Instrumentation	101
3.3.2.8	Analysis Procedure	102
3.3.2.9	Scoring and Measurement	102
3.3.3	<i>Analysis and Interpretation</i>	111
3.3.3.1	Performed Analysis	113
3.3.3.2	SSUCD vs. UNL – Inconsistencies.....	114
3.3.3.3	SSUCD vs. UNL – Completeness	116
3.3.3.4	SSUCD vs. UNL – Understandability	118
3.3.3.5	SSUCD vs. UNL - Fault-Free and Non-Analytical Information	119
3.3.3.6	Airline Ticketing System vs. Banking System	122
3.3.3.7	Group A vs. Group B	125
3.3.4	<i>Threats to Validity</i>	127
3.3.4.1	Conclusion Validity	127
3.3.4.2	Internal Validity	128
3.3.4.3	Construct Validity	128
3.3.4.4	External Validity	129
CHAPTER 4 - IMPROVING THE QUALITY OF USE CASE MODELS USING ANTIPATTERNS		132
4.1.	INTRODUCTION	132
4.2.	RELATED WORK	133
4.2.1.	<i>Computer-supported Verification of UC Models – State of the art</i>	133
4.2.2.	<i>Other Approaches</i>	133
4.3.	UC MODELING ANTIPATTERNS	135
4.3.1.	<i>Advantages of Using Antipatterns: What Can Antipatterns Do?</i>	136
4.3.2.	<i>Matching Antipatterns With UC Models</i>	137
4.3.3.	<i>Using OCL to Describe Unsound Diagrammatic Structures</i>	138
4.3.4.	<i>Domain Independent vs. Domain Dependent Antipatterns</i>	141
4.3.5.	<i>A Systematic Review Process for Antipattern Development</i>	144
4.3.5.1	Data Classification Scheme and Scope	146
4.3.5.2	Search Strategy	147
4.3.5.3	Filtering the Results.....	148
4.3.5.4	Filtering the Results.....	149
4.3.5.5	Results Analysis	150
4.3.6.	<i>Examples of UC Modeling Antipatterns</i>	151
4.4.	TOOL SUPPORT USING ARBIUM	222
4.5.	EVALUATION.....	224
4.5.1.	<i>Definition and Motivation</i>	224
4.5.2.	<i>Case Study Formulation</i>	225
4.5.3.	<i>Analysis and Interpretation of the Results</i>	230
4.5.4.	<i>Discussion of Results and Validation</i>	240
4.6.	COMPARISON OF ALTERNATIVE APPROACHES.....	243
CHAPTER 5 - DEVELOPING COMPREHENSIVE ACCEPTANCE TESTS FROM USE CASES		246
5.1.	INTRODUCTION	246

5.2.	DEVELOPING ACCEPTANCE TESTS FROM USE CASES	247
5.2.1.	<i>Phase 1: Developing High Level Acceptance Tests for Each Use Case</i>	256
5.2.2.	<i>Phase 2: Performing Robustness Analysis</i>	260
5.2.3.	<i>Phase 3: Developing Executable Acceptance Tests for Each Use Case</i>	268
5.3.	TOOL SUPPORT WITH UCAT.....	270
5.4.	EVALUATING THE EFFICIENCY OF THE DEVELOPED TESTS.....	272
5.5.	THE RESTOMAPPER SYSTEM CASE STUDY	272
5.5.1.	<i>UC: Generate Restaurant Map for City</i>	277
5.5.1.1	Examining the UC Descriptions and Creating its HLATs (Phase 1)	278
5.5.1.2	Robustness Analysis (Phase 2).....	279
5.5.1.3	Creating Executable Acceptance Tests (Phase 3)	280
5.5.2.	<i>Efficacy of the Developed Acceptance Tests</i>	282
5.6.	ROLE OF THE DEVELOPED ACCEPTANCE TESTS	283
CHAPTER 6 - CONCLUSIONS.....		285
6.1.	SUMMARY	285
6.2.	CONTRIBUTIONS AND RESULTS.....	286
6.2.1.	<i>Improving the Understandability of Functional Requirements with AGADUC</i>	286
6.2.2.	<i>Reducing Inconsistencies with SSUCD</i>	286
6.2.3.	<i>Using Antipatterns to improve the Quality UC models</i>	288
6.2.4.	<i>Producing Acceptance Tests from UC Models</i>	290
6.3.	FUTURE WORK	291
6.3.1.	<i>Future Work Based on Structured UCs</i>	291
6.3.2.	<i>Future Work Based Using Antipatterns</i>	292
6.3.3.	<i>Future Work Based Developing Acceptance Tests from UC Models</i>	293
BIBLIOGRAPHY		294
APPENDIX A - SUCD E-BNF.....		310
APPENDIX B - ACTOR AND UC DESCRIPTIONS OF THE LIBRARY SYSTEM CASE STUDY.....		314
APPENDIX C - SSUCD E-BNF		316
APPENDIX D - SCORING UCS DEVELOPED IN THE EXPERIMENT.....		318
D.1.	SCORING UCS FROM THE BANKING SYSTEM DEVELOPED IN UNL	318
D.2.	SCORING UCS FROM THE AIRLINE TICKETING SYSTEM DEVELOPED IN SSUCD.....	320
APPENDIX E - SYNTAX FOR CREATING COLUMNFIXTURES AND ROWFIXTURES		324
APPENDIX F - ROBUSTNESS ANALYSIS OF THE "GENERATE RESTAURANT MAP FOR CITY" UCS		327
APPENDIX G - ANALYZING UC "DISPLAY ROLLOVER INFORMATION".....		330
G.1.	EXAMINING THE UC DESCRIPTION AND CREATING ITS HLATs (PHASE 1).....	331
G.2.	ROBUSTNESS ANALYSIS (PHASE 2)	332

List of Tables

Table 1-1: Examples of errors and mistakes that can be committed during Requirements Engineering.	5
Table 2-1: UC description template	13
Table 2-2: Quality attributes of a UC model	22
Table 3-1: shows the UC description of the Borrow Book UC described in SUCD	50
Table 3-2: Quality principles that should be present in modeling languages	78
Table 3-3: A summary of SSUCD’s language constructs and their purposes	79
Table 3-4: Five dependent variables and their corresponding hypotheses	98
Table 3-5: Experimental design.....	100
Table 3-6: Details of the two systems used in this experiment	100
Table 3-7: defect examples.....	103
Table 3-8: Scoring Strategy.....	108
Table 3-9: Mann-Whitney test for the ‘Inconsistencies’ results.....	115
Table 3-10: Cliff’s delta for the ‘Inconsistencies’ results.....	115
Table 3-11: Mann-Whitney test for the ‘Completeness’ results.....	118
Table 3-12: Cliff’s delta for the ‘Completeness’ results.....	118
Table 3-13: Mann-Whitney test for the ‘Understandability’ results.....	119
Table 3-15: Mann-Whitney test for the ‘Fault-Free’ results.....	121
Table 3-16: Mann-Whitney test for the ‘Non-Analytical’ results	121
Table 3-17: Cliff’s delta for the ‘Fault-Free’ and ‘Non-Analytical’ results	122
Table 3-18: Descriptive statistics of the results.....	123
Table 3-19: Mann-Whitney test for all quality attributes	123
Table 3-20: Cliff’s delta for all quality attributes	123
Table 3-21: Descriptive statistics of the results.....	126
Table 3-22: Mann-Whitney test for all quality attributes	126
Table 3-23: Cliff’s delta for all quality attributes	127
Table 4-1: Antipattern template.....	136
Table 4-2: Types of antipatterns.....	142
Table 4-3: First Iteration Matches	231
Table 4-4: First iteration analysis	232
Table 4-5: Second iteration matches	239
Table 4-6: Second Iteration Analysis	239
Table 4-7: Addressing issues in the MAPSTEDI UC model.....	240
Table 4-8: Examining the MAPSTEDI UC model for violations of the heuristics presented in (Berenbach 2007)	243
Table 5-1: User acceptance testing vs. system testing.....	248
Table 5-2: Desirable characteristics of a technique aimed at developing acceptance tests	250
Table 5-3: HLAT format: The first column of a HLAT defines a unique test ID.	259
Table 5-4: Robustness diagram objects.....	261
Table 5-5: Properties of the RestoMapper UC model	274
Table 5-6: Objects in the domain model	276
Table 5-7: HLATs for the <i>Generate Restaurant Map for City</i> UC	279
Table 5-8: Results of performing robustness analysis on the Basic Flow	280
Table 5-9: Results of performing robustness analysis on the Alternative Flow	280
Table 5-10: Coverage provided by the created EATs	283
Table G-1: HLATs for the <i>Display Rollover Information</i> UC	331
Table G-2: Results of performing robustness analysis on the <i>Basic Flow</i>	335
Table G-3: Results of performing robustness analysis on the <i>Alternative Flow: Clicked on coordinates with multiple restaurants</i>	336
Table G-4: Results of performing robustness analysis on the <i>Alternative Flow: Clicked on coordinates with no restaurants</i>	337

List of Figures

Figure 1-1: The Requirements Engineering Process and its Sub-disciplines	4
Figure 2-1: Explanatory UC diagram	12
Figure 2-2: The application of the REUCD process to systematically generate UC diagrams from descriptions	32
Figure 2-3: V-Model development process	36
Figure 3-1: The AGADUC process	43
Figure 3-2: Conversion of a <i>Header</i> to activities in swimlanes	45
Figure 3-3: Transforming RESUME and AFTER statements	46
Figure 3-4: The Library system UC diagram	49
Figure 3-5: The UCADs of the entire UC model	54
Figure 3-6: AREUCD generating the UCADs for the UC model	56
Figure 3-7: UC name and its representation	63
Figure 3-8: Abstraction and implementation in UCs and their representation	64
Figure 3-9: Generalization between UCs its representation	65
Figure 3-10: Associations between UCs and actors and its representation	65
Figure 3-11: The <i>include</i> relationship represented in the UC description body	66
Figure 3-12: The <i>extend</i> relationship represented in the UC description body.	68
Figure 3-13: Systematically generating UC diagrams from descriptions and description skeletons from diagrams	71
Figure 3-14: A screenshot of SAREUCD after transforming the descriptions to an object model	75
Figure 3-15: A UC description	77
Figure 3-16: The UC diagram after three UC descriptions are read	90
Figure 3-17: The UC diagram after five UC descriptions are read	93
Figure 3-18: The UC diagram after all UCs and actors are read	94
Figure 3-19: Illustration of the box and whiskers plot's diagrammatic notation	111
Figure 3-20: Inconsistencies - Airline Ticketing System	114
Figure 3-21: Inconsistencies - Banking System	115
Figure 3-22: Completeness - Airline Ticketing System	117
Figure 3-23: Completeness - Banking System	117
Figure 3-26: Fault-Free - Airline Ticketing System	119
Figure 3-27: Fault-Free - Banking System	120
Figure 3-28: Non-Analytical - Airline Ticketing System	120
Figure 3-29: Non-Analytical - Banking System	121
Figure 4-1: The simplified version of the UC metamodel used in ARBIUM	141
Figure 4-2: A good scenario of an actor being directly associated with a <i>generalized</i> UC	152
Figure 4-3: The execution flow of a <i>generalization</i> relationship	153
Figure 4-4: A bad scenario of an actor being directly associated with a <i>concrete generalized</i> UC	154
Figure 4-5: The <i>generalized</i> UC is set to be <i>abstract</i> to make sure that one of its <i>specializing</i> UCs services the actor's request	155
Figure 4-6: Direct access to the <i>generalized</i> UC is avoided and replaced with direct access to its <i>specializing</i> UCs	156
Figure 4-7: UC Car Not Found was incorrectly used for the purposes of containing common functionality and exception-handling behavior.	157
Figure 4-8: The shared UC is broken into two separate UCs, each serving a different purpose. .	159
Figure 4-9: Functional decomposition of the Prepare Coffee UC	160
Figure 4-10: Improper use of the <i>extend</i> relationship to promote functional decomposition	163
Figure 4-11: <i>Extending</i> UCs disjointed to properly handle different exceptional situations.	166
Figure 4-12: Sequencing a set of UCs to make a phone call	167
Figure 4-13: Creating a virtual call sequence between UCs using pre and postconditions	168
Figure 4-14: A good scenario of directly accessing an <i>extending</i> UC to allow independent initiation of an optional service	170

Figure 4-15: The <i>extension</i> UC is used to notify the actor of an exceptional situation.	171
Figure 4-16: The actor is allowed to directly access the <i>extension</i> UC since the actor is the source of information required by the <i>extension</i> UC.	171
Figure 4-17: The <i>extending</i> UC communicates with the actor decide how to deal with an exceptional situation.	172
Figure 4-18: Setting the navigation allows the UC to initiate communication with the actor.	175
Figure 4-19: The <i>base</i> UC should be the one conveying the <i>extension</i> UC its required information instead of the actor.	175
Figure 4-20: Setting the navigation direction to ensure that the actor does not start the interaction with the extension UC.	176
Figure 4-21: Two actors that play a similar role when executing a UC	176
Figure 4-22: A model representing instances of an actor	177
Figure 4-23: Two actors appropriately associated with a UC	177
Figure 4-24: The overlapping roles between the two actors should be <i>generalized</i> into a separate actor.	179
Figure 4-25: The model should represent the role of a class of users not instances of them.	179
Figure 4-26: an <i>abstract</i> UC <i>including</i> subroutine behavior and being <i>extended</i> by a UC containing exceptional or optional behavior	182
Figure 4-27: An <i>abstract</i> UC <i>including</i> UCs that contain specialized behavior	183
Figure 4-28: The <i>abstract</i> UC now set to be <i>concrete</i>	184
Figure 4-29: The <i>abstract</i> UC is associated with its <i>specializing</i> UCs using the <i>generalization</i> relationship	185
Figure 4-30: Multiple <i>generalizations</i> of one UC	186
Figure 4-31: The <i>generalized</i> UC should be <i>included</i> by the other UCs that need it	187
Figure 4-32: Duplicating functionalities for the <i>generalized</i> and <i>specialized</i> UCs	187
Figure 4-33: Only the appropriate <i>include/extend</i> relationships remain.	189
Figure 4-34: An actor directly association with an unimplemented <i>abstract</i> UC	190
Figure 4-35: Two actors initiating one UC.	192
Figure 4-36: Only one actor should initiate a UC	194
Figure 4-37: Two different actors with the same name	194
Figure 4-38: The actors are given more distinguishing names.	196
Figure 4-39: Two subsystems are presented as two UC diagrams, both containing the name actor	197
Figure 4-40: A Keyboard actor is used to enter billing information.	205
Figure 4-41: A Printer actor is required to print statements.	205
Figure 4-42: The Keyboard actor is replaced with the actual actor Customer.	207
Figure 4-43: Extending the metamodel to support actor to actor associations	213
Figure 4-44: Two UCs trading information to provide a service	214
Figure 4-45: A main UC is communicating with other UCs by “calling” them to retrieve necessary information.	215
Figure 4-46: A UC communicating with the other to provide exceptional behavior.	215
Figure 4-47: Extending the metamodel to support UC to UC associations <i>context</i>	217
UseCase.	217
Figure 4-48: Communicating UCs merged into one.	218
Figure 4-49: Communicating UCs are <i>included</i> by a UC that provides a separate complete service	218
Figure 4-50: The UC containing exceptional behavior now <i>extends</i> the <i>base</i> UC	219
Figure 4-51: An overview of how ARBIUM and USE can automate the detection process.	224
Figure 4-52: The UC diagram of the “Database Access” subsystem	227
Figure 4-53: The UC diagram of the “Database Queries” subsystem	227
Figure 4-54: The UC diagram of the “Database Integrator” subsystem.	227
Figure 4-55: The UC diagram of the “Database Edits” subsystem	228
Figure 4-56: The UC diagram of the “Administrative Process” subsystem	228
Figure 4-57: The Database Access UC diagram after the first iteration.	237
Figure 4-58: A merged view of the remaining three UC diagrams after the first iteration.	238
Figure 4-59: The Administrative Process UC diagram after the first iteration.	238

Figure 5-1: The overall process of developing high and executable acceptance tests per UC	255
Figure 5-2: Developing HLATs by analyzing the UC text and domain model	260
Figure 5-3: The banking system UC model.....	263
Figure 5-4: The “Withdraw Cash” UC description (left) and domain model (right).....	263
Figure 5-5: The robustness diagram corresponding to the “Withdraw Cash” UC.....	265
Figure 5-6: The <i>updated</i> “Withdraw Cash” UC description (top) and domain model (bottom) ..	266
Figure 5-7: ActionFixture example of performing transactions..	269
Figure 5-8: UCAT – associating FIT tests with the “Withdraw Cash” UC.....	271
Figure 5-9: UCAT – displaying test results of the “Calculate Investments” UC	271
Figure 5-10: The UC diagram of the RestoMapper application	273
Figure 5-11: The RestoMapper UC model into UCAT	274
Figure 5-12: The initial domain model of the RestoMapper system	275
Figure 5-13: Textual description of the <i>Generate Restaurant Map for City</i> UC.....	278
Figure 5-14: EAT of the <i>Basic Flow</i>	281
Figure 5-15: EAT of the Alternative Flow: Invalid Zoom Setting	281
Figure 5-16: Subsection of the V-Model development process emphasizing the role of acceptance tests.....	284
Figure D-1: Example Banking system UC diagram developed by a subject.....	318
Figure D-2: Example Airline Ticketing system UC diagram developed by a subject.....	321
Figure E-1: <i>ColumnFixture</i> example of calculating the return on one year investments..	324
Figure E-2: RowFixture example of checking account activities.....	325
Figure E-3: ActionFixture example of performing transactions.....	325
Figure G-1: Textual description of the <i>Display Rollover Information</i> UC.....	331
Figure G-2: EAT of the <i>Basic Flow</i>	338
Figure G-3: EAT of the Alternative Flow: Clicked on coordinates with multiple restaurants	339
Figure G-4: EAT of the Alternative Flow: Clicked on coordinates with no restaurants	339

Chapter 1

Introduction

Software Requirements Engineering is an important aspect of any software project. Requirements Engineering is the process of determining what is to be produced in a software system. In developing a complex software system, the requirements engineering process has the widely recognized goal of determining the needs for, and the intended external behavior, of a system. In their pioneering book on Software Requirements Engineering (Sommerville and Sawyer 1996), Sommerville and Sawyer define Requirements Engineering as follows:

“Requirements are...a specification of what should be implemented. They are descriptions of how the system should behave, or of a system property or attribute. They may be a constraint on the development process of the system.”

It can be deduced from this broad definition that there are different types of requirements information that to need to be gathered, which can be generally split into three categories (Wieggers 2003):

- *Business requirements* describe why the product is being built and identify the benefits that both the customers and the business will reap.

- *User requirements*, often captured in the form of use cases, describe the tasks or business processes a user will be able to perform with the product.
- *Functional requirements* describe the specific system behavior that must be implemented. The functional requirements are the traditional "shall" statements found in a software requirements specification (SRS) document.

1.1. The Requirements Engineering Process

The term Requirements Engineering is a general term used to encompass all activities related to the requirements process. The requirements engineering process can be fine-grained into two main categories: (a) Requirements Development; and (b) Requirements Management (Wiegiers 2003). Requirements Development is further subdivided into four main activities (see Figure 1-1). Even though these activities usually do not occur exclusively, they generally occur in sequence. The four activities of Requirements Development are:

- **Elicitation**

This activity is concerned with the acquisition of information and data that will ultimately be used to determine the needs or conditions to meet for a new or altered software product. This activity ideally begins with identifying stakeholders and determining their expectations using a number of techniques such as conducting interviews, workshops and observing them perform duties in their workplace. Stakeholders include beneficiaries, customers and end-users. There are also other sources for requirements that need to be considered, such as

previous software products that are in place or have been used in the past and general domain knowledge.

- **Analysis**

This activity involves an analysis of the information and data gathered during the elicitation activity. Analysis is performed to:

- Determine and discard irrelevant and redundant data;
- Identify conflicting requirements of the various stakeholders; and
- Determine other requirements “missed” during the requirements elicitation activity.

The ultimate goal of performing requirements analysis is determining exactly what the stakeholders actually need and what can be expected from the software product.

- **Specification**

Requirements specification is the activity of presenting and documenting the requirements. This activity is required for two critical reasons: (a) to allow business stakeholders such as customers and end-users to verify and “sign-off” on the software product to be built; and (b) for the development team to determine a design solution that will realize these requirements. Requirements can be presented in multiple forms and using a number of techniques. For example:

- A Software Requirements Specifications (SRS) document.
- Use Case Models.

- User Stories.
- Formal methods.
- Prototypes.

Each technique has advantages and disadvantages.

▪ **Verification**

Upon specifying the requirements, they need to be verified. This is a critical activity that ensures that the “right” system will be built. At the business end, stakeholders such as the customer and end-users ensure that their needs will be satisfied by verifying and agreeing upon the specified capabilities, behavior and limitations of the software product. At the development end, developers verify the specified requirements to ensure that they have an accurate understanding of the software system’s capabilities, behavior and limitations.

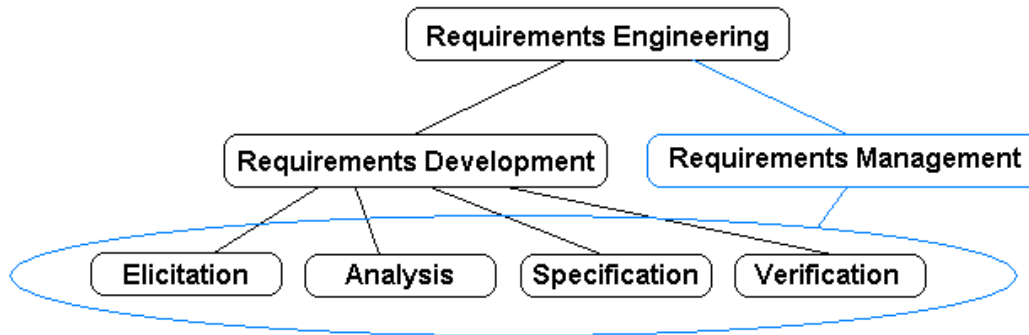


Figure 1-1: The Requirements Engineering Process and its Sub-disciplines

Requirements Management, much like Software Management, involves the tracing, estimation and overall management of the activities included in Requirements Development, and it encompasses the entire life cycle of a project (Hood et al. 2007; Leffingwell et al. 2000). Some software development methodologies require that the entire (or at least the majority of the) Requirements

Engineering phase be conducted up front. However, it is uncommon that the information and data necessary to produce a complete and stable set of requirements is available up front. Hence, there are other development methodologies that divide the Requirements Engineering effort throughout the development life cycle. Regardless of the development methodology used, Requirements Management is required for administrative, legal and financial purposes.

1.2. Issues with Requirements Engineering

Requirements engineering can impact the success of a given project in multiple ways. Improper practice or lack of Requirements Engineering can lead to the development of the “wrong” system or a system that is later judged unsatisfactory or unacceptable, has high maintenance costs, or undergoes frequent changes (Wiegiers 2003). Poor Requirements Engineering practices may occur during any of the Requirements Engineering activities. Mistakes and improper practice may occur during any activity of the Requirements Engineering process (see Figure 1-1). This is further clarified with examples as shown in Table 1-1.

Table 1-1: Examples of errors and mistakes that can be committed during Requirements Engineering.

Activity	Examples of Errors and Mistakes due to Improper Practice
Elicitation	<ul style="list-style-type: none"> ▪ Information and data gathered can be incorrect. ▪ Important requirements are missed. ▪ Inadequate user involvement. ▪ Inadequate preparation to interview users or for conducting

	<p>workshops.</p> <ul style="list-style-type: none"> ▪ Insufficient domain knowledge acquisition.
Analysis	<ul style="list-style-type: none"> ▪ Inability to detect conflicting requirements. ▪ Inability to discard redundant and irrelative information and data. ▪ Improper classification of stakeholders. ▪ Improper definition of project scope. ▪ Incorrect assumptions.
Specification	<ul style="list-style-type: none"> ▪ Specified requirements are incorrect. ▪ Specified requirements are ambiguous. ▪ Requirements are specified in a form that is too difficult to understand. ▪ Requirements specifications are incomplete.
Validation	<ul style="list-style-type: none"> ▪ Inadequate user and customer involvement to properly verify all specified requirements. ▪ Inadequate involvement from the development team to verify all specified requirements. ▪ Test cases produced are insufficient to encompass all (or at least the most common and critical) usage scenarios.
Management	<ul style="list-style-type: none"> ▪ Incorrect tracking of requirements specifications leading to a set of requirements that are out-of-date, incorrect and incomplete. ▪ Incorrect tracking of the Requirements Engineering budget and

	<p>inaccurate effort estimation techniques, leading to cost overruns and schedule delays.</p> <ul style="list-style-type: none">▪ Improper handling of legal issues leading to costly law suits.
--	--

Mistakes and errors such as those presented in Table 1-1 lead to the injection of defects at the Requirements phase. It is a well known fact that the cost of detecting and fixing bugs or defects introduced at the requirements phase escalates significantly as they propagate to later development phases such as coding and testing (Boehm 2005; Fagan 1976; Gilb et al. 1993; Wohlin et al. 1990). Therefore, it is most beneficial to improve Requirements Engineering to eliminate or at least minimize any defects.

1.3. Thesis Contribution

Requirements Engineering is a thriving research area. Researchers and practitioners in academia and industry search for ideas and solutions to improve every aspect of Requirements Engineering. A detailed roadmap of Requirements Engineering research can be found in (Betty et al. 2007). This thesis presents research that aims to improve Requirements Specification and Verification, in particular to improve the quality and usage of Use Case (UC) Models. UC Modeling, which will be discussed in detail in Chapter 2, is a popular technique for specifying functional requirements. Given its popularity, especially since UC models are part of the very widely used UML (OMG 2005), UC models provide an excellent venue to improve Requirements specification and validation efforts,

and hence improving the Requirements Engineering process in general. This can be achieved by improving various aspects of UC models. In particular, this thesis makes the following contributions:

- An approach that bridges the gap between the analysis phase and the design phase by systematically transforming UC models into activity-like diagrams as a means to unambiguously communicate the intended functional requirements.
- A structure that will improve quality in UC models by ensuring their consistency.
- A technique based upon detecting antipatterns (discouraged modeling practices) that will prompt modelers to reconsider their design and perhaps change it to improve the quality of the developed UC model.
- A technique that utilizes UC models to produce a comprehensive set of a User Acceptance Tests to improve the validation activity of functional requirements by its users.

1.4. Thesis Outline

The remainder of this thesis takes the following form:

Chapter 2: The body of the thesis begins with an introduction to UC modeling, its basic concepts and common terminology. This Chapter also presents a review of UC modeling issues as described in the literature, which suggested the main areas of research which should be undertaken to improve UC modeling.

Chapter 3: This Chapter introduces the concept of utilizing a structure to author UC descriptions to improve the overall UC modeling efforts. This Chapter has three major Sections:

- Section 3.1 presents the AGADUC process and the SUCD structure, which were developed to systematically generate activity-like diagrams that provide a more precise presentation of a system's functional requirements.
- Section 3.2 presents the SSUCD structure, which is a simplified version of the previously developed SUCD. SSUCD is used to aid UC authors in developing consistent UC models and thus improving the quality of the models they create.
- Section 3.3 presents an empirical evaluation of SSUCD vs. unstructured natural language (UNL) to gauge the accessibility of SSUCD by its potential users and to compare its effectiveness in reducing inconsistencies in comparison to using UNL.

Chapter 4: This chapter presents work that resulted in developing a new technique that utilizes antipatterns as a mechanism for improving the quality of existing UC models. The technique is based on recognizing questionable modeling decisions and prompting the modeler to reconsider these decisions. The result of this technique is that the modeler would make better educated modeling decisions, which may lead the modeler to change the model accordingly or leave it unchanged.

Chapter 5: This chapter presents a new technique that will improve Requirements Validation by developing a more comprehensive set of User Acceptance Tests. The technique primarily utilizes UC models as a basis for developing acceptance tests.

Chapter 6: The final chapter summarizes the content and contributions of this thesis, considers further work related to this research, and presents some conclusions.

Chapter 2

A Review of UC Modeling Issues and Possible Improvements

As a prelude to describing the issues associated with and possible improvements for UC modeling, this Chapter begins with introducing UC modeling, its main components as well as its notational syntax and semantics. This Chapter then provides an in-depth review of the UC modeling literature to expose the limitations, drawbacks and possible improvements that motivated the research work presented in subsequent Chapters.

2.1 UC Modeling

UC modeling (OMG 2005), since it was introduced in the early 1990s by Ivar Jacobson has been constantly gaining wide acceptance by analysts, designers, testers and other stakeholders of a project. UC modeling can be used to drive the design phase, the testing phase (Jacobson 1992, Kaner et al. 2003) and can be utilized for managerial purposes such as effort estimation (Anda et al. 2001b) and business modeling (Jacobson et al. 1995). The success experienced by UC modeling is chiefly because it is very simple to use to effectively describe the functional requirements of a system. Another attractive aspect of UC modeling is

that it contains a small diagrammatic notational subset and a large degree of natural language. This allows all of the stakeholders within a project to understand the UC model – even those who are not technically equipped. This in turn will ensure that all stakeholders have a common understanding and agreement upon the capabilities and features of the system.

2.2 Components of a UC Model

A UC model contains three main components; (a) a diagram, (b) textual descriptions of the UCs, and (c) a glossary. Each component serves a different purpose and they are explained in further detail below:

I. UC Diagram: The diagram serves as a visual summary of the involvement between the system’s users and the services it offers. A UC diagram is composed of actors, UCs, the system boundary and several types of relationships between actors and UCs (see Figure 2-1). The modeling semantics behind each diagrammatic element is further explained in Section 2.3.

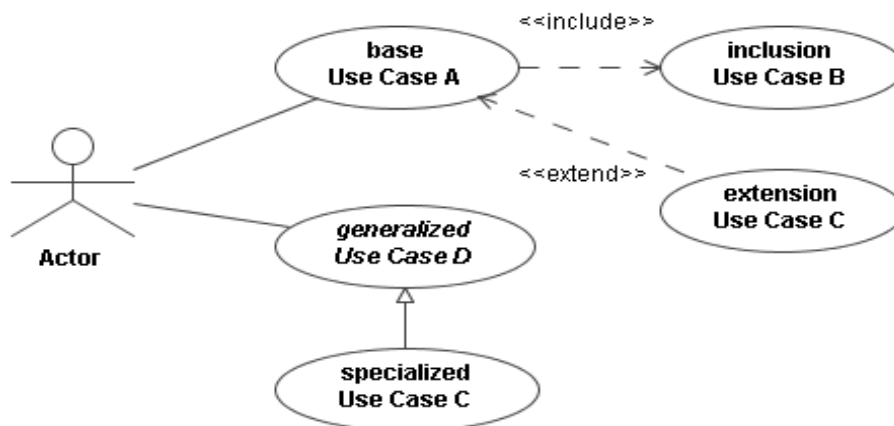


Figure 2-1: Explanatory UC diagram

II. UC Descriptions: Each UC must have a written description that clearly describes its behavior and purpose. Therefore, there is a “UC Description” component for each UC. A template is usually used to compose UC descriptions. There exist many templates to describe UCs. Various templates are used to describe UCs at various levels of abstraction, or to target specific domains. The template considered throughout this thesis is shown in Table 2-1. This template is considered since it contains sections that are commonly present in most UC templates.

Table 2-1: UC description template

Brief Description: A brief description of the UC’s main behavior.

Preconditions: All the preconditions that must be satisfied before the execution of the UC can commence.

Basic Flow: A list of events that would “normally” occur when the UC is performed.

Alternative Flows: A list of events that cover behavior that is optional, exceptional or just significantly different from the basic flow.

Sub-flows: A list of events representing a segment of behavior within the UC. Those events need to have a clear purpose and are “atomic”.

Postconditions: All the postconditions that must be satisfied before the execution of the UC can be successfully completed.

Extension Points: Particular locations in the flow of events where additional behavior can be inserted or attached.

Special Requirements: Any additional special requirements required by the

UC for its proper execution.

III. Glossary: This component is not exclusive to UC models. The glossary is used to describe terms found in the entire set of documents of a project. For the purposes of UC models, the glossary should contain a brief description of the actors, the UCs and any other terms described in the UC model.

2.3 Semantics of the UC Modeling Notation

Constructing UC models requires knowledge of its notation and the semantics behind each notational construct. As mentioned previously, a UC diagram contains UCs, actors and a limited set of relationships that combine these elements together. In this Section, a detailed explanation of each notational construct and its semantics is presented.

2.3.1 Use Cases

UCs are the building blocks of a UC diagram. UCs represent the set of services that the underlying system provides to its beneficiaries. A service represented by a UC must be complete and meaningful. Graphically UCs are depicted as ovals with its intended service written in the center of the oval. There exist two types of UCs: (a) *concrete* UCs, and (b) *abstract* UCs. In conformance with Object-Oriented concepts, a *concrete* UC can be initiated (“performed”) to provide a complete and meaning service to an actor. This is the nominal type of UC. An *abstract* UC cannot be solely initiated. An *abstract* UC needs to be *implemented*

by other *concrete* UCs in order to provide a meaningful service to an actor. An *abstract* UC provides a high-level description of a functionality that needs to be performed. UCs that implement an *abstract* UC provide details of how this functionality is performed. For a UC to implement the functionality of an *abstract* UC, it is linked to it through an *implementation* relationship. An *abstract* UC is graphically distinguished from *concrete* UCs by having its label in an *italic* font and by having the <<abstract>> stereotype depicted in the UC oval. The *implementation* relationship as well as other types of relationships will be discussed in greater detail in Section 2.3.5.

2.3.2 Actors

An actor is any external entity that interacts with the system at hand. An actor does not have to be a human. An actor can be other systems too. Beneficiaries of a system's services are usually modeled as actors however not every actor is a beneficiary. Primary actors are the actors that are involved with the system to attain a service from the system. Secondary actors are actors that are involved with the system in order to facilitate the delivery of a service to the primary actor. Graphically actors can be depicted using any icon. Traditional human actors are depicted as stick-man figures.

2.3.3 Relationships

There are a limited number of relationships that can exist between elements of a UC model. Relationships define the nature of the association between elements of

a UC model. There are relationships that are only valid between UCs; a single type of relationship that is only valid between actors and UCs; and a single type of relationship that is only valid between actors.

2.3.4 Association

The association relationship is the only type of relationship that is allowed between a UC and an actor. The association relationship simply means that the actor is “involved” with the given UC and can benefit from its services. The association relationship also means that an actor can initiate a UC, or the UC can initiate an interaction with the actor. Graphically, an association relationship is represented using a solid line between the actor and the UC. Directed association relationships are similar to regular association relationship with the only difference being that it specifies that only one of the involved parties is allowed to initiate an interaction. This is determined graphically by amending an arrowhead to one end of the association relationship. If the arrow is pointing towards the actor, this means that only the UC is allowed to initiate an interaction between itself and the actor. Meanwhile, if the arrow is pointing towards the UC, this means that only the actor is allowed to initiate an interaction between itself and the UC.

2.3.5 Relationships between UCs Only

There exist four types of relationships between UCs

- **Inclusion**

An inclusion relationship is used to “factor-out” a number of “steps” that are common between a number of UCs. The inclusion relationship visually shows the commonality in the behavior of two or more UCs. An inclusion UC contains routine-type behavior. An inclusion UC is a UC that is included by another (base) UCs. When a base UC is performed, it “invokes” the behavior contained in the inclusion UC every time. There is no condition set to “invoke” the inclusion UC. It is important to note that since the inclusion UC contains behavior that would normally be in the base UC, the base UC is now “incomplete” by itself, and that is why the inclusion UC needs to be “invoked” every time. The notation of an inclusion relationship is shown in Figure 2-1. Note that the inclusion UC is the one being pointed at by the arrow.

- **Extension and Extension Points**

An extension relationship is used to allow one UC:

- to handle an exceptional (or erroneous) situation that occurs within the *base* UC; or
- it is used to insert additional optional functionality.

An exceptional situation can be thought of as a very complicated alternative flow.

Whereby, if we would describe an alternative flow within the base UC, then the

description of the alternative flow would be so long that it will obscure the principle purpose of the base UC. Exceptional situations often lead to situations where the intended service is not delivered at all. In fact, there would be “recovery” or “clean-up” procedures required due to that exceptional situation arising.

Additional optional behavior on the other hand is different but much simpler to identify. Additional optional behavior simply performs additional procedures or functionality in addition to the functionality already described in the base UC. This behavior is optional in nature with respect to the functionality already described in the base UC. It is also useful to model behavior that is intended for future releases, or a service that in itself is quite different from the service provided by the base UC.

Regardless of the intent behind an extension relationship, a condition must be set and satisfied in order for the extension UC to execute. The condition is a property of the extension relationship itself. The notation of an extension relationship is shown in Figure 2-1. Note that the extension UC is the one that originates the arrow while the base UC is the one that is being pointed to by the arrow.

Extension Points represent specific locations within the base UC where the behavior from the extension UC will be inserted. Graphically an extension point is depicted by drawing a horizontal line under the name of a UC and listing the name of the extension point underneath.

- **Generalization**

This relationship is analogous to the generalization relationship of Object Oriented design. A child is said to specialize the parent UC, while the parent UC is said to generalize the child UC. The graphical notation is indeed the same as in UML class diagrams. The parent UC will contain general behavior that is applicable to all children UCs that specialize it.

- **Implementation**

The implementation relationship exists between an abstract UC and a concrete UC that is said to implement it. The graphical notation is exactly the same as the generalization relationship which can be a source of confusion. Abstract UCs as mentioned earlier are “incomplete” UCs; hence, they cannot be “initiated”. That is, they cannot be executed on their own (obviously because they are “incomplete”). An abstract UC will usually describe a service that can be performed in a number of ways. The means of implementation is described by the child UC. The notation of the implementation relationship is similar to that of the generalization relationship, where the arrowhead points towards the parent UC.

2.3.6 Actor Generalization

The actor generalization relationship has the same semantics as the UC generalization relationship. The actor generalization relationship indicates that a parent actor contains general characteristics with respect to the child actors that specialize it. Actors are external entities to the system and hence intricate details

of how they work should not be described but rather how they relate to the system. The actor generalization relationship is merely used to provide further information about how the actors involved with the system relate to each other, but not for the purposes of providing an in-depth explanation of their inner workings. The actor generalization relationship is depicted similarly to the UC generalization relationship.

The previous sections provided a brief introduction to UC modeling. A more in-depth explanation of UC modeling concepts and its notation can be located in (Bittner et al. 2002; Overgaard et al. 2003).

2.4 Improving UC Modeling

The general motivation behind the research presented in thesis is to improve two aspects of UC modeling:

1. Improve the quality of UC models produced; and
2. Improve the Requirements Engineering Process using UC models.

2.4.1 Improving the Quality of UC Models

As a prerequisite to improving quality in UC models, it is necessary to identify the quality attributes that a UC model should embody. The literature has identified many quality attributes of UC models and have also proposed many approaches to improve the quality of UC models (Anda et al. 2002, 2001a; Belgamo et al. 2005; Ben Achour et al. 1999; Bittner et al. 2002; Booch et al.

2005; Cockburn 1995, 2000; Constantine et al. 1999; Firesmith 1999; Harwood 1997; Jaaksi 1998; Jacobson et al. 1992; Kaner et al. 2003; Kulak et al. 2000, Larman 2001; Mattingly et al. 1988; McCoy 2003; Regnell et al. 1995; Rosenberg et al. 1999, 2007; Schneider et al. 1998; STEAM 2009). For example, (Ben Achour et al. 1999; Cockburn 1995, 2000; Kaner et al. 2003; STEAM 2009) present styling guidelines and templates to improve the authoring process of UCs. Other templates were also presented in (Cockburn 1995, 2000; Höst et al. 2000; Jacobson et al. 1992; Larman 2001; McBeen 2007). In (Kaner et al. 2003), guidelines were proposed to help identify actors and UCs. Other guidelines and techniques aim to help modelers avoid UC modeling pitfalls (Anda et al. 2002; McCoy 2003). Even though the heart of a UC model is in its UC descriptions (Anda et al. 2001a), the UC diagrams serve a different yet important purpose and are commonly used in industry, for example (MAPSTEDI 2008; FAIN 2008; SCM 2008; CancerGrid 2008). In turn, other guidelines were proposed to enhance the quality of UC diagrams (Bittner et al. 2002; Rosenberg et al. 1999). Based on the literature, the quality attributes of UC models can be divided into five major categories as shown in Table 2-2. It can be deduced that an improvement in any quality attribute, while not changing any other, improves the overall understandability of the UC model.

Table 2-2: Quality attributes of a UC model

Quality Attribute	Definition
Consistency	The UC diagram must conform to the concepts contained in the UC descriptions and vice versa. Consistent facts and information must be present across UC descriptions. If a UC model contains more than one UC diagram, consistency must exist between UC diagrams with respect to elements that they depict.
Completeness	The underlying requirements must correctly be represented by the UC diagram and textual descriptions. This means that all information and facts that are expected to be in the UC descriptions and diagram must be present.
Fault-Free	The UC diagram and descriptions must not contain any information or facts that are incorrect, which misrepresent the underlying requirements.
Analytical	The model should be analytical, meaning that it should only describe what the system should do. This includes the exclusion of any design or implementation decisions, including interface details. Except those explicitly defined by the customer.
Understandability	The model must be presented in a readable form. The information contained in the UC descriptions must be

	precise and unambiguous. The model should also not contain repeated information as this may lead to confusion. All stakeholders must share a common understanding of the presented functional requirements.
--	---

A UC model lacking any of the quality attributes, presented in Table 2-2, is likely to lead to harmful consequences. Anda *et al.* (Anda et al. 2001a) outline a comprehensive list of potential harmful consequences that can result from a deficiency in any quality attribute.

2.4.2 Improving the Requirements Engineering Process Using UC Models

The Requirements Engineering phase is very influential towards the overall success of a Software Development project since it affects the success of each subsequent development phase.

As mentioned in Section 1, the Requirements Engineering process consists of two sub-disciplines; Requirements Development and Management. In order to improve the Requirements Engineering Process, it is necessary identify the role of UC modeling in performing these activities.

➤ Requirements Development

▪ Elicitation

In UC modeling, a UC is used to document one or more usage scenarios that pertain to achieving a goal. From an elicitation stand point, UC modeling prompts

the requirements analyst to identify functional requirements in terms of the services the system will offer its beneficiaries. The analyst will then be prompted to identify the possible successful and unsuccessful scenarios that may occur while attempting to attain these services. The analyst will also be prompted to identify the potential external entities (actors) that will be interacting with the system while it performs its duties.

- **Analysis**

Upon identifying the services that a system will provide and its usage scenarios, an analyst will be able to determine key aspects of the system, such as:

- The most common usage scenarios and the exceptional ones. This will allow the analyst to determine the Basic Flow and Alternative Flows.
- Common and repetitive workflows. This will allow the analyst to determine Sub-flows and inclusion UCs.
- Additional required usage scenarios such as exception handling scenarios.
- The most common users of the system.

These analyses are made possible through various features of a UC model such as the availability of various types of relationships and the basic components of a UC description template. To further elaborate, the existence of the inclusion relationship will prompt an analyst to consider common and repetitive workflows.

- **Specification**

The simplicity of the requirements specification using UC models is one of its most attractive features. The UC diagrams notational constructs and syntactical rules are limited and relatively easy to learn. UC descriptions are authored using UNL, which allows the author the highest degree of flexibility and freedom in documenting the requirements.

- **Validation**

The relative simplicity of the UC modeling, which makes UC models easy to construct, also makes UC models easy to read. The readability of UC models makes them accessible to stakeholders who do not have a strong technical background. This accessibility allows all stakeholders to have a strong and common understanding of a UC model, which will improve the validation efforts. The UC diagram provides an initial and very quick form of validation. Visually, it can be easily determined whether the requirements specifications are missing key functional requirements (UCs). Similarly, it can also be determined whether the requirements specifications contain unnecessary or redundant requirements. The UC descriptions are authored using templates and UNL, which improves their readability. The readability of UC descriptions allows its reader to better validate the underlying requirements.

➤ **Requirements Management**

UC modeling facilitates a number of managerial activities, for example:

- Tracking the progress of a project.
- Effort estimation (Anda et al. 2001b) and work allocation.
- Prioritization of requirements and scheduling (Firesmith 1999).

2.5 Thesis Contributions towards Improving UC Modeling

The following subsections will present a literature review and outline specific issues with UC modeling that led to the research presented in this thesis. The techniques proposed in these Chapters do not solve all UC modeling issues nor do they provide silver bullet solutions. The techniques however offer genuine solutions that contribute towards the two main goals previously mentioned in Section 2.4 and help alleviate a number of UC modeling issues.

2.5.1 Improving the Presentation of Functional Requirements in UC Models

UC models describe functional requirements as a set of interactions between a software system and its environment. In essence, UC descriptions state a set of workflows that will allow a system's users to benefit from its services. It is critical that designers have a common and precise understanding of what these workflows are. Otherwise they are in danger of building the 'wrong' system. In order to improve the understandability and readability of UC models, it is important to clearly define the underlying workflows, removing any source of

ambiguity and ensuring that all team members have a common and consistent understanding of these workflows. Traditionally, UC descriptions are authored using UNL. However, relying on textual descriptions is insufficient to fulfill such an important requirement. There are several factors attributing to such a shortcoming:

- (a) Workflows may contain concurrent flows, loops, branches and conditions. These elements are difficult to describe precisely using textual descriptions.
- (b) The problem described in (a) is compounded further if the workflows span several UCs, which is not unusual.
- (c) Analysts' writing skills vary significantly amongst individuals.

In Section 3.1, a new approach named AGADUC (Automated Generation of Activity Diagrams from UCs) was developed to overcome these limitations by graphically depicting workflows contained within UC descriptions. The UC descriptions are embedded with a structure named SUCD (Structured UC Descriptions) to facilitate AGADUC. SUCD also ensures consistency within the UC model.

2.5.2 Improving Consistency in UC Models

The large degree of informality contained in the UC descriptions often causes UC descriptions to be inconsistent with their corresponding UC diagrams. Moreover, inconsistencies may reside within the UC descriptions themselves. In a UC driven approach (Jacobson et al. 1995) such as the Rational Unified Process (RUP) (Kroll et al. 2003; Kruchten 1998), UC models are used to produce other UML

artifacts such as activity and sequence diagrams (El-Attar et al. 2006; Gomaa 2000, 2002; Jacobson et al. 1992; Larman 2001; Overgaard et al. 2005; Rosenberg et al. 1999). Hence, it is important to invest in producing high quality UC models that will yield the production of other high quality UML artifacts. Consistency is a key quality attribute of UC models. Ensuring the consistency between UC descriptions and their corresponding diagrams requires a great deal of discipline from analysts, which seldom exists. Moreover, producing consistent UC models has been chiefly dependant on the experience of analysts. The expertise of analysts in industry varies significantly. Often, junior analysts are required to develop UC models, which will be highly vulnerable to inconsistencies. The produced inconsistent UC models may potentially lead to the production of low quality software systems.

2.5.3 Consequences of Inconsistencies in UC Models

Many researchers have determined that inconsistencies in a UC model have harmful consequences. Inconsistencies can exist in UC models in various forms. The consequences of inconsistencies are discussed in the literature.

- The UC modeling defects outlined by Anda et al. (Anda et al. 2001a) indicate that inconsistency is a key category of defects which severely hampers the overall quality of a UC model. The consequences of the stated forms of inconsistencies were also outlined, which was shown to affect every aspect of the development process – from producing wrong, missing and inaccurate functionalities to producing systems that are difficult to test.

- Chandrasekaran (Chandrasekaran 2008) has explained that inconsistencies in a UC model are generally symptomatic of one of two problems; firstly, the UC model might be handling concepts that are not defined or understood properly. Secondly, there may be an ambiguity in the domain model.
- Lilly (Lilly 1990) and Bittner et al. (Bittner et al. 2002) outlined a number of inconsistencies that explicitly exist in UC diagrams as well as other types of inconsistencies that may exist throughout a UC model. These authors have also explained the harmful consequences of these inconsistencies. For example, in (Lilly 1990), it is shown that an inconsistent system boundary may cause designers to implement the behavior of entities external to the system. This, in turn, requires more effort from the development team than is actually required, causing the project to fall behind schedule and go over budget.
- Ambler (Ambler 2007) warns that inconsistencies in UC models are usually a sign of missing or vague information. The literature has repeatedly shown that teams often fail due to a lack of details in a UC model rather than too much detail (Bittner et al. 2002). Ambler also warns that too many inconsistencies may cause the UC model to become “out-of-date” and therefore rendering it useless. Therefore, UC models need to remain consistent to be effective in the development process.
- Consistency has always been a sought after and an essential quality attribute for UC models (Anda et al. 2001a; Ben Achour et al. 1999; Firesmith 1999; Jaaksi 1998; Kulak et al. 2000; Rosenberg et al. 1999). Reviewing UC models

is a highly recommended practice (Armour et al. 2000; Kulak et al. 2000; Schneider et al. 1998) useful to assure that UC models possess a great deal of consistency.

- Other researchers have devised techniques to incorporate and ensure consistency in UC models. McCoy (McCoy 2003) introduces a tool, which provides a template for modelers to input information about their UCs into a repository. The template ensures consistency during entry of the information. Ben Achour et al. (Ben Achour et al. 1999) compiled a set of styling and content guidelines to improve the quality of the UC descriptions. A number of these guidelines are either directly or indirectly aimed at ensuring consistency within the UC descriptions. Butler et al. (Butler et al. 2002) introduced the concept of refactoring UC models. Butler explained that refactoring can improve the consistency of the UC models. Ren et al. (Ren et al. 2004) has developed a tool that implements the refactoring concepts presented in (Butler et al. 2002).

2.5.4 Inconsistencies: A Closer Look

An obvious argument at that point would be: if the heart of the UC model is in the descriptions while the diagrams only serve as a visual roadmap then:

Why bother with the UC diagram? Why not just use the UC descriptions only to drive the development process? In such a case, when only the descriptions are considered, then ensuring the consistency between the UC descriptions and their diagrams is not important!

Even though this argument might be valid for very trivial systems, there still remain several problems if only the UC descriptions are considered. If the system is very complex, then a UC description might span over five pages to be adequately described (Bittner et al. 2002). In such a case, if a team member wanted to know about the actors associated with a given UC, it would be more efficient and accurate to simply look up this information in the diagram rather than going through several pages of text. UC diagrams are able to provide an overview of a system at a glance; while examining a set of UC descriptions cannot. Therefore, stakeholders might be misled about the general purpose of the system if the UC diagram did not accurately represent the descriptions. Hence, UC diagrams remain an indispensable component of UC models, and therefore if a UC diagram does not have an accurate representation of the descriptions, then this will lead to the design of a faulty system.

It can be concluded that it is desirable to minimize inconsistencies within UC models. Therefore, it is essential to devise a structure that will aid the production of consistent UC descriptions. It is also important that this structure can be used to ensure the consistency between the UC descriptions and their corresponding diagrams; while maintaining the ability of these diagrams to be understandable to all stakeholders, including “non-technical” stakeholders.

In Section 3.2, a structure is proposed to assist with the description of UCs. The proposed UC description structure is called SSUCD (**S**imple **S**tructured **U**C **D**escriptions). SSUCD serves as a guideline for authors in producing their

UCs. Moreover, the SSUCD form will allow the UC descriptions to be machine readable. A technique named **Reverse Engineering of UC Diagrams (REUCD)** was devised, which will systematically generate UC diagrams from UCs that are described in the SSUCD form. UC diagrams are developed at a much higher level of abstraction than UC descriptions. Hence, UC diagrams can be accurately reverse engineered from UC descriptions using REUCD. REUCD extracts limited information from UC descriptions to generate UC diagrams. Figure 2-2 shows an overview of SSUCD and REUCD. The REUCD process may also be reversed, whereby the UC diagram is initially developed and used to systematically generate ‘skeletons’ for the UC descriptions. Details about the events occurring in the UCs can then be manually completed. However, the theme throughout this thesis will be aimed at initially composing UC descriptions then systematically generating the diagrams from them, since UC descriptions contain all the information required to produce a complete UC diagram. A tool named SAREUCD (**S**imple **A**utomated **REUCD**) was developed to automate the REUCD process and increase the speed and accuracy of its application.

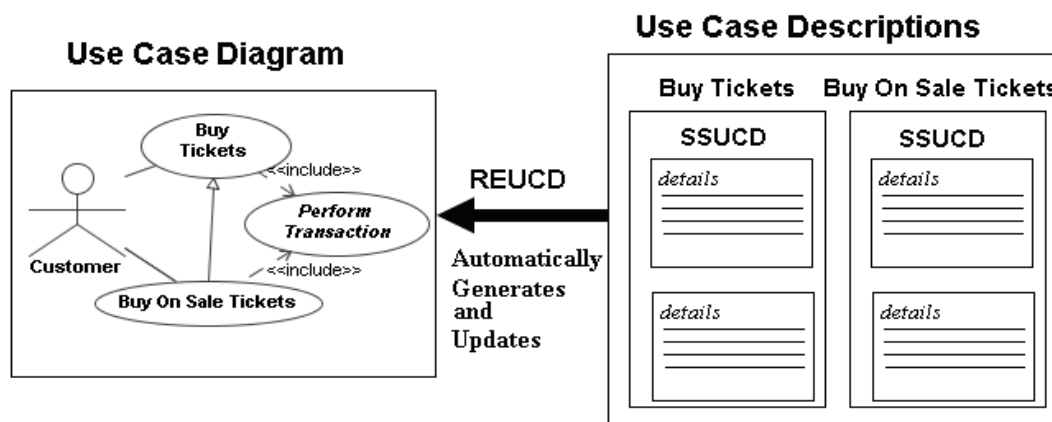


Figure 2-2: The application of the REUCD process to systematically generate UC diagrams from descriptions

Using the SSUCD structure and the REUCD process ensures that the UC descriptions and their diagram(s) are consistent with each other. For example, if the descriptions state that a certain UC is associated with a certain actor, then it will be ensured that an association relationship in the UC diagram will be depicted linking the given UC with the given actor. The SSUCD structure and the REUCD process do not however ensure that inconsistencies, present in the segments that are written in UNL, will be detected or eliminated. These segments require domain expertise to verify their consistency. For example, if a UC states that a theatre's seating capacity is 1200 while another UC states that the given seating capacity is 1400, then this type of inconsistency requires manual inspection (or review) to be detected and eliminated.

2.5.5 Reducing Inconsistencies in UC Models with SSUCD – An Empirical Evaluation

The SUCD structure, which was developed to facilitate the AGADUC process, can also be used to ensure consistency within a UC model (see Section 3.1.2). However, a preliminary experiment conducted has shown that SUCD is too complex to be effectively used to create consistent UC models. This is because the SUCD structure was primarily devised to facilitate the systematic generation

of UML activity-like diagrams from UC descriptions. The preliminary experiment compared the usage of SUCD with UNL to develop UC models; it involved 17 graduate Software Engineering students who voluntarily agreed to participate in this experiment. The results of the initial experiment indicated that the SUCD structure was too complex to utilize in comparison to UNL, causing subjects to inject many inconsistency errors into their UC models. UNL UC models contained an average of 2.71 inconsistency mistakes per student, while SUCD UC models contained an average of 3.98 inconsistency mistakes per student. Upon further analysis of the results, approximately 88% of the inconsistency errors present in the SUCD UC models were due to syntactical errors, the remaining 12% were due to incorrect omissions by the subjects. Further analysis revealed that approximately 67% (out of the 88%) of the inconsistency errors were caused by inappropriately utilizing structural elements that SUCD possesses which allow the systematic transformation of UC models to UML activity-like diagrams. These additional structural features are unnecessary for the purpose of only ensuring the consistency between UC diagrams and their corresponding UC descriptions.

Even though SSUCD is a much simpler version of SUCD, it was still unknown if it was simple enough to be accessible to its potential users. Therefore, a subject-based empirical study that evaluates the usability of SSUCD was conducted and presented in Section 3.3. The subjects chosen exemplify potential users whom do not have prior knowledge of SSUCD. None of the students who

participated in the initial experiment have also participated in the main experiment presented in Section 3.3.

2.5.6 Using Antipatterns to Improve the Quality of UC Models

The notation and guidelines for creating UML artifacts, including UC models, are clearly defined in (OMG 2005). However, mechanisms to construct semantically correct and verifiable diagrams are not discussed (OMG 2005). A UC model must accurately represent an analytical view of a system's functional requirements. In Chapter 4, an approach is presented that aims to tackle this issue by searching for the existence of antipatterns in UC models. Antipatterns represent debatable diagrammatic and textual structures. Their detection prompts a "review" of the debatable structures to either undertake corrective actions or to verify their correctness. The effectiveness of this technique is dependent on the original state of a UC model. Applying this technique will help bring a given UC model into a form that more accurately represents its system's functional requirements, ideally yielding a flawless UC model.

2.5.7 Towards Achieving the Benefits of Acceptance Testing for UC-Driven Development Processes

There are many benefits in using acceptance testing (Sauvé et al. 2006). Creating acceptance tests is a cost effective procedure, as it allows the customer to express, from a new viewpoint, a system's requirements through a set of tests, which in turn provides developers with a better understanding of a system's expected behavior. Ultimately, this will lead developers to build a system that more closely

meet the customer's expectations and requirements. Moreover, the resulting set of tests can evolve as a project's requirements develop and change. Acceptance testing can be used as a mechanism to define acceptable external quality. Acceptance tests can be used to track a project's progress and to determine when development is complete. User acceptance tests are created in the early phases to obtain customer approval and to drive the development process. These tests are constructed using simple, but sufficient, syntax to allow them to be understandable to non-technical stakeholders, while retaining the traditional machine executable nature of software tests. In addition, these tests partially fulfill the perceived need for increased customer involvement during requirements construction and system evaluation as the tests are produced in consultation with the customer (Agarwal et al. 2002; Good et al. 1989; Goodhue et al. 1959; Gould et al. 1983, 1985, 1991; Mantei al. 1988; Rosson et al. 1987). Acceptance testing is therefore a key component of the V-model development process and it is the basis for all other testing phases (see Figure 2-3).

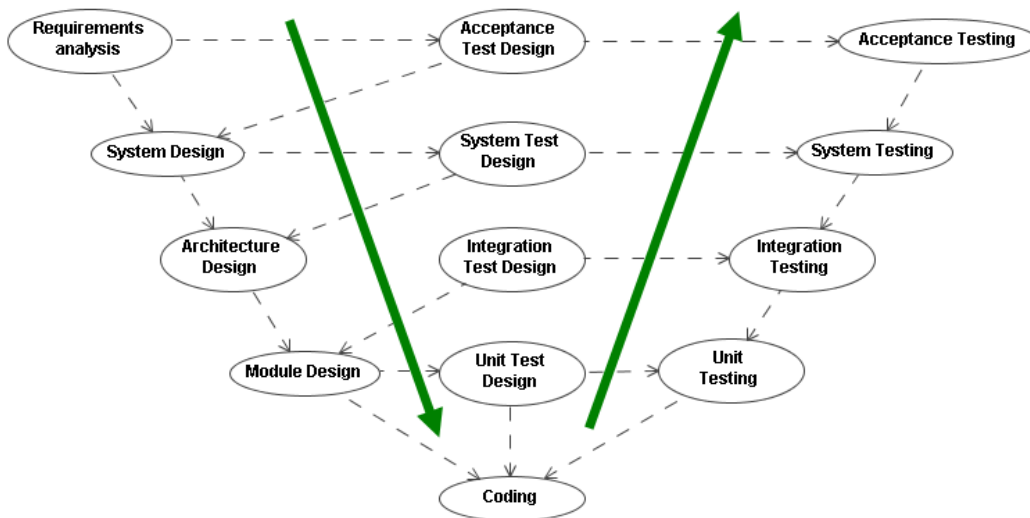


Figure 2-3: V-Model development process

Acceptance tests are developed from requirements artifacts. In agile processes, acceptance tests are often constructed from **User Stories** (USs) (Cohn 2004); however, large-scale development projects that deploy a more rigorous development process such as the V-Model do not utilize USs. It is common for large-scale software projects within a V-Model development process to deploy a model-oriented approach throughout the development process. The UML in particular has become the de-facto modeling language for large-scale object-oriented software development, which has resulted in the widespread use of UC models (OMG 2005) for requirements analysis and modeling. Yet, there lacks a process that allows analysts and customers to develop acceptance tests from UC models. In Chapter 5, such a process is defined; that is, using requirements artifacts normally available during early development phases to drive, or at least support, the production of user acceptance tests. UC models place an emphasis on system boundaries, and user-to-system expectations and interactions, providing an excellent source for user acceptance tests. The process defined in Chapter 5 is based on using UC models, robustness diagrams (Jacobson et al. 1992; Rosenberg et al. 1999, 2005) and domain models. A tool named UCAT (**Use Case Acceptance Tester**) was developed to provide automation support for executing acceptance tests developed through this approach. It is important to note that the approach proposed in Chapter 5 does not attempt to replace or improve upon any other approaches that develop acceptance tests. The approach provides a mechanism to develop executable acceptance tests based on UC models.

Throughout this thesis, each research area presented is accompanied with further literature review. Each literature review will be focused on presenting the motivation behind its corresponding research component. Wherever applicable, each literature review will also detail how other approaches compare to the approach presented.

Chapter 3

Using Structure to Improve the Quality of Use Case Models

3.1 Improving the Presentation of Functional Requirements in UC Models

3.1.1 Introduction

The AGADUC (Automated Generation of Activity Diagrams from UCs) process uses UC models to generate UML activity-like diagrams that represent the embedded workflows in the UC textual descriptions. This hybrid solution combines the notation of UC diagrams and activity diagrams and hence the resulting artifacts are called UCADs (UC Activity Diagrams). To facilitate this technique, UC descriptions were created using a structure named SUCD (Structured UC Descriptions). Another advantage of embedding SUCD's structural constructs in UC descriptions is that it can be used to systematically ensure consistency within a UC model. Automation support for this approach is provided by the tool AREUCD (Automated Reverse Engineering of UC Diagrams). A simplified Library system case study is presented to demonstrate the feasibility of the proposed approach and the application of AREUCD.

3.1.2 Related Work

Automated generation of activity diagrams from UC models is an active area of research. At the time of conducting this research there exist two tools that automate the generation of activity diagrams from UC models. The tools are called the “Optimal Trace” (Optimal Trace 2008) and “TopTeam Analyst” (TopTeam 2008). Both tools do not produce UML activity diagrams, but rather a simplified version of activity diagrams, which their developers refer to as ‘Flow diagrams’. ‘Flow diagrams’ is not a UML standard, however as mentioned earlier, they do bear a resemblance to UML activity diagrams. This Section presents a technique that was developed to automatically produce UCADs that adhere to the syntax rules and notation standard of UML activity diagrams. The technique overcomes limitations experienced by the two tools mentioned earlier by providing features to support the following activity diagram modeling concepts:

- Activities and other basic notation such as ‘start’ and ‘end’ of flows.
- Concurrencies.
- Diagram nesting: The technique provides a mechanism that will allow modelers to setup their diagrams at the abstraction level they need.
- Alternative flows that branch from a specific location in the activity diagram as well as alternative flows branching from general areas in the activity diagrams
- Exception handling flows that initiate from a specific location in the activity diagram as well as alternative flows initiate from general areas in the activity diagrams.

- Generation of separate activity diagrams for (basic flows + alternative flows), Extension Points and Sub-flows.
- Combination of UC modeling and activity diagram notations to support the modeling of relationship between UCs and between UCs and actors. Hence, allowing modelers to review the relational dependencies in a UC model.
- Support for abstract UCs and their corresponding activities.

3.1.3 The AGADUC process and the SUCD Structure

SUCD is a simple structure, with very limited syntax, that acts as a guideline to the authoring process and provides means to enhance clarity in the description of the UC workflows. The formal grammar of SUCD is presented in appendix A. SUCD provides mechanisms to precisely describe concurrent flows, looping, and branching and condition evaluation. The formality of SUCD allows our featured tool AREUCD to automatically generate the corresponding UC diagrams from a set of UC descriptions. Most authoring styles utilize a template to guide the authoring process. A study performed by (Anda et al. 2001a) has shown that the use of templates significantly increases understandability. Templates force analysts to consider and identify key aspects of each UC. The study has also shown that UC descriptions that utilize a template are easier to understand by its readers. A survey in (Anda et al. 2001a) of the different templates used by (Cockburn 2000; Harwood 1997; Jaaksi 1998; Kulak et al. 2000; Mattingly al. 1988; Schneider et al. 1998) has shown that even though each template is unique,

they all share common sections. Therefore, the SUCD structure is based on a template that contains these common sections (see Table 2-1). SUCD embeds certain structure in a select subset of the common sections, namely: (a) Use Case Name, (b) Basic Flow, (c) Alternative Flows, (d) Sub-flows and (e) Extension Points. This template can be easily altered and tailored to cater to any specific needs as long as it does not affect the five sections named previously. It is important to note that this Section focuses mainly the AGADUC process while only referring to SUCD. We strongly recommend to our readers to take a quick overview of the SUCD structure. An article containing the complete specifications of SUCD and its formal grammar is also located at (STEAM 2009b).

The AGADUC process is dependent on UC descriptions being described using the SUCD structure in order to produce UCADs. The use of UCADs minimizes the gap between the analysis phase and the design phase by improving the understandability of the workflows. Automation of this process provides a great advantage in that there will be no additional effort required to generate the UCADs or to ensure their consistency with the UC model. Changes applied to the UC model are automatically applied to the UCADs. Conversely, changes applied to the UCADs are automatically applied to the UC model. These advantages allow AGADUC to be utilized within agile software development processes, such as XP (Extreme Programming) (Beck 1999), as well as MDA (OMG 2005) (Model-Driven Architecture) software development processes such as RUP (Kroll

et al. 2003; Kruchten 1998). Figure 3-1 provides an overview of the concepts and advantages of the AGADUC process.

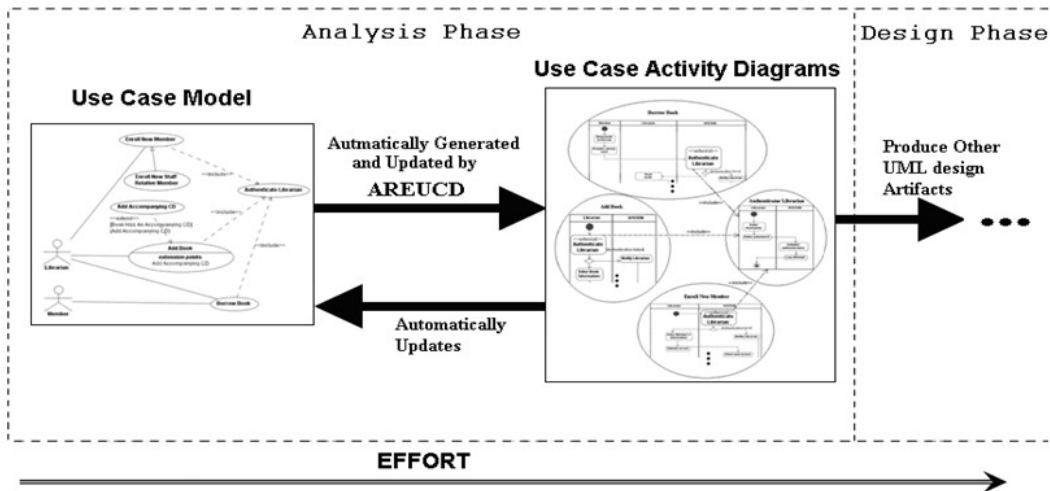


Figure 3-1: The AGADUC process

3.1.4 Mapping UC Descriptions to UCADs

In this section we explain the systematic mapping of the UC descriptions (in SUCD structure) to UCADs. This section will show how the notation of UC diagrams and activity diagrams can be integrated to model the workflows between different UCs and the workflow within each UC to enhance the overall readability of the analysis model. As mentioned in the previous section, there are only five sections in a UC description that are structured; Use Case Name, Basic Flow, Alternative Flow, Sub-flow and Extension Points Sections. For every UC, AGADUC generates a single activity diagram to represent the Basic Flow and Alternative Flows merged together. AGADUC also generates a separate activity diagram for each Sub-flow and Extension Points. All the activity diagrams are

generated simultaneously to show how they relate to each other (see Figure 3-1). The five sections are constructed using *Headers*, which is the basic building block. Minor syntactical features differentiate these sections to cater to their unique purposes. The following subsections will start by explaining the basic concept of a *Header*, and how it is used to create sequences of activities. Later, it will be shown how other syntactical features of SUCD can be used to graphically construct concurrent flows, loops, decision nodes and conditions.

- **Headers: The Basic Building Block**

Headers describe a conceptual task that needs to be performed. *Actions* enclosed in *headers* describe the steps that are required to perform the desired task. Each *action* enclosed in a *header* is directly represented by an *activity* in an activity diagram. *Actions* and *activities* have a one-to-one mapping. The {Enter Member Information} *header*, which is responsible for adding the new member's information into the system, is translated into the following *activities* shown in Figure 3-2.

- **Swimlanes**

Swimlanes are used to assign the responsibility of a given *action/activity* to a specific actor, unless it was performed by the system. A swimlane is created for each actor involved in an activity diagram of a flow, in addition to a swimlane created for the system. Each *action* is suffixed by the entity that is responsible for that *action*, whether it was an actor or the system itself (see Figure 3-2).

▪ **Concurrent Flows: Using AFTER and RESUME Statements**

The AFTER and RESUME statements are embedded before the first *action* in a *header*, and after the final *action* in a *header*, respectively. The AFTER statement is used to model the concept of flow *joining*. AFTER statements indicate that the *actions* of the corresponding *header* will not be performed unless the final *action(s)* of the stated *headers* in the AFTER statement are completed. This creates a synchronization bar to sink all the involved flows into the current *header* (see Figure 3-3).

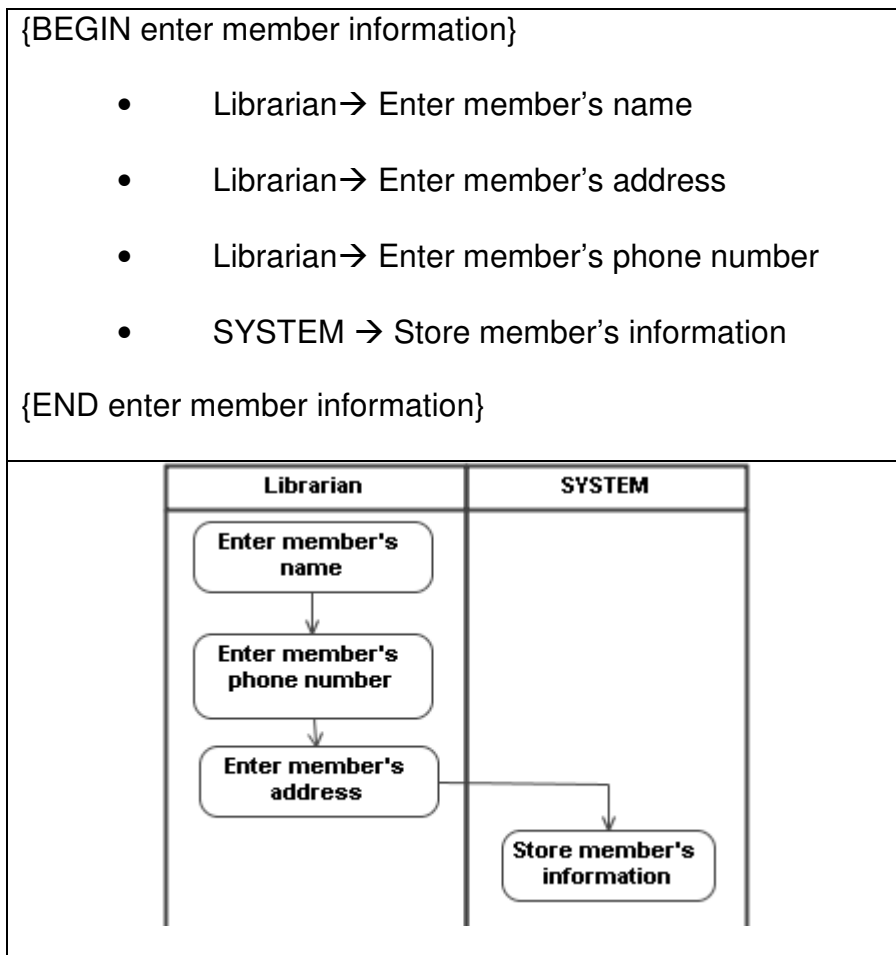


Figure 3-2: Conversion of a *Header* to activities in swimlanes

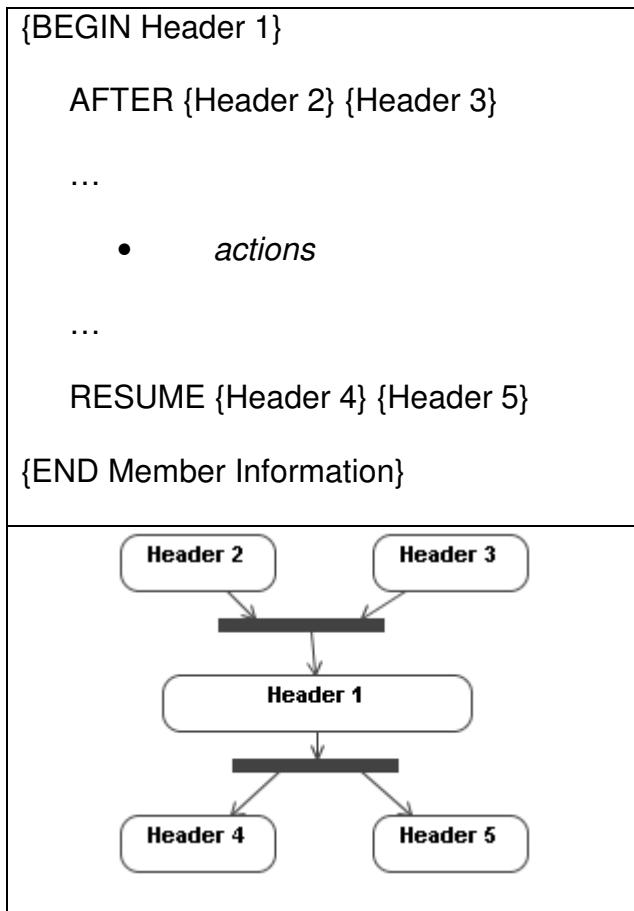


Figure 3-3: Transforming RESUME and AFTER statements

Conversely, the RESUME statement is used to model the concept of flow *forking*. RESUME statements indicate that when the final *action* in the current *header* is completed, the first *action(s)* in the *headers* stated in the RESUME statement must be performed concurrently. This creates a synchronization bar to dissect the flow to reach the corresponding *activities* (see Figure 3-3). Note that the *activities* labeled HeaderX are only high-level views of the sequences of *actions* they represent.

It can be easily inferred that the concept of looping can be easily modeled by stating an earlier *header* in a RESUME statement.

- **Branching and Condition Evaluation: Using AT and IF Statements**

AT and IF statements are used to facilitate the declaration of alternative flows. An alternative flow initiates from a discrete location in the Basic Flow upon a given condition being satisfied. The location is indicated by the AT statement, which states the *header* and the enclosed *action* where a corresponding condition will be evaluated. This creates a decision diamond immediately after the stated *action*. The condition itself is stated using the IF statement. The condition is depicted in square brackets on the branching flow. Alternative flows cannot have their own alternative flows, as it is believed that such situation should warrant the creation of a separate UC (extension UC). The AT and IF statements are also used to describe Extension Points (explained next). After all, Extension Points are just alternative flows that are too complex to fit within the context of the *base* UC, where it is also believed that it may obscure the real purpose of the UC. The alternative flows described within Extension Points may actually initiate from any type of flow within the *base* UC. Hence, the FLOW statement is used to state which flow in the *base* UC does the given *header* in the AT statement reside (see Figure 3-5). If an Extension Point is public, the condition stated by the IF statement is depicted on the *extend* relationship arrow connecting the *base* UC and the *extension* UC. Otherwise, if an Extension Point is private to a *base* UC, the condition is simply depicted on the flow arrow. The application of alternative flows and Extension Points are further illustrated later in the Library case study (Section 3.1.4).

- **Performing Sub-procedures: Using INCLUDE and PERFORM Statements**

Inclusion UCs and Sub-flows are used to model sub-procedural behavior that would otherwise cluster the Basic Flow of the *base* UC if it was described within the Basic Flow. Clustering the Basic Flow greatly reduces its readability. At any given point in a Basic Flow, if the behavior of an *inclusion* UC is required, the INCLUDE statement is used by providing the name of the *inclusion* UC. The *include* relationship notation is extended between the *activity* requiring the *inclusion* UC and the *inclusion* UC. Similarly, if the behavior described in a Sub-flow was required, the PERFORM statement is used by providing the name of the desired Sub-flow. The control flow notation is extended between the *activity* requiring the Sub-flow and the Sub-flow UCAD. This link is stereotyped with <<Perform>>. The application of these concepts is further illustrated later in the Library case study (Section 3.1.4).

- **Abstraction and Generalization**

Abstraction is also supported by SUCD, however since *abstract* UCs do not contain any *actions*, then there will be no corresponding *activities* generated, yielding to an empty UCAD. The generalization relationship does not model the workflow between UCs; hence, the UCADs of the parent and child UCs are separately modeled.

3.1.5 The Library System Case Study

The Library system presented in this section is simplified due to space restrictions. However, this case study illustrates most features and concepts of generating UCADs using AGADUC. The Library system only allows its members to borrow books. It is a requirement for a Librarian to authenticate into the system before being able to perform this transaction. If a Member attempts to borrow a book while having an overdue balance, the system provides a friendly reminder to the Member to pay off his balance. Figure 3-4 shows the UC diagram of the Library system.

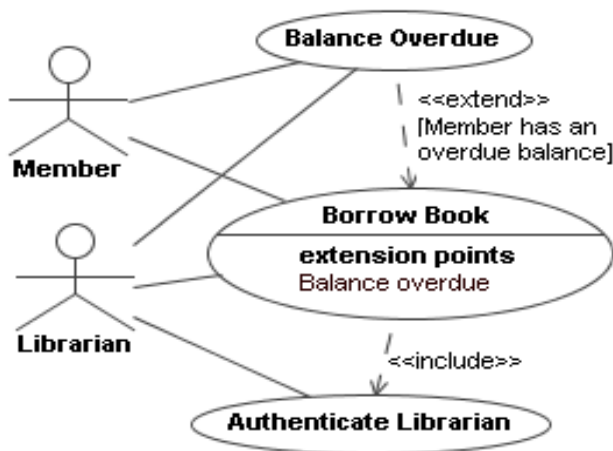


Figure 3-4: The Library system UC diagram

Table 3-1 shows the description of the Borrow Book UC in SUCD format. The remaining two UC descriptions as well as actor descriptions are shown in Appendix B.

Table 3-1: shows the UC description of the Borrow Book UC described in SUCD

Use Case Name:

Borrow Book

Brief Description:

This use case is initiated by a Member to allow that member to borrow a book. A Librarian is then involved to carry out the transaction.

Preconditions:

The book must exist

Basic Flow:

{BEGIN Use Case}

{BEGIN bring book to borrow}

- Member -> Brings the book he/she would like to borrow

- PERFORM Retrieve book information (2)

- Member -> Provides library card

- Librarian -> Scans member's card

{END bring book to borrow}

{BEGIN authenticate librarian}

- INCLUDE Authenticate Librarian (1)

{END authenticate librarian}

{BEGIN scan book}

- Librarian -> Scan's book's barcode

RESUME {update member's record} {update book's status} (5)

{END scan book}

{BEGIN update member's record}

- Librarian -> Updates the Member's record with the newly borrowed book

RESUME {END}

{END update member's record}

{BEGIN update book's status}

- SYSTEM -> Changes the book's status in the database to 'Borrowed'

{END update book's status}

{END Use Case}

Alternative Flows:

Sub-flows:

```
SUB-FLOW Retrieve book information
{BEGIN enter and retrieve book information}
• Librarian -> enter's the book's name or barcode
• SYSTEM -> retrieve the given book's information
from database
{END enter and retrieve book information}
```

Postconditions:

The number of borrowed books in the member's record is increased by one

Extension Points:

```
PRIVATE EXTENSION POINT
FLOW Basic Flow (3)
AT {scan book} (4)
• Librarian -> Scans the book's barcode
IF barcode cannot be scanned
{BEGIN enter barcode manually}
• Librarian -> Enters the book's barcode number
manually
{END enter barcode manually}
CONTINUE {update member's record} {update book's
status}
```

PUBLIC EXTENSION POINT

Balance overdue (6)

Special Requirements:

System must be online

In lay terms, the Borrow Book UC starts with the Member providing the book that he/she would like to borrow. The Librarian scans the book to retrieve its information from the system and then authenticates to get permission to carryout the borrowing transaction. Retrieval of the book's information is performed by the Sub-flow 'Retrieve book information'. As stated by the private Extension Point 'enter barcode manually', if the book's barcode cannot be scanned, the Librarian then enters the barcode manually. A public Extension Point is available to be 'plugged' into by an *extension* UC to model the behavior required to handle

an overdue balance. Other information such as the pre and postconditions of the UC is written in natural language, which can be obtained directly from the UC description.

This UC illustrates a number of key points. Firstly, to authenticate the Librarian, the *inclusion* UC Authenticate Librarian was included using the INCLUDE statement (see (1) in Table 3-1). Secondly, the Basic Flow that requires the behavior described in the Sub-flow in order to retrieve the given book's information, which was done using the PERFORM statement (see (2)). Thirdly, the private Extension Point provided, stated that the condition 'IF barcode cannot be scanned' must be evaluated at the Basic Flow (using FLOW (see (3)), at the *header* 'scan book' (using AT (see (4)), when the Librarian attempts to scan the book's the barcode. Fourthly, a public Extension Point was declared to allow an *extension* UC to handle an overdue balance situation (see (6)). The *extension* UC would then be Balance Overdue. Finally, the flow at the Basic Flow was forked after the book was scanned (using RESUME, see (5)).

3.1.5.1 Displaying UCADs and Their Dependencies

Traditional UC diagrams only show the relationships between the UCs. On the other hand, the greatest advantage of AGADUC is that it clearly shows how UCs are dependent on each other, and the internal details of how and when they interact with one another. Moreover, AGADUC also depicts the dependencies between different types of flows within a UC, such as that between a Basic Flow and a Sub-flow. Traditionally, such dependencies will need to be uncovered by

iterating through pages of text, leading the reader to lose sight of the dependencies between the workflows. In the Library system, there is one *include* and one *extend* relationship. The **Borrow Book UC** has a single Sub-flow and one private Extension Point, each having a separate workflow that is required by the Basic Flow. Figure 3-5 shows all the UCADs of the entire model.

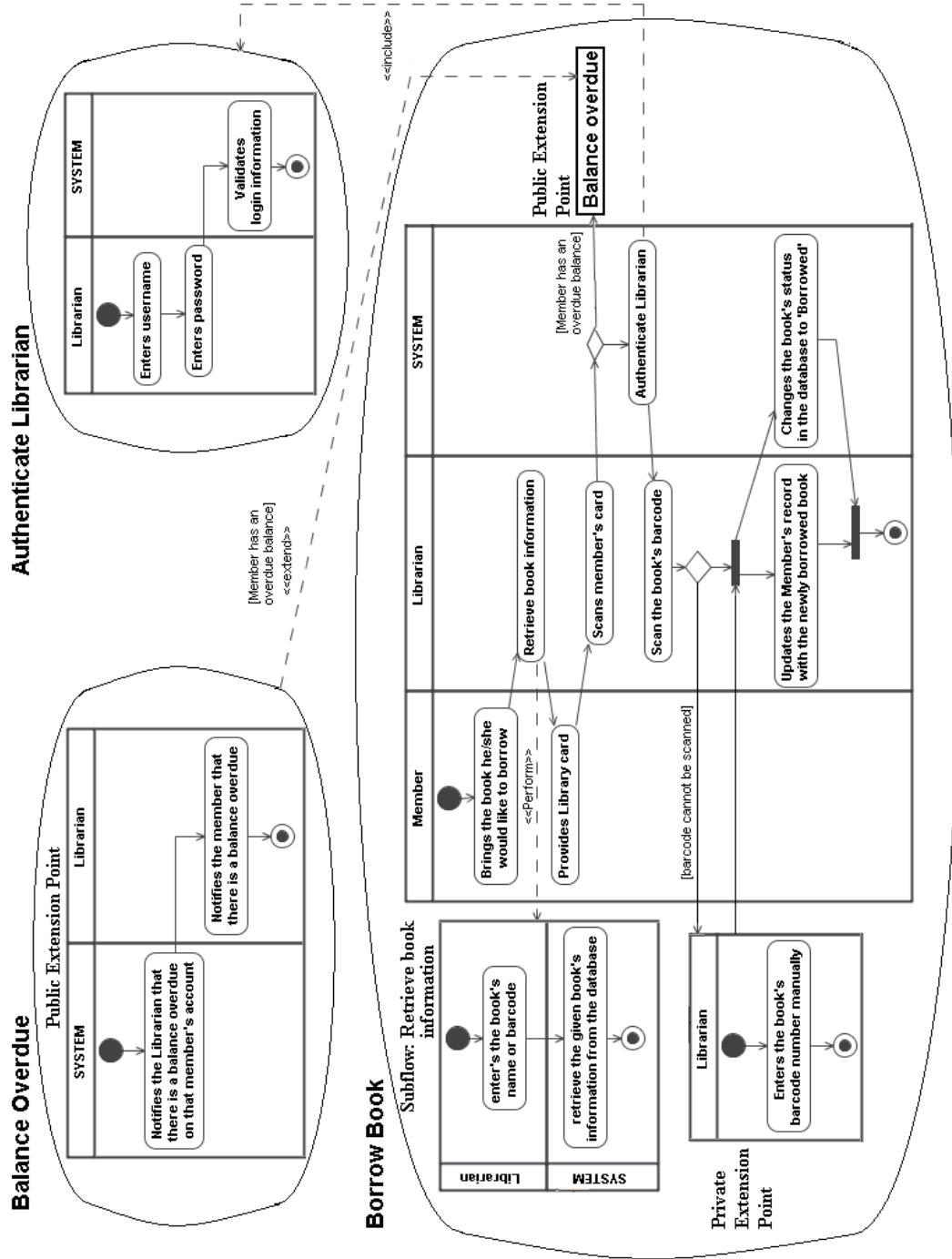


Figure 3-5: The UCADs of the entire UC model

3.1.6 AREUCD – Automating the ARADUC Process

Upon completing the descriptions file, AREUCD generates the UCADs for UCs, showing the entire set of workflows of each UC and between UCs. Figure 3-6 shows AREUCD after it has analyzed the UC descriptions and generated the UCADs. It can be shown that the *include* relationship can be traced to a specific UCADs inside the *base* UCs (see Figure 3-6). Upon examining the UCADs of these flows (middle pane), it is possible to trace the specific *action/activity* that triggered the *include* relationship. Such information cannot be retrieved from traditional UC diagrams. Similarly with the *extend* relationship, it can be shown that the extension behavior is provided by the Extension Point located at the Balance Overdue UC, whereby the location inside the *base* UC where the *extension* behavior may be inserted, and the condition that needs to be evaluated are shown.

AREUCD can generate the activity diagrams that are contained within the UCs. The activity diagrams are generated in XML format, which is the industry standard to store model data. The XML files representing the activity diagrams are also located at (STEAM 2009b). AREUCD has also computed all the workflows that spanned several activity diagrams, using PERFORM (7), INCLUDE (8) and Extension Points (9) (see Figure 3-6). Since the notation required to model UCADs is not yet available in UML modeling tools, it is not possible for AREUCD to generate XML files that show the relations between any two UCADs.

With AREUCD, there is very minimal effort required to apply the AGADUC process. Upon support availability for the required notation by leading UML modeling tools, AREUCD will in turn be upgraded to produce XML files that show complete UCADs.

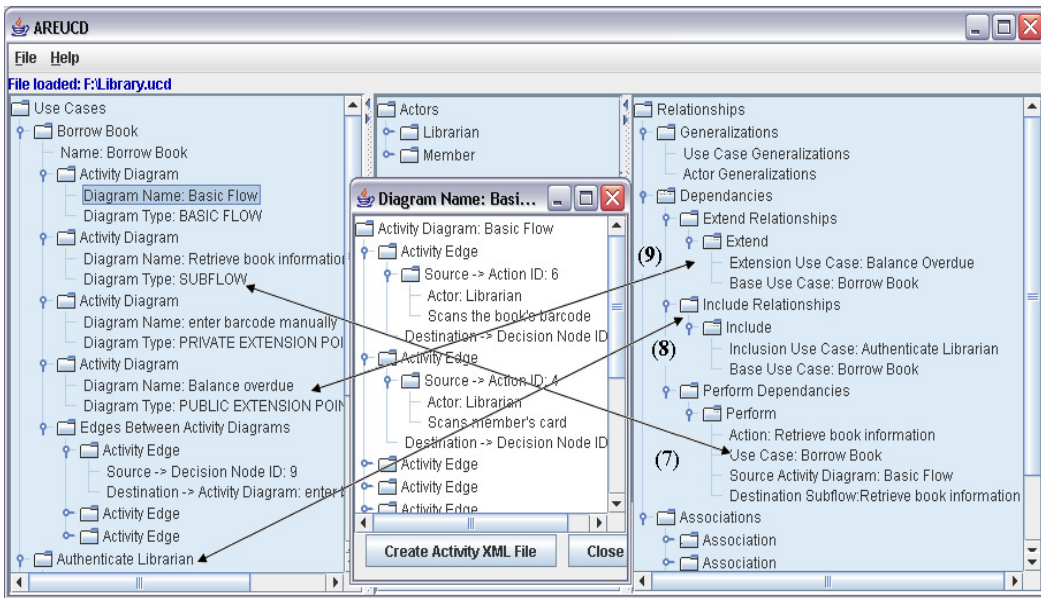


Figure 3-6: AREUCD generating the UCADs for the UC model

3.2 Producing Consistent Use Case Models via Reverse Engineering of Use Case Descriptions

3.2.1 Introduction

Informal UC models are prone to contain problems, which lead to the injection of defects at a very early stage in the development cycle. In this section, a structure named SSUCD is presented that will aid the detection and elimination of potential defects caused by inconsistencies present in UC models. The structure contains a small set of formal constructs that will allow UC models to be machine readable while retaining their human readability by retaining a large degree of UNL. This Section will also propose a process which utilizes the structured UCs to systematically generate their corresponding UC diagrams and vice versa. Finally, a tool was developed to provide support for the new structure and the new process. To demonstrate the feasibility of this approach, a simple study is conducted using a mock online hockey store system.

3.2.2 Related Work in UC Authoring

There have been many different approaches to authoring UC descriptions. Each approach is devised to describe UCs at different levels of detail and structure. For example, UCs can be described using a single short paragraph. Caution must be exercised while describing UCs in such form as this approach tends to assume that other stakeholders have a great degree of domain knowledge, which is not always the case. On the other hand, UCs can be described using a “full blown”

approach that mentions every possible detail. Some approaches structure the UC descriptions very carefully, while others do not incorporate any structure. Johansson (Johansson 2004) analyzed and discussed problems that arise when attempting to construct a UC model and write the corresponding descriptions of UCs for a weather station system. The problems were principally caused by a lack of guidelines for authoring UC descriptions. The paper concludes by urging for guidelines for UC modeling.

- A number of authors have developed such guidelines, principally: Ben Achour et al. (Ben Achour et al. 1999) examined two different types of guidelines; styling guidelines (SGs) and content guidelines (CGs). The styling guidelines were mainly derived from current best practices such as those presented in (Harwood 1997; Schneider et al. 1998). The styling guidelines are used to improve the quality of UC structures. On the other hand, content guidelines were mainly derived from linguistics, artificial intelligence and previous experiences in applying Case Grammars to requirements analysis. The content guidelines are used to indicate the expected contents of UCs. The authors present an evaluation of their work which comprises seven hypotheses, three of which are related to CGs and the remaining four are related to SGs. The experimental procedure involved 69 software engineers who had professional experience and participated in a half day presentation on UC authoring and modeling. The results of the study conclude that UC authoring guidelines generally improve the quality of the descriptions. The authors emphasize that even though authoring guidelines helped, they rarely lead to perfect UCs.

Therefore, the authors suggest that the UC descriptions should be checked whenever quality is an issue.

- Firesmith (Firesmith 1999) described a broader range of guidelines. These guidelines fall into the following categories: modeling tools and languages; modeling externals; modeling UCs; modeling UC paths; and general guidelines.
- A number of UC authoring guidelines have been devised to capture the requirements for special types of systems. Anderson et al. (Anderson et al. 2001) described styles of documenting business rules in UCs. Constantine et al. (Constantine et al. 1999) and Biddle et al. (Biddle et al. 2002) described styles that lead to ‘essential’ UCs. Wirfs-Brock (Wirfs-Brock 1993) has also promoted a conversational style of authoring UCs.
- Cockburn (Cockburn 1995) described a set of eighteen different styles of writing UC descriptions, collected while working upon various projects, matching one by Jacobson (Jacobson et al. 1992). The work presented by Bittner et al. (Bittner et al. 2002) further developed this style of UC authoring. However, this style in its current state experiences several limitations:

- a) The authoring style lacks the required amount of structure to allow the UC descriptions to be machine readable, which will impede the systematic process of:
- Generating UC diagrams from the descriptions.
 - Generating the ‘skeletons’ of UC descriptions from the diagrams.
 - Verifying the consistency between the UC diagram and the descriptions.
- b) There is no mechanism available to:
- Declare *generalization* relationships between UCs.
 - Declare *generalization* relationships between actors.
 - Declare *abstract* UCs.
 - Declare *abstract* actors.
 - Declare that a UC *implements* an *abstract* UC.
 - Allow an *extension* UC to reference the *base* UC it *extends*.

The structure SSUCD was developed to overcome the limitations outlined above. The key feature of SSUCD is that it will ensure the consistency between the UC descriptions and their corresponding diagrams.

3.2.2.1 Maintaining the Readability of UC Descriptions

The core feature behind the popularity of UC models is the great deal of UNL that the UC descriptions contain. The informality contained in UC descriptions makes it accessible by stakeholders who are not familiar with common programming jargon and acronyms. Customers are often not technical specialists and thus the

informality in UC descriptions allows them to read and review the UC descriptions and provide feedback. UNL is an indispensable component of UC descriptions. Unfortunately, it is impossible to formally analyze UC descriptions that are completely composed of UNL. UC descriptions become highly vulnerable to poor quality attributes such as inconsistencies, incorrectness and incompleteness. UC descriptions can be formally analyzed only if they adhere to a formal structure. However, describing UCs using only formal constructs will greatly reduce their readability and make them inaccessible to many stakeholders. Therefore, a tradeoff must exist between the amount of UNL and formal constructs that UC descriptions may contain. The SSUCD structure provides a hybrid solution to this problem. The SSUCD structure contains a very limited set of formal constructs, the minimal amount required, while allowing analysts the flexibility and liberty of using as much UNL as possible. The SSUCD structure will allow a great deal of formal analysis to be performed on the descriptions while retaining their readability. Further structure can be added to the SSUCD descriptions in order to generate other types of UML artifacts. For example, the SUCD structure (STEAM 2009b) adds more formality and structure to SSUCD descriptions to allow the systematic generation of activity diagrams.

3.2.3 Simple Structured UC Description (SSUCD)

In this section we describe the SSUCD structure. UCs described using the SSUCD structure contains four main sections, these are: (a) Use Case Name, (b) Associated Actors, (c) Description, (d) Extension Points and Extended Use Cases.

With the exception of the “Description” section, these sections utilize a handful of keywords to embed the required structure. All keywords are written in uppercase for readability purposes. The “Description” section on the other hand is populated using UNL to allow for maximum flexibility and expressiveness by UC authors. Other sections can be added to cater to specific needs; the additional sections must be contained as subsections of the “Description” section. There have many templates presented in the literature for describing UCs (Cockburn 1995; Harwood 1997; Jacobson et al. 1992; Kulak et al. 2000; Mattingly al. 1988; Schneider et al. 1998). The structured sections incorporated by SSUCD are the common sections found in many templates presented in the literature.

3.2.3.1 A Brief Introduction to the Elements of SSUCD

For a fully detailed reference guide to SSUCD and its syntax, we refer interested readers to (STEAM 2009). The subsequent sections will briefly present the structural elements of SSUCD and how they are used to map UC descriptions to diagrams (see Table 3-2), which is further illustrated using the Online Hockey Store System presented in Section 3.2.7.

(a) Use Case Name Section:

The “Use Case Name” section states characteristic properties about a given UC. This section starts with the label “Use Case Name:”

Structural elements and keywords:

a) The name of the UC:

The “Use Case Name” section must state the name of the UC.

b) If the UC is *abstract*:

This is stated using the keyword ABSTRACT. If the UC is not *abstract* then this keyword is omitted. On the other hand, if the UC implements an *abstract* UC, then this is stated using the keyword IMPLEMENTS followed by the name of the *abstract* UC. Similarly, if the UC does not implement any *abstract* UCs, then this keyword is omitted.

c) If the UC specializes other UCs:

This is stated using the keyword SPECIALIZES followed by the name of the parent UC. If the UC does not have any parents, then this keyword is omitted.

Mapping information and examples:

a) The name of the UC:

The name stated in the “Use Case Name” section must have a UC symbol (an oval) in the diagram with a matching name (see Figure 3-7).



Figure 3-7: UC name and its representation

b) Abstract UCs and their Implementation:

The name of an *abstract* UC is displayed in *italic* font in the diagram. A UC implementing an *abstract* UC creates a *generalization* relationship arrow in the diagram, originating from the implementing UC and directed towards the *abstract* UC (see Figure 3-8).

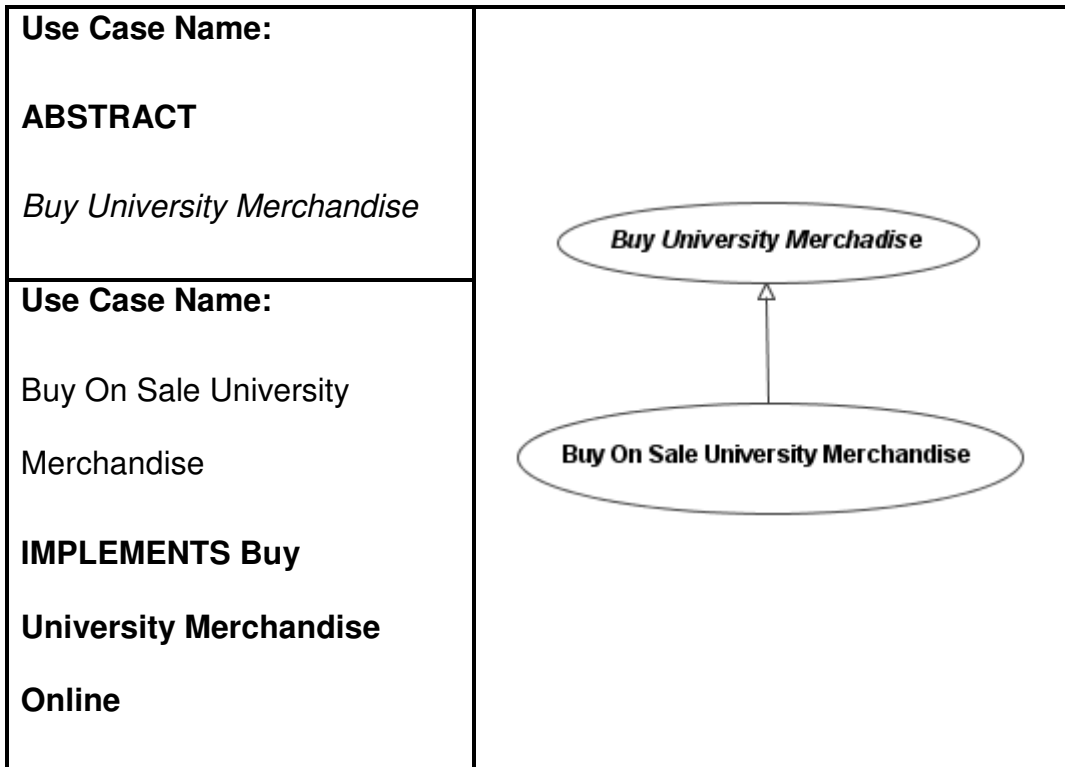
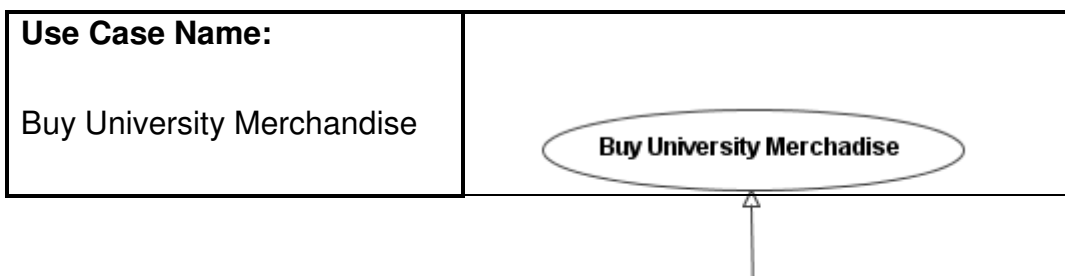


Figure 3-8: Abstraction and implementation in UCs and their representation

c) Generalization Between UCs:

The UC name, as *specialized* by a child UC, creates a *generalization* relationship link between the involved UCs, originating from the child UC and directed towards the parent UC (see Figure 3-9).



Use Case Name:	
Buy On Sale University Merchandise	
SPECIALIZES Buy University Merchandise	

Figure 3-9: Generalization between UCs its representation

(b) Associated Actors Section:

Actors are associated with UCs to perform the described behavior and to achieve a certain goal. Actors can be associated with UCs for various reasons. Each UC must specify the actors that are involved with it. The “Associated Actors” section is used to list the involved actors with only commas separating them.

Mapping information and example:

Actors listed in this section must have an association relationship link connecting the UC and the corresponding actors in the diagram (see Figure 3-10).

Example:

Use Case Name:	<pre> graph TD Member((Member)) --- UC((Enroll New Member)) Librarian((librarian)) --- UC </pre>
Associated Actors:	
Librarian, Member	

Figure 3-10: Associations between UCs and actors and its representation

(c) Description Section:

The “Description” section contains the core behavior of the UC. As mentioned earlier, the “Description” section is intentionally designed to be populated using natural language to allow UC authors utmost flexibility with respect to describing their UCs. Another reason is to minimize the amount of learning required by the users of SSUCD. If an author needs to add a new section, the new section is simply written using natural language as part of the “Description” section.

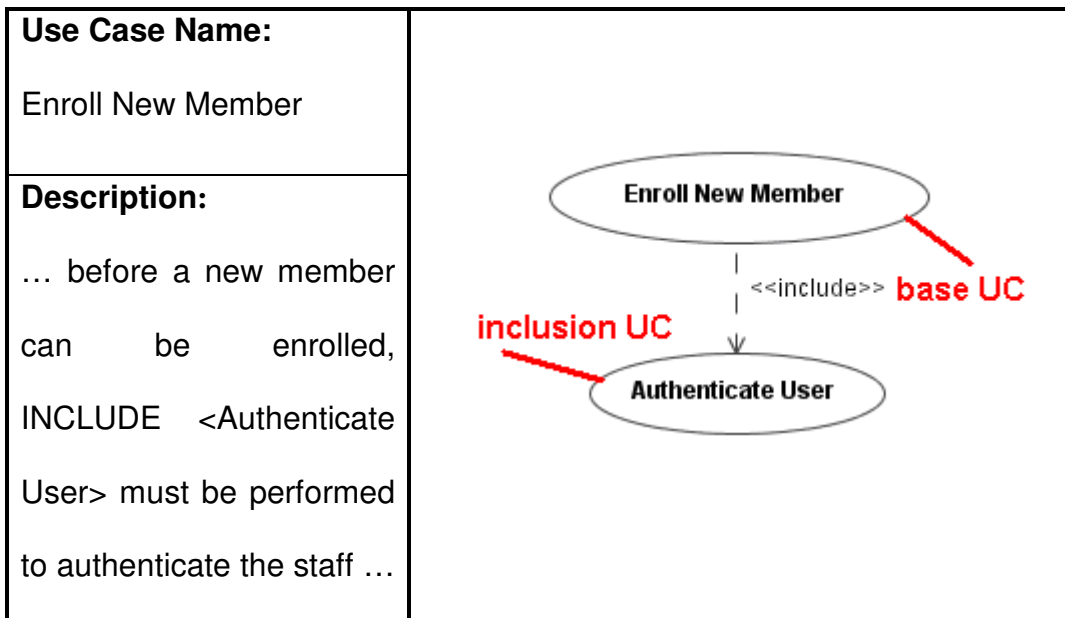


Figure 3-11: The *include* relationship represented in the UC description body

Structural elements and keywords:

There is only one keyword in this section which states that the given UC *includes* another UC. An *include* relationship is stated using the keyword INCLUDE followed by the name of the *inclusion* UC enclosed in angled brackets “INCLUDE <*inclusion* UC name>“.

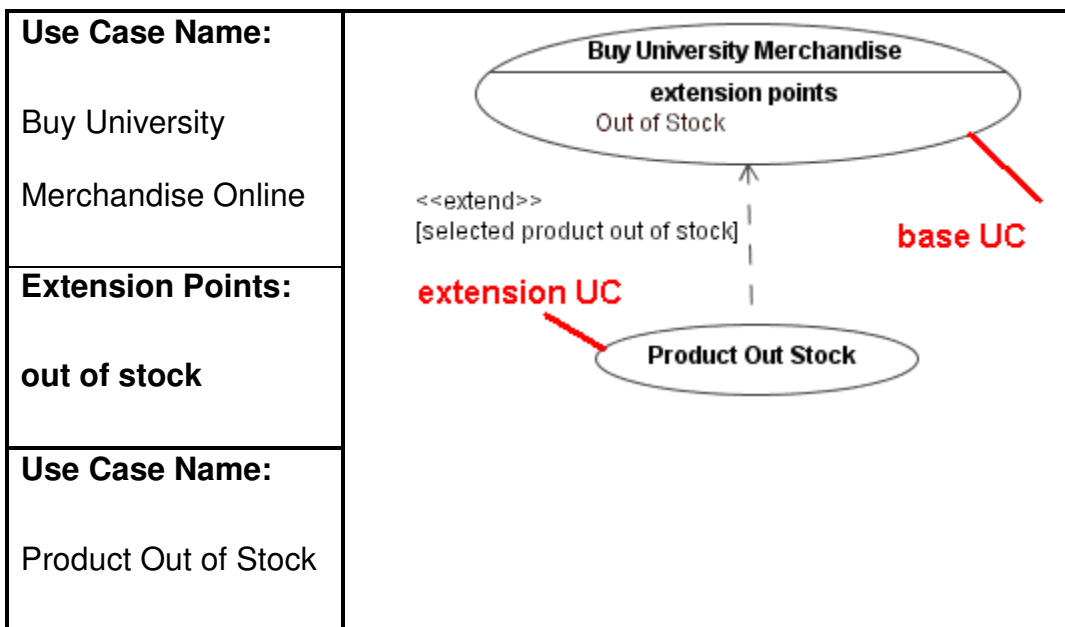
Mapping information and example:

An INCLUDE statement present in the “Description” section of a UC creates an *include* relationship link originating from the *base* UC and directed towards the *inclusion* UC stated in the INCLUDE statement (see Figure 3-11).

(d) Extension Points Section and Extended Use Cases Section:

The “Extension Points” section lists all the public extension points that belong to the given UC. Although there are two types of extension points; public and private, only public extension points appear on the UC diagram. Hence, private extension points can be described using natural language within the Description “section” without the need to add further structure. The “Extended Use Cases” section lists all the UCs that the given UC *extends*.

Example:



<p>Extended Use</p> <p>Cases:</p> <p>Base UC Name: Buy University Merchandise Online</p> <p>Extension Point: out of stock</p> <p>IF selected product is out of stock</p>	
--	--

Figure 3-12: The *extend* relationship represented in the UC description body.

Structural elements and keywords:

- The Extension Points Section

Base UCs that are extended should not have any knowledge of their *extension* UCs. *Base* UCs only provide public extension points for *extension* UCs to specify the locations where the extended behavior will be inserted. This is because *base* UCs are expected to be complete even without the incorporation of the *extension* UCs. Public extension points listed under an “Extension Points” section are separated using carriage return.

- The Extended Use Cases Section

Conversely, *extension* UCs are expected to have knowledge of the *base* UCs they *extend*. The “Extended Use Cases” section lists the *base* UCs that the given UC

extends. An *extended* UC is stated using the keyword “Base UC Name:” followed by its name. If an *extension* UC *extends* a base UC at a given public extension point, the extension point is stated using the keyword “Extension Point:” followed by the name of the extension point. Therefore, using the “Extension Point” construct is optional since stating a public extension point for a given *extend* relationship is optional. If a condition needs to be set for an *extend* relationship, this is stated using the keyword “IF” followed by the condition written in natural language. Specifying a condition for an *extend* relationship is optional. Hence, using the “IF” construct is also optional (see Figure 3-12).

3.2.3.2 Formalizing the SSUCD Structure Grammar

It is essential for the grammar and constructs of the SSUCD structure to be formalized. Formalizing the SSUCD structure will provide a strict guideline to UC authors when composing UC descriptions, so that there is no disagreement or ambiguity as to what is allowed and what is not. The grammar of the SSUCD structure is defined in E-BNF and can be located at (STEAM 2009).

3.2.4 Consistency and Mapping Rules between UC Descriptions and Diagrams

In this section we will introduce the REUCD process which is used to systematically map SSUCD’s structural constructs to diagrammatic notations that form UC diagrams. This systematic process is automated using the tool

SAREUCD (see Section 3.2.5), which will ensure the consistency and speed of the process.

The process of generating UC diagrams from UC descriptions and vice versa is analogous to generating complete and accurate UML class diagrams from code and generating code structures from UML class diagrams. The reason UML class diagrams cannot be used to generate complete programs is because they act as a visual summary of a program's static structure. UML class diagrams are at a higher level of abstraction compared to code. On the other hand, a complete program will contain more than enough details required to generate complete and accurate UML class diagrams.

UC descriptions (analogous to code) contain far more details than UC diagrams (analogous to class diagrams). UC diagrams are at a higher level of abstraction than the descriptions. Therefore, given a set of UC descriptions, a complete and accurate UC diagram can be systematically produced (see Figure 3-13a). However, if modelers choose to create UC diagrams manually first, which is often the case; a 'skeleton' of the UC descriptions can be systematically produced (see Figure 3-13b). Detailed descriptions of the UC are later added manually by analysts to 'flesh out' the generated 'skeletons'. After the UC descriptions are complete, an updated version of the UC diagram can be systematically generated. Users of SSUCD and REUCD will not be burdened with performing these transformations since they will be carried out by the tool SAREUCD.

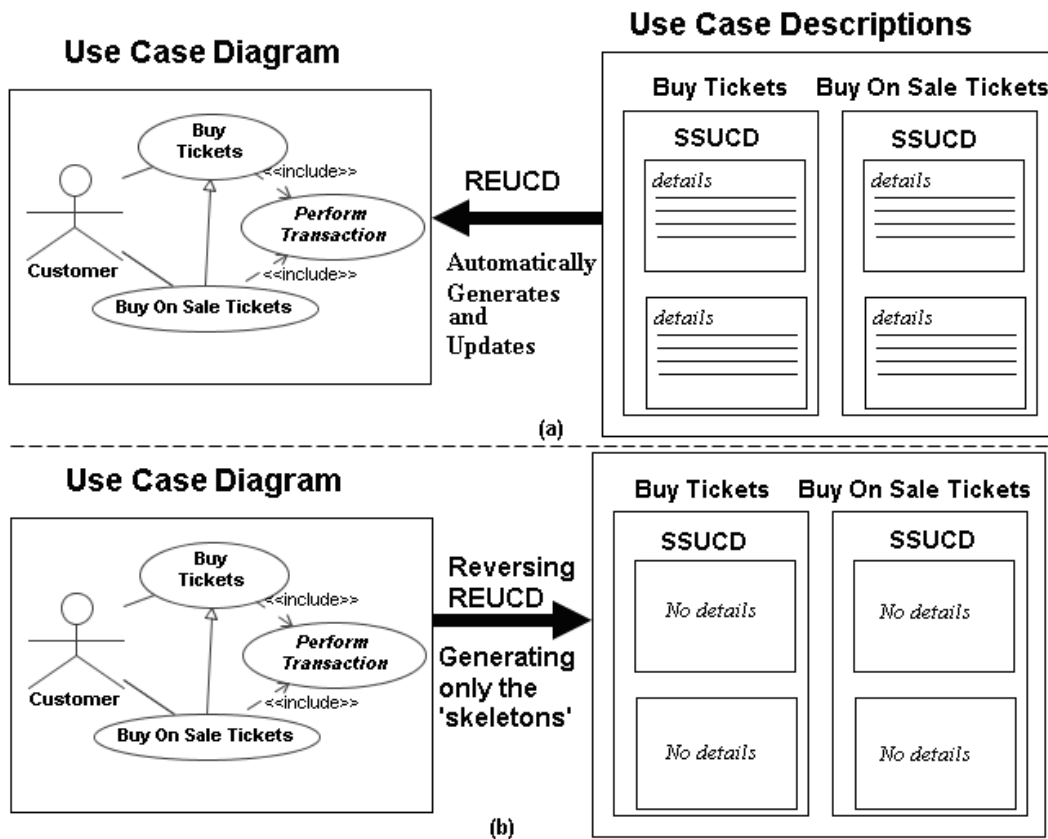


Figure 3-13: Systematically generating UC diagrams from descriptions and description skeletons from diagrams

Consistency rules and mapping concepts between UC description structures and UC diagrams are shown below:

- **Use Case Name:**

- 1) A use case description with a given use case name generates a use case in the diagram with a matching name.
- 2) Every use case description must have a corresponding use case in the diagram with matching names (see Table 1-1).
- 3) Every use case in a diagram must have a corresponding use case description with matching names (see Table 1-1).

- **The *INCLUDE* Statement:**

- 1) An *inclusion* use case in a diagram must be initiated at least once by each *base* use case that *includes* it, using the *include* statement in the Description section.
- 2) An *INCLUDE* statement in a *base* use case must refer to a use case that exists in the diagram
- 3) An *INCLUDE* statement in a *base* use case results in an *include* relationship link between the *base* use case and the *extension* use case.

- **Public Extension Points:**

- 1) At the *base* use case, any depicted public extension points must be stated under the Extension Points section.
- 2) At the *extension* use case, the Base UC Name must state the name of a use case that exists in the use case diagram.
- 3) The use case name indicated under the Base UC Name section must be connected to the given use case in the use case diagram using an *extend* relationship.
- 4) The extension point stated in the Extension Point Section must exist at the *base* use case stated in the Base UC Name section stated right above it.
- 5) If an 'IF' statement is used, the condition stated must be depicted as a UML condition in the use case diagram as part of the *extend* relationship link.

- **Associations Between Actors and Use Cases:**

- 1) An actor must be depicted in the diagram.
- 2) An actor must be linked with that use case using an *association* relationship in the diagram.

- **Generalization Between Use Cases:**

- 1) A use case name stated as *generalized* in another use case's Use Case Name section creates a *generalization* relationship link between the involved use cases.
- 2) A *specializing* use case must refer to a different use case that exists in the diagram
- 3) A *specializing* use case must refer to a use case in the diagram that is *specializes*.
- 4) The *specializing* use case must have a *generalization* relationship directed towards the *generalized* use case in the diagram

- **Abstract Use Cases and their Implementation:**

- 1) The name of an *abstract* use case is displayed in *italic* font in the diagram. A use case implementing an *abstract* use case creates a *generalization* relationship arrow in the diagram, originating from the implementing use case and directed towards the *abstract* use case.
- 2) An implementing *concrete* use case must refer to an *abstract* use case that exists in the diagram.

- 3) An implementing *concrete* use case must refer to an *abstract* use case that it implements.
- 4) The entire *header* tree structure of the *abstract* use case must exist and be implemented in the *concrete* use case.
- 5) A use case description written in *abstract* form (*italics*) must have a corresponding use case in the diagram with its name displayed in *italics*.
- 6) An *abstract* use case in the diagram must have a corresponding use case description written in *abstract* form (*italics*).

- **Generalization Between Actors:**

- 1) An actor name stated as *generalized* in another actor's Actor Name section creates a *generalization* relationship link between the involved actors.
- 2) A *specializing* actor must refer to a different actor that exists in the diagram
- 3) The *specializing* actor must have a *generalization* relationship directed towards the *generalized* actor in the use case diagram.
- 4) Every actor description with a given name must have a corresponding actor in the diagram with a matching name.
- 5) Every actor in the diagram with a given name must have a corresponding actor description with a matching name.

3.2.5 Tool Support Using SARUECD

Tool support is essential for the effective application of the REUCD process. For a highly complex software system, the corresponding UC model may contain up

to four hundred UCs. UCs are not sorted in any chronological order. Relationships linking UCs with other UCs and actors are also not sorted in any fashion either. Therefore, performing the REUCD process for such a system manually is a very cumbersome task that is error prone. Even for a relatively smaller UC model, one that contains twenty UCs, the application of the REUCD process is still vulnerable to mistakes.

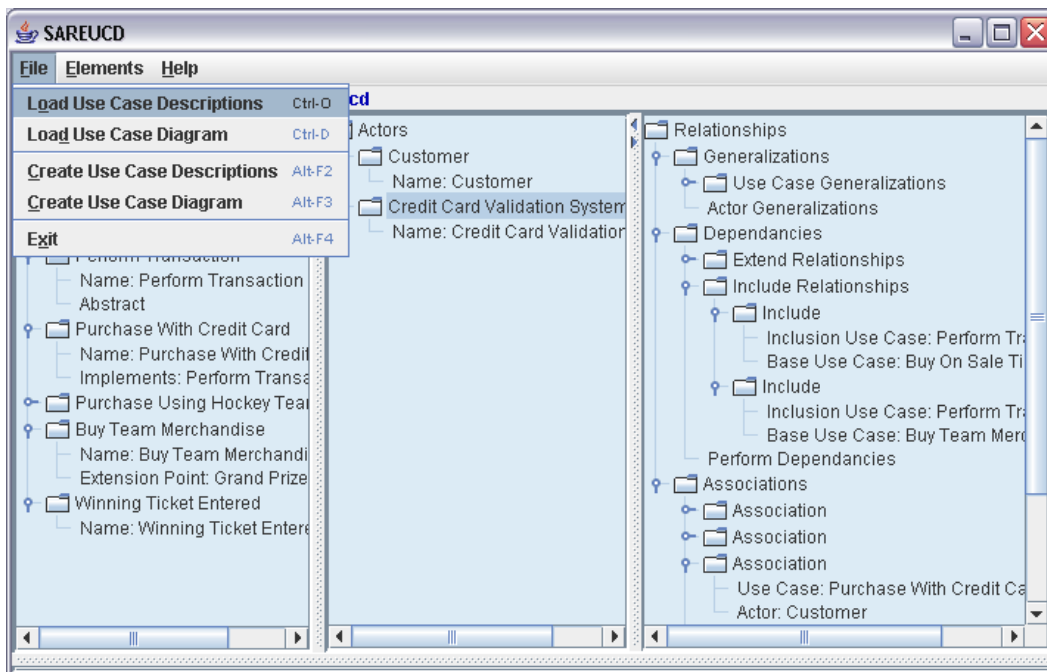


Figure 3-14: A screenshot of SAREUCD after transforming the descriptions to an object model

The tool SAREUCD (Simple Automated REUCD) supports the generation of UC diagrams from UC descriptions and vice versa (see Figure 3-14). In order to generate UC diagrams from UC descriptions, SAREUCD is loaded with a UC description file. SAREUCD parses through the descriptions of all the given UC

descriptions and actors and generates a file containing the corresponding UC diagram. The UC diagram is generated in XML in order to be viewable by most UML modeling tools. However, since the format of the generated XML files generated by UML modeling tools vary, the XML files generated by SAREUCD is only viewable by MagicDraw 10.5. Conversely, in order to generate UC description 'skeletons', SAREUCD is loaded with UC diagram file. The UC diagram can be generated by a UML modeling tool. The UC diagram must be in XML format; however this is not an issue since almost all UML modeling tools generate information about their models in XML format. Upon parsing the diagram or description files, the properties of the given UC model is displayed.

It is impractical to require or expect UC authors to spend a great deal of time and effort learning the syntax of SSUCD and its consistency and mapping rules, especially since UC authors often have a business background rather than a technical background. Even if the syntax of SSUCD and its mapping rules were understood, creating the UC descriptions and their diagrams manually is error prone. Authors may inject many syntactical and inconsistency errors in the descriptions. It is highly desirable to reduce the time and effort spent learning SSUCD. Therefore, SAREUCD provides a simple GUI interface for creating and editing UC descriptions (see Figure 3-15). This saves authors the time to learn many keywords and other syntax rules.

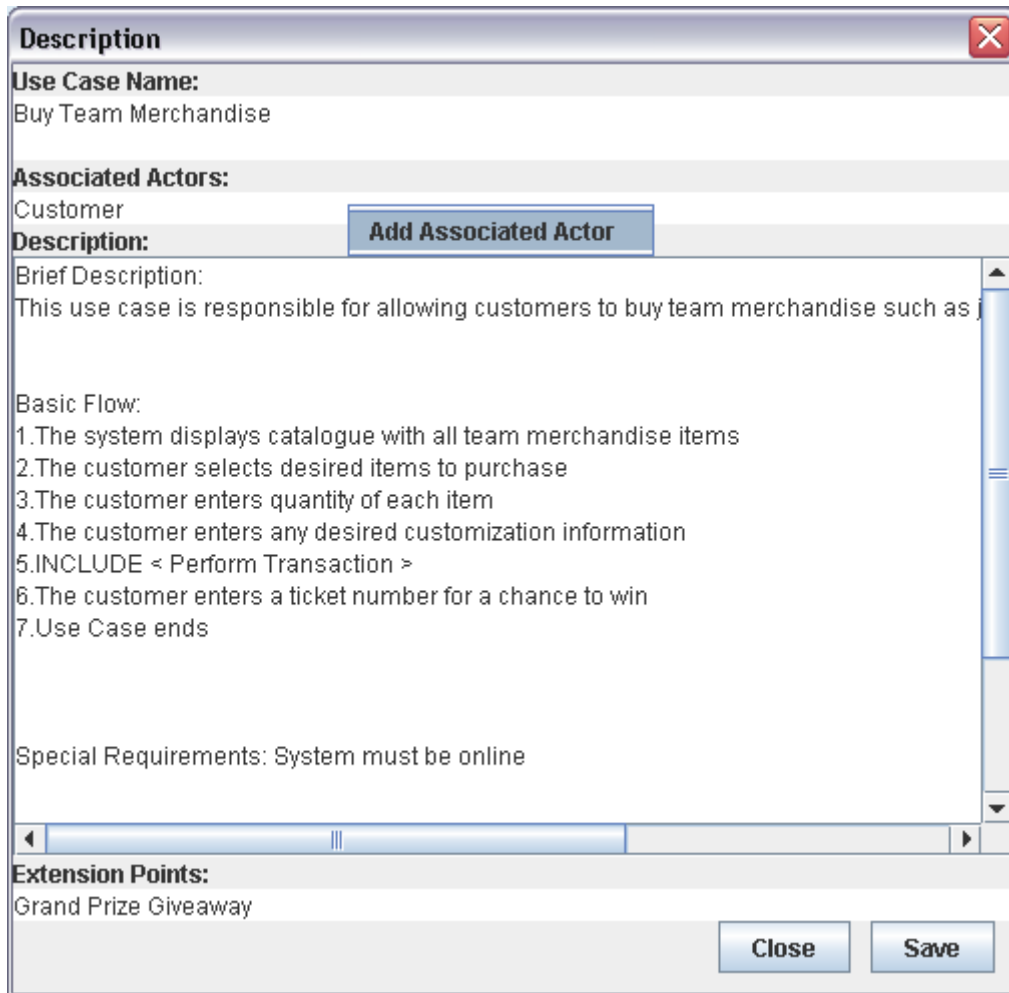


Figure 3-15: A UC description

3.2.6 SSUCD Modeling Language Design

Languages are designed to achieve a purpose, whether it is to create programs or models. To create a high quality modeling language, certain quality principles must be considered and embedded into the modeling language. The presence of these quality attributes in any modeling language is essential to its usability and its adoption for widespread use. The literature has provided many guidelines for constructing languages. Paige et al. (Paige et al. 2000) presented guidelines and quality principles specifically for modeling languages. These quality principles

are summarized in Table 3-2. This section will discuss the design of the SSUCD modeling language and its adherence to these quality principles.

Table 3-2: Quality principles that should be present in modeling languages

Simplicity	The language does not contain any unnecessary complexity.
Uniqueness	There are no overlapping features or redundant ones.
Consistency	The language elements and features allow the required goals to be met.
Seamlessness	The ability to generate code from the models.
Reversibility	Changes at any point in the development can be propagated back to the models.
Scalability	The ability to model large and small systems.
Supportability	The ability for humans to utilize the language and the availability of tool support.
Reliability	The language promotes the development of reliable software.
Space economy	Models produced must be concise, showing the required information without clustering the view.

Simplicity

The fundamental purpose of SSUCD is to ensure the consistency between the UC diagrams and their descriptions. All language constructs are designed with this goal in mind. If there are any segments in the UC descriptions that are not reflected in the diagrams, then they are not structured. Instead, they are populated using natural language, which is the original and most flexible method of authoring. The following is a summary of the entire list of constructs provided by

SSUCD, sorted by their corresponding section, and how they affect the presentation and the consistency between the descriptions and the diagrams (see Table 3-3):

All language constructs are aimed towards achieving the ultimate goal of consistency between the descriptions and the diagrams. There are no constructs in SSUCD that do not contribute towards this goal. Furthermore, the grammar indicates that many sections, such as the “Extension Points”, can be entirely omitted if not required, which further simplifies the authoring task.

Uniqueness

As shown above in Table 3-3, there are no overlapping features provided within SSUCD. All language features serve a unique purpose and are vital to ensure consistency, hence there is also no redundant features.

Table 3-3: A summary of SSUCD’s language constructs and their purposes

Section	Keyword	Diagram representation
Use Case Name	ABSTRACT	<i>Abstract</i> UCs appear in italic font in the diagrams.
	SPECIALIZES	Results in the creation of generalization relationship links in the diagrams.
	IMPLEMENTS	Results in the creation of generalization relationship links in the diagrams.

	The name of the UC in natural language	A UC with the given name is shown in the diagram.
Description	INCLUDE	The INCLUDE statement can be embedded within the text, and it will result in the creation of an <i>include</i> relationship link in the diagram.
Extended Use Cases	Base UC Name	Results in the creation of an <i>extend</i> relationship link in the diagrams.
	Extension Point	Optional to the user. The extension point name is displayed on the <i>extend</i> relationship link.
	IF	Optional to the user. The condition is displayed on the <i>extend</i> relationship link.
Extension Points	The name of a public extension point	Results in the display of an extension point within the oval of the given UC in the diagram.

Consistency

Using the list of features provided by SSUCD, UC authors can ensure the consistency between the UC descriptions and the diagrams. This is evident by the ability of the tool SAREUCD to generate UC diagrams from the descriptions and vice versa.

Seamlessness

This quality is intended for design modeling languages that are required to provide an easy and direct transition to code. Hence, this quality is not directly applicable to SSUCD since it is an analytical modeling language that is not intended to show a solution which results in code, but rather to provide an analytical view of what the system is required to do. However, for the purposes of SSUCD, it can be shown from Table 3-3 that SSUCD constructs can be mapped directly to the notation of UC diagrams. Therefore, for a given set of descriptions, there is no complex computation required to develop their corresponding diagram.

Reversibility

The REUCD process utilizes the SSUCD constructs to systematically generate UC diagrams from the descriptions. Moreover, the REUCD process can also be reversed to systematically produce UC description (skeletons) from UC diagrams. Both these process (forward and reverse) are automated using the tool SAREUCD. Reversibility of SSUCD is discussed in great detail at (STEAM 2009).

Scalability

UC modeling, in its current form, is used to model very large systems. However, these large models suffer from very poorly written descriptions since they are embedded with numerous inconsistency errors. SSUCD does not impede or hinder the production of UC descriptions. In fact, it provides an interface that guides the author while developing the descriptions. SSUCD, along with SAREUCD, encourages the author to consider aspects that would normally be ignored if the UCs were to be authored in a traditional fashion. For example, if a given *base* UC *includes* another *inclusion* UC, SSUCD requires the authors to consider where exactly in the behavior of the *base* UC will the behavior of the *inclusion* UC be performed. With regards to the additional effort required for creating syntactically correct descriptions, and avoiding the injections of human errors, SAREUCD eliminates this problem in two ways; it guides the authoring process to prevent the injection of errors and it performs all the required syntax checks to notify the user of any existing errors and how to correct them. Therefore, it can be argued that using SSUCD can only help the scalability of UC modeling as a functional requirements elicitation technique. The simplicity of SSUCD allows it be also used to model small systems as well. This is evident by the Online Hockey Store System case study presented in Section 3.2.7, which can be considered a relatively simple system.

Supportability

Perhaps the most important quality principle; if users do not have the adequate support to be able to use the language, then the language is useless. Users of any language, whether it is a modeling language or a programming language require tool support to help them produce models and programs. Tool support provided by SAREUCD is essential to the usability of SSUCD and the execution of the REUCD process. As mentioned previously, it is unrealistic to expect users of SSUCD to review its E-BNF syntax specifications in order to use it. SAREUCD was designed to perform most of the duties directly related to adopting SSUCD when describing UCs, whereby users need to be only concerned with writing the UCs instead of worrying about adhering to syntax rules.

Reliability

Producing reliable software is the principal objective of SSUCD. SSUCD ensures that the two major components of UC models are consistent. Consequently, understandability of UC models will significantly improve, which is vital to the success of a project that utilizes some form of a UC driven development process. Reliability and the cost of inconsistencies are discussed in great detail in Chapter 2.

Space economy

SSUCD's structural elements exist only within the UC descriptions. Visually, the presence of SSUCD within the textual descriptions is only in the form of a

handful of English keywords. Therefore, the size of the UC descriptions in large will visually remain the same whether or not they were structured with SSUCD. Viewing the descriptions with SAREUCD further enhances their readability since SAREUCD hides a large subset of SSUCD's keywords and structure to present the descriptions in a more natural form (see Figure 3-15).

3.2.7 Online Hockey Team Store System Case Study

The following case study is used to demonstrate how UC descriptions are presented in the SSUCD form and to demonstrate the application of the REUCD process. This case study will also illustrate the concepts, described in Sections 3.2.3 and 3.2.4, to systematically generate UC diagrams from UC descriptions using the REUCD process. The case study is about a simplified Online Hockey Team Store system. The presented system is simplified for clarity, yet complex enough for the purposes of demonstrating the SSUCD structure and the REUCD process.

The system allows customers to purchase tickets for upcoming hockey games. To buy a ticket, a customer needs to choose the game he/she would like to attend from the team's online calendar. The customer selects the desired section in an area where he/she would like their tickets to be along with the quantity of tickets requested. Upon retrieval of this information, the system will search the database for the requested tickets. If the tickets are available, the customer is prompted to either accept or reject the offered seats. If the customer accepts the offered seats, the customer is then directed to a billing page where the purchase

transaction can take place. Otherwise, if the tickets are not available, the customer is informed about the unavailability and then requested to submit another search for tickets. Occasionally, tickets for certain games in certain sections of the hockey arena may go on sale. Unlike regular priced tickets, a customer may purchase a maximum of six on sale tickets. The system also allows customers to purchase team merchandise such as hockey jerseys, sticks, and pucks. When choosing a merchandise item, the customer may provide customization requests for an extra cost. Available customization options depend on the type of item. For example, if the item was a hockey jersey, the customer may choose to have his/her name sewed on the jersey along with their favorite number. Meanwhile, if the item was a steel pen, the customer may have a name (or other words) engraved on the pen. To boost merchandise sales, a customer may enter a ticket number while purchasing merchandise for a chance to win a grand prize. A customer may purchase tickets and team merchandise using a credit card or a team hockey card. If the customer chooses to purchase using a credit card, an external credit card authorization system is utilized to verify the validity of the given credit card information. Meanwhile, if the customer chooses to purchase using a team hockey card, the customer is requested to enter a PIN. The system internally verifies the PIN with the associated hockey team card to approve the transaction. For any purchase, the customer is requested to enter billing information. The billing information is used for market survey purposes and delivery of tickets and team merchandise. Billing information would include the customer's name, phone number and address.

This simplified system contains seven UCs and two actors. The formal UC and actor descriptions are presented below. For illustrative purposes, the evolution of the UC diagram is shown below (see Figure 3-16→3-18).

Actors:

1) Actor: Customer
Brief Description: This actor may purchase hockey tickets at regular price or on sale. This actor may also purchase team merchandise. The actor will be requested to pay using a credit card or a team hockey card.

2) Actor: Credit Card Validation System
Brief Description: This actor ensures the validity of a given a credit card number and an expiry date.

Use Cases:

1) Use Case Name: Buy Tickets
Associated Actors: Customer
Description:

Preconditions:

At least one game and one seat is available

Brief Description: This UC is responsible for allowing customers to purchase as many tickets as they need in any section.

Basic Flow:

The system presents the different sections that exist in the arena and the price for a single seat in each section. The customer then enters information about the required tickets and submits order request. The system searches for the required tickets and prompts the Customer to accept or reject the offered seats. The customer accepts to purchase ticket and INCLUDE <Perform Transaction> to complete the transaction.

Alternative Flows:

- If tickets not available, the system notifies the Customer that the requested tickets are unavailable and the UC restarts.
- If the tickets were rejected, the system notifies the Customer that the cancellation has been confirmed and the UC restarts

Postconditions: If tickets are issued, these seats become unavailable for future Customers

2) Use Case Name:

Buy On Sale Tickets

SPECIALIZES Buy Tickets

Associated Actors:

Customer

Description:

Preconditions:

At least one game and one seat is available

Brief Description: This UC is responsible for allowing Customers to purchase a maximum of six on sale tickets.

Basic Flow:

The system presents the different sections that exist in the arena and the price for a single seat in each section. The Customer then indicates interest to purchase on sale tickets and submits an order to request tickets. The system retrieves the information about the required tickets and searches for them. The system prompts the Customer to accept or reject the offered seats. The Customer accepts to purchase tickets and INCLUDE <Perform Transaction> is performed to complete the transaction.

Alternative Flows:

- If the tickets not available, the system notifies the Customer that the requested tickets are unavailable and the UC restarts.
- If the Customer requested too many tickets, the system notifies the Customer that the requested number tickets exceed the maximum allowed of six and the UC restarts.
- If the tickets were rejected, the system notifies the Customer that the cancellation has been confirmed and the UC restarts

Postconditions: If tickets are issued, these seats become unavailable for future Customers

3) Use Case Name:

ABSTRACT

Perform Transaction

Brief Description: *This UC is responsible for allowing customers to pay for their selected items*

Preconditions:

At least one ticket is requested for purchase

Postconditions:

If tickets are issued, these seats become unavailable for future customers

If merchandise is sold, the merchandise database is updated

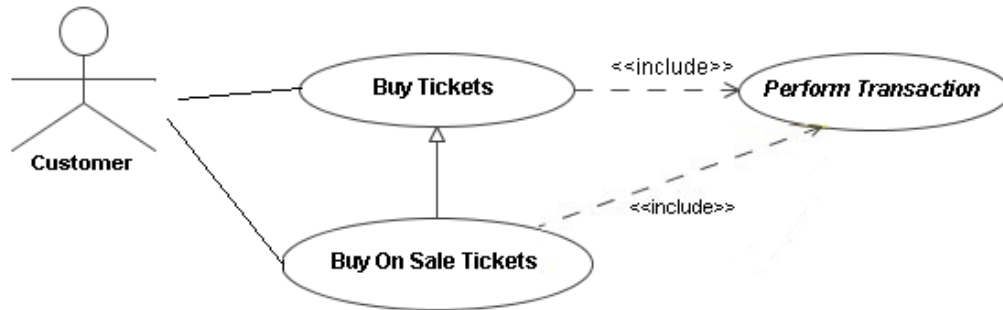


Figure 3-16: The UC diagram after three UC descriptions are read

4) Use Case Name:

Purchase With Credit Card

IMPLEMENTS Perform Transaction

Associated Actors:

Customer, Credit Card Validation System

Preconditions:

At least one item is requested for purchase

Brief Description: This UC is responsible for allowing Customers to pay for their selected items using a credit card

Basic Flow:

The system requests Customer to enter billing information. The Customer then enters the billing information and selects to pay using a credit card. Upon entering and submitting the credit card information, the given credit card is validated by the Credit Card Validation System and a receipt is printed.

Alternative Flows:

- If the credit card information is incorrect, the system notifies the Customer that the credit card information is incorrect and requests the Customer to enter the credit card information once again.

Postconditions: If tickets are issued, these seats become unavailable for future Customers

5) Use Case Name:

Purchase Using Hockey Team Card

IMPLEMENTS Perform Transaction

Associated Actors:

Customer

Preconditions:

At least one item is requested for purchase

Customer has a hockey team card with a set PIN

Brief Description: This UC is responsible for allowing Customers to pay for their selected items using a preauthorized payment plan setup on their hockey team card

Basic Flow:

The system requests Customer to enter the billing information. The Customer then enters the billing information and selects to pay using a hockey team card. The Customer then enters team hockey team card number and PIN. The system verifies the card number and PIN and prints a receipt.

Alternative Flows:

If the card information is invalid the system notifies the Customer that the hockey card information is incorrect and requests the Customer to enter the hockey card information once again.

Postconditions: If tickets are issued, these seats become unavailable for future Customers

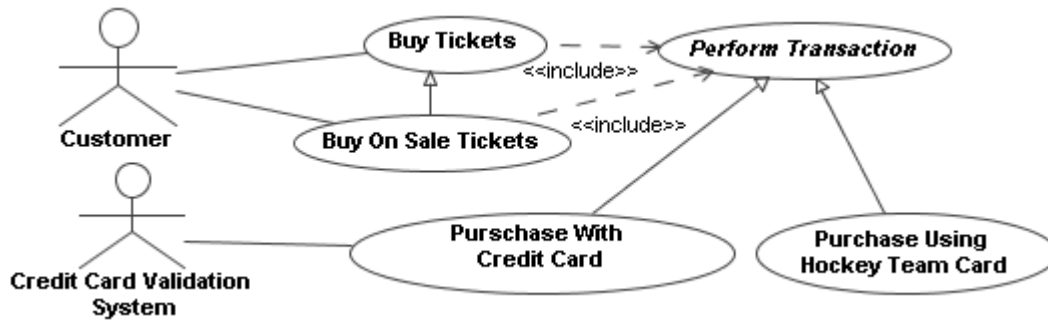


Figure 3-17: The UC diagram after five UC descriptions are read

<p>6) Use Case Name:</p> <p>Buy Team Merchandise</p>
<p>Associated Actors:</p> <p>Customer</p>
<p>Brief Description: This UC is responsible for allowing customers to buy team merchandise such as jersey, hockey sticks, mugs and other collectibles</p> <p>Basic Flow:</p> <p>The system displays catalogue with all team merchandise items. The Customer then selects the desired items to purchase, the desired quantity and any desired customization information. The INCLUDE <Perform Transaction> UC is performed to complete the transaction. The Customer finally enters a ticket number (if one is available) for a chance to win.</p>
<p>Extension Points:</p> <p>Grand Prize Giveaway</p>

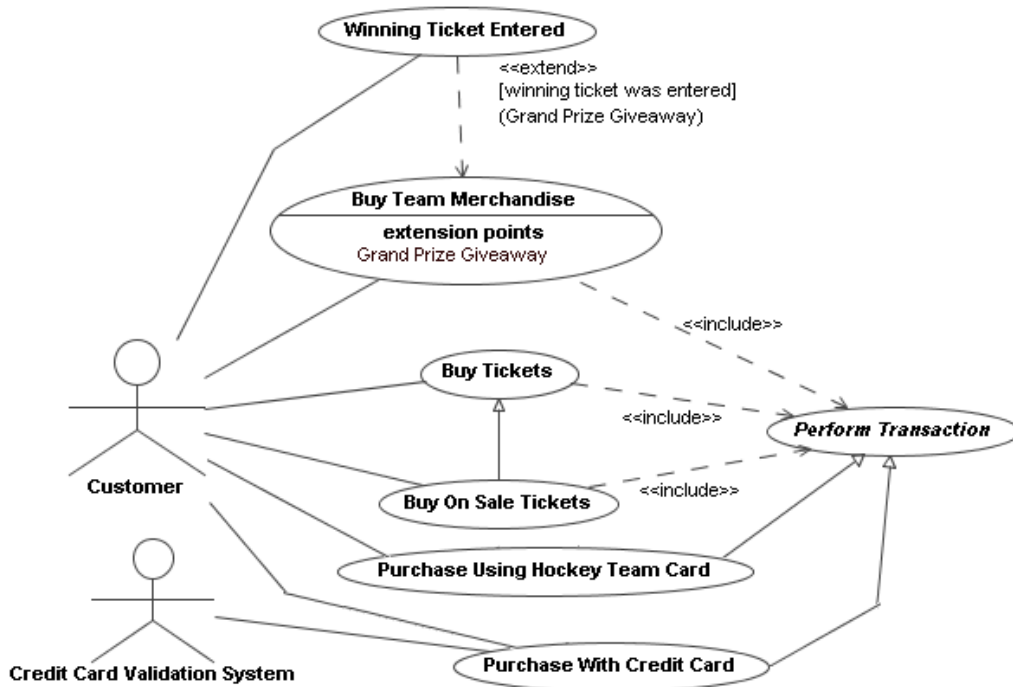


Figure 3-18: The UC diagram after all UCs and actors are read

<p>7) Use Case Name:</p> <p>Winning Ticket Entered</p>
<p>Associated Actors:</p> <p>Customer</p>
<p>Extended Use Cases:</p> <p>Base UC Name: Buy Team Merchandise</p> <p>Extension Point: Grand Prize Giveaway</p> <p>IF the winning ticket was entered</p>
<p>Brief Description: This UC is responsible for the situation where a winning ticket was entered.</p>
<p>Basic Flow:</p>

If the winning ticket was entered, the system notifies the Customer that they won the grand prize. The Customer enters phone number for a service representative to call

The final UC diagram (Figure 3-18) was systematically generated despite the descriptions containing very limited structure. Therefore, as discussed in Section 3.2.6, the ultimate goal was achieved by providing the minimal amount of structure without adding unnecessary complexities. The UC descriptions file and the XML file representing the diagram may be found at (STEAM 2009).

3.3 A Subject-Based Empirical Evaluation of SSUCD's Performance in Reducing Inconsistencies in Use Case Models

3.3.1 Introduction

While the SSUCD structure was specifically designed to be simpler to use than SUCD, this is no guarantee that it will possess suitable characteristics for its user group. This user group includes sub-groups with little or no technical background. Hence, the suitability or usability of these characteristics need to be evaluated via a subject-based empirical study, which is presented in this section. The controlled experiment described in this Section took place at the University of Alberta, Canada. This experiment follows the well-known experimentation process proposed by Wohlin et al. (Wohlin et al. 2000). According to the template proposed in (Wohlin et al. 2000), the following subsections describe the experiments: definition, context, hypotheses formulation, subject selection, design, instrumentation and measurement techniques, and validity evaluation, respectively.

3.3.2 Experimental Planning

3.3.2.1 Experiment Definition

The main research question posed by this experiment is whether the usage of the SSUCD structure to author UC descriptions results in developing UC models with higher consistency levels in comparison to using traditional UNL. The secondary

research question is whether the usage of the SSUCD structure changes the other quality attributes in comparison to using UNL. It is possible that even if SSUCD improves consistency that it reduces the overall quality of UC models by negatively impacting other quality attributes. The usage of UNL is used in this experiment as the control situation since it is the most commonly used form in industrial settings.

Since the issue of inconsistencies between UC diagrams and descriptions is the only issue that is tackled *directly* by SSUCD, we first assess whether using SSUCD will indeed improve the consistency between these components. In addition, we assess its impact upon other quality attributes affecting UC models.

If SSUCD's structural elements were excluded from UC descriptions, the resulting artifact would simply be UC descriptions in UNL form. Therefore, the only independent variable is the use of the SSUCD structure; and hence two treatments exist, SSUCD and UNL (non-SSUCD). This experiment also has five dependent variables upon which the treatments are compared: inconsistency mistakes (*I*), content completeness (*C*), false facts and information (*I'*), non-analytical facts and information (*NA*) and other elements that reduce understandability (*U*).

3.3.2.2 Experiment Context

This experiment involved Electrical/Computer/Software Engineering graduate students. It was conducted as part of a voluntary mini-course, which did not contribute towards the subjects' degree requirements. The course was divided into

two major components. The first was a series of five one-hour lectures to introduce UC modeling concepts and techniques, and to allow them to practice these techniques using a number of examples. The second component of the course was two lab exercises that constituted the experimental tasks. The students were not informed about the hypotheses under investigation.

3.3.2.3 Hypotheses Formulation

Five hypotheses were produced to account for the potential effects of using SSUCD to develop UC descriptions (see Table 3-4). The alternative hypothesis (H_a) for the consistency variable (I) indicates that it was expected that there would be less instances of inconsistency when using SSUCD. Inconsistency is the only one-tailed hypothesis as SSUCD was intentionally designed to tackle this issue, and thus it is expected that UC models constructed with SSUCD will contain lower counts of inconsistencies. However, since SSUCD was not designed to directly improve the other attributes, the remaining hypotheses are considered as non-directional hypotheses (see Table 3-4).

Table 3-4: Five dependent variables and their corresponding hypotheses

Dependent Variable	Null Hypothesis (H_o)	Alternative Hypothesis (H_a)
Inconsistency	$(H_o1): I (SSUCD) \leq I (UNL)$	$(H_a1): I (SSUCD) > I (UNL)$
Completeness	$(H_o2): C (SSUCD) = C (UNL)$	$(H_a2): C (SSUCD) \neq C (UNL)$
Fault-Free	$(H_o3): F' (SSUCD) = F' (UNL)$	$(H_a3): F' (SSUCD) \neq F' (UNL)$
Non-Analytical	$(H_o4): NA (SSUCD) = NA (UNL)$	$(H_a4): NA (SSUCD) \neq NA (UNL)$
Understandability	$(H_o5): U (SSUCD) = U (UNL)$	$(H_a5): U (SSUCD) \neq U (UNL)$

3.3.2.4 Subject Selection

All graduate level students with an undergraduate background relative to software development were invited to participate. In total, 34 students voluntarily agreed to participate. It is important to note that none of the students participated in the original study (see Section 2.5.4). Informal interviews with the subjects have indicated that none of them had previous exposure to UC modeling. It is beneficial that the subjects did not have UC modeling experience as there would have been a tendency to ignore the techniques and concepts taught in the lectures; and instead, apply the techniques that they were more familiar with from their experience. However, it must be noted that the fact that the subjects lack any UC modeling experience may raise concern with respect to external validity. This issue is discussed in more detail in Section 3.3.6.4.

It is not possible to determine and compare the subjects' relative educational experience, as the subjects have pursued their undergraduate studies in various universities situated in various countries, and undergraduate programs vastly differ. However, the fact that they are all graduate students is indicative of their general abilities.

3.3.2.5 Experimental Design and Tasks

This experiment required all subjects to consider two distinct systems, an Airline Ticketing system (Overgaard et al. 2005) and a Banking system (Gomaa 2000). The ideal solutions for both of these systems are presented in their respective sources. It was critical to use externally developed systems to eliminate biases,

since SSUCD is developed through this research work. The subjects were randomly assigned into two groups (A and B) of 17 subjects each. For each part of this experiment, the subjects were given the Requirements Documents (RDs) of the respective systems and were asked to develop the entire UC model. To mitigate the effect of individual and group abilities, a 2×2 partial factorial design with repeated measure is utilized (see Table 3-5 for details).

Table 3-5: Experimental design

	Group A		Group B	
Week 1	Introduction to UC modeling - 2 lectures (approx. 2 hours total)			
Week 2	UC modeling practice using UNL and SSUCD – 3 lectures (approx. 3 hours)			
Week 3	UNL	Develop Airline Ticketing system	SSUCD	Develop Airline Ticketing system
Week 4	SSUCD	Develop Banking system	UNL	Develop Banking system

Table 3-6: Details of the two systems used in this experiment

	Airline Ticketing System	Banking System
# of UCs	3	4
# of actors	1	1
# of relationships and their types	2 associations 1 extend 1 include	3 associations 3 include
# of functional facts	11	21

Table 3-6 shows the structural and content details of both systems; note that the Banking system was slightly modified from its original version so that its “difficulty level” would be closer to the Airline Ticketing system. The Banking

system still contains one more UC than the Airline Ticketing system, and requires more information and functional facts to be stated in the corresponding UC descriptions. Meanwhile, the relationships contained in the Airline Ticketing system seem to be more “complex” to identify than those of the Banking system, even though the Banking system contains two more relationships. Relationships from the Banking system are more repetitive and more explicitly stated than the relationships from the Airline Ticketing system. The Airline Ticketing system contains an *extend* relationship which the Banking system lacks.

3.3.2.6 Time Allocation

As the exercises were relatively small, subjects were expected to finish them in approximately 1 hour (± 15 minutes). Subjects did not have to face any timing pressures since both sessions were 3 hours long. All subjects finished their tasks and no great time differences were witnessed.

3.3.2.7 Instrumentation

Tool support is available for SSUCD using SAREUCD (El-Attar et al. 2006a). SAREUCD provides a GUI interface that allows its users to focus entirely on describing their UCs without accounting for SSUCD’s syntactical requirements. Tool support is also available to describe UCs in UNL, such as [Analyst Pro 2008, Optimal Trace 2008, Use Case Studio 2008, TopTeam 2008). The Subjects were not allowed to use any tools that are specifically designed to support the authoring of UCs, whether in SSUCD or UNL. The rationale behind this decision is to

compare the effectiveness of using SSUCD and UNL as a means to write UC descriptions, and eliminate any biases that might be introduced by tool support.

3.3.2.8 Analysis Procedure

Under the assumption that all deficiencies have an equal unit weighting, the quantitative data presented in this Section can be considered as discrete count data. Unfortunately, we have no causal explanation as to the nature of the distribution that the data points are sampled from. Using a statistical exploratory analysis approach, we examined the various data sets for their compliance to normality assumptions using the Shapiro-Wilk test (Shapiro et al. 1972). This test was selected as it tends to be more powerful than other common “normality” tests (such as Anderson-Darling and Kolmogorov-Smirnov) and does not require that the mean or variance of the hypothesized normal distribution to be specified in advance. For more details on this technique see (Shapiro et al. 1972). This test indicated that several of our datasets are non-normal. Hence, we will adopt a conservative approach in all of our quantitative analysis and consider all datasets as being sampled from non-parametric distributions.

3.3.2.9 Scoring and Measurement

This section presents examples of the defects listed in Chapter 2 as well as how they were scored. Please note that the entire set of defect examples is very extensive and would require a great deal of space to present. Table 3-7 presents a

large cross section of defect examples. Table 3-8 presents the scoring strategy for each quality attribute.

Table 3-7: defect examples

Category	Examples
Inconsistency	<p>1. Diagram vs. Diagram Inconsistencies:</p> <ul style="list-style-type: none"> ▪ A UC, actor or relationship that is depicted in the UC diagram but not described or mentioned in the UC descriptions.
	<p>2. Diagram vs. Descriptions Inconsistencies:</p> <ul style="list-style-type: none"> ▪ A relationship originating from UC-A (or Actor-A) towards UC-B (or Actor-B) in one diagram, while another diagrams show that the relationship originates from UC-A (or Actor-A) towards UC-B (or Actor-B). ▪ A relationship that is present between two elements in one diagram but not present between the same elements in another diagram. ▪ Two elements sharing a particular type of relationship between them in one diagram but share another type of relationship between them in another diagram.
	<p>3. Diagram vs. Descriptions Inconsistencies:</p> <ul style="list-style-type: none"> ▪ Two contradicting statements.
Completeness	<p>1. A missing statement of necessary facts.</p> <ul style="list-style-type: none"> ▪ Assuming a UC called “Deposit Funds” that describes

	<p>the functionality regarding the depository of funds in an ATM machine. If that UC fails state that the customer must have at least one account at the bank that owns the ATM machine in order to be able to deposit the funds, then this counts as a defect.</p>
	<p>2. An exclusion of necessary activities that take place in a UC.</p> <ul style="list-style-type: none"> ▪ For the “Deposit Funds” UC, assuming the ATM requires the customer to indicate the type of deposit that they will be making (cash or check), if the UC fails to state that the customer should be prompted for this information, then this counts as defect.
	<p>3. A missing statement of a dependency between elements.</p> <ul style="list-style-type: none"> ▪ For the “Deposit Funds” UC, assuming that the UC depends on another UC that validates the customer’s PIN through an <i>inclusion</i> relationship, if the UC fails to state this dependency, then this counts as defect.
	<p>4. A missing actor description.</p> <ul style="list-style-type: none"> ▪ All actors must be described. If an actor description is missing, then this counts as a defect.
	<p>5. A missing diagrammatic element or description.</p> <ul style="list-style-type: none"> ▪ If an actor, UC or relationship is supposed to be depicted in the UC diagram but is not, then this counts as a defect.

Fault-Free	<p>1. An incorrect fact or information about an actor.</p> <ul style="list-style-type: none"> ▪ Assuming an actor representing a customer of a lottery system and the description of the actor states that the actor should be at least 18 years of age, when the actor should actually be at least 21 years of age.
	<p>2. An incorrect activity that occurs in a UC.</p> <ul style="list-style-type: none"> ▪ After an email is downloaded, it is stated that the entire set of emails in the inbox are scanned for viruses, when only the downloaded email should be scanned.
	<p>3. An incorrect fact stated in a UC.</p> <ul style="list-style-type: none"> ▪ The seating capacity of a stadium is stated as 90,000, when it should be 95,000.
	<p>4. An incorrect dependency (or dependencies) between elements.</p> <ul style="list-style-type: none"> ▪ A UC responsible for allowing visitors to browse the online Library catalogue indicates a dependency on another UC that allows visitors to login as members, when visitors do not need to login since anyone (not only members) can browse the Library catalogue.
Non-Analytical	<p>1. The description of any GUI elements.</p> <ul style="list-style-type: none"> ▪ The use of drop-down menus on a certain page or the use of certain colors in the design of a website.

	<p>2. The statement of using a certain algorithm or procedure to perform a certain activity.</p> <ul style="list-style-type: none"> ▪ Determining the optimum route between two locations <hr/> <p>3. Any presumption regarding how an actor, which is an external entity, will perform its own internal tasks.</p> <ul style="list-style-type: none"> ▪ For example, how an external Credit Card Verification system actually verifies a credit card.
<p>Understandability</p>	<p>1. Unnecessarily repeated information or facts about actors or in UCs.</p> <ul style="list-style-type: none"> ▪ For the “Deposit Funds” UC, stating multiples times that the customer must have his PIN validated. Validating the PIN should occur once at the start of the UC. Stating this action multiple might lead to confusion as to whether the system should validate the PIN multiple times while that UC is being performed. <hr/> <p>2. Very small UCs: these are UCs that do not contain enough behavior to deliver a meaningful service to an actor and hence need to be “linked” together or “combined” in order to provide such a service.</p> <ul style="list-style-type: none"> ▪ Assuming in a telephony system, a set of UCs named “Enter Destination Number”, “Get Connection” and “Ring Destination Phone”. These UCs <i>collectively</i> are responsible for carrying out a phone call. These UCs

	<p>should be combined into a single UC called “Place Phone Call”.</p>
	<p>3. Very big UCs: these are UCs that offer more than one meaningful service; each service should ideally be delivered by a separate UC.</p> <ul style="list-style-type: none">▪ Assuming in an ATM system a UC is responsible for withdrawing, depositing and transferring funds. This UC offers more one service and should be split into three UCs named “Withdraw Funds”, “Deposit Funds” and “Transfer Funds”.
	<p>4. Ambiguous information stated in an actor or UC description.</p> <ul style="list-style-type: none">▪ For a cruise control system, a UC stating that the cruise control will be deactivated once vehicle to traveling at a <i>very low speed</i>. It is ambiguous as to what is the exact threshold to automatically deactivate the cruise control system.

Table 3-8: Scoring Strategy

Category	Scoring Strategy
Inconsistency	<p>All “Inconsistency” defects are scored similarly (as a discrete count of ‘1’) regardless of their type. “Inconsistency” is scored as the sum of unique inconsistencies committed in the UC model. For example, for inconsistencies of type (c), if a fact was stated as “A” three times while stated as “B” elsewhere, then this counts as one “Inconsistency” defect since the inconsistency is the same ($A \neq B$). Meanwhile, if a fact was stated as “A”, “B” and “C” in three different locations of the descriptions, then this counts as three “Inconsistency” defects since $A \neq B$, $A \neq C$ and $B \neq C$.</p>
Completeness	<p>All “Completeness” defects are scored similarly (as a discrete count ‘1’) regardless of their type. If a fact-A was required to be stated at two unique locations in the UC descriptions but was missed both times, this is scored as two defects. Note that scoring of the textual descriptions in the “Completeness” category is based on the facts, relationships and activities that are supposed to be present, not the existence of the actual UC description. For example, if fact-A and fact-B are supposed to be described in UC-A but were instead described under a similar UC, called UC-B, then no</p>

	<p>“Completeness” defects are scored, instead another defect is scored under a different category depending on the actual situation.</p>
Fault-Free	<p>All “Fault-Free” defects are scored similarly (as a discrete count ‘1’) regardless of their type. “Fault-Free” is scored as the sum of unique “Fault-Free” defects committed in the UC model. For example, if a UC states fact-A twice and fact-A is incorrect then this counts as a single “Fault-Free” defect. If a UC states a correct fact-A and an incorrect fact-B, then this counts as a single “Fault-Free” free defect as well as an “Inconsistency” defect.</p>
Non-Analytical	<p>All “Non-Analytical” defects are scored similarly (as a discrete count ‘1’) regardless of their type. “Non-Analytical” is scored as the sum of unique “Non-Analytical” defects committed in the UC model. For example, if it was mentioned multiple times that volume control will be handled as two “Up” and “Down” buttons rather than a drag-able lever, then this counts as one “Non-Analytical” defect. If UC-A states that it will be presented as a menu item in a menu, and UC-B states that will be presented as another menu item, then this counts as two “Non-Analytical” defects since these are two distinct GUI decisions.</p>
Understandability	<p>All “Understandability” defects are scored similarly (as a</p>

discrete count ‘1’) regardless of their type. In the case of unnecessary repeated information (type (1)), defects are scored based on the number of repetitions that a certain fact or information was repeated. For example, if a fact-A was unnecessarily repeated five times, then this counts as a five “Understandability” defects. For the remaining types of “Understandability” defects, scoring based on the sum of unique “Understandability” defects committed in the UC model. For example, if ambiguous information such as *very low speed* was stated multiple times, then this counts as a single “Understandability” defect.

Appendix D shows two partial UC models developed by subjects during the experiment and their evaluation with respect to “Inconsistency” only. One partial UC model is of a Banking system developed with UNL, while the other of an Airline Ticketing system developed with SSUCD. The respective UC models were not shown in their entirety due to space limitations. Each partial UC model consists of a UC diagram¹ and three UC descriptions. A detailed walkthrough of how “Inconsistency” defects were detected and scored is also presented. Scoring of other quality attributes are not shown due to space limitations as they would require the presentation of the entire UC model and very lengthy walkthroughs of how their defects were detected and scored.

¹ The UC diagrams in appendix D were redrawn using a UML modeling tool for clarity and presentation purposes.

3.3.3 Analysis and Interpretation

For each of the non-parametric variables, we present a descriptive summary in terms of a notched box and whiskers plot (see Figure 3-19). The notched box shows the median, lower and upper quartiles, and confidence interval around the median. The vertical lines show the non-parametric 95% percentile range. The upper and lower horizontal lines show the upper and lower quartiles respectively. The middle horizontal line shows the median. Tilted lines stemming from the median show the confidence interval around the median. The dotted-line connects the nearest observations within 1.5 IQRs (inter-quartile ranges) of the lower and upper quartiles. The crosses (+) and circles (o) indicate possible outliers – observations more than 1.5 IQRs (near outliers) and 3.0 IQRs (far outliers) from the quartiles.

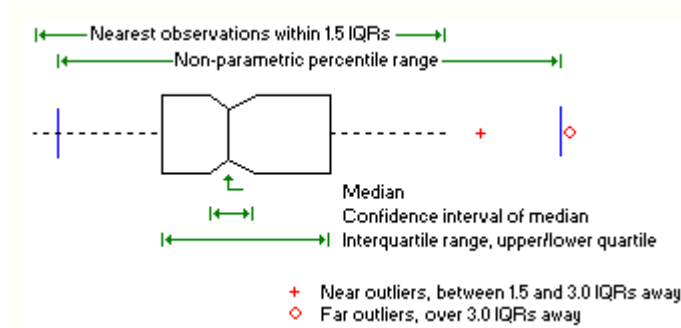


Figure 3-19: Illustration of the box and whiskers plot's diagrammatic notation

In addition, we test for differences between the medians of related samples using the Mann-Whitney U statistic (of the 1st sample) as described in (Siegel et al. 1988). The probability provided should in general be considered as an underestimation due to the presence of a number of ties within the datasets which prevents us from using an exact test. The confidence intervals around the

difference between medians are also computed using the well-known Hodges-Lehmann method (Lehmann 1998); all confidence intervals are given at the standard 95% level.

Finally, for major results from statistical significant testing, we will provide an estimate of the size of the difference between the two groups by estimating the associated effect size. Cliff's delta (Cliff 1993, 1996, 1996b) is used as a non-parametric effect size measure. Kromrey *et al.* (Kromrey et al. 1998, 2005) and Hess *et al.* (Hess et al. 2005) have empirically demonstrated that Cliff's delta is superior to Cohen's *d* and Hedges' *g* when the data is non-normal or possesses variance heterogeneity. Cliff's delta examines the probability that individual observations within one group are likely to be greater than the observations in the other group:

$$\Delta = \Pr(x_{i1} > x_{j2}) - \Pr(x_{i1} < x_{j2})$$

Where x_{i1} is a member of population one and x_{j2} is a member of population two.

Cliff's $\hat{\delta}$ has two alternatives in terms of estimating its associated variance. In this article, we will utilize the "consistent" estimate of the variance² as it allows the construction of the associated asymmetric confidence intervals, at 95%, around the sample value of $\hat{\delta}$. However, it should be noted that Cliff (Cliff 1996b) states that this approach produces highly conservative confidence intervals, especially with low numbers of subjects, and advises against hypothesis testing based upon these estimates.

² Kromrey and Hogarty [38] empirically demonstrated that the choice of variance procedure is relatively unimportant across a wide range of circumstances.

Unlike Cohen's d , Cliff's $\hat{\delta}$ has no universally accepted linguistic interpretations and hence we will in general refrain from directly inferring linguistic size statements from it. Although clearly, values approaching the extremes of the effect size range can be considered "large", to borrow terminology from Cohen's d . Despite Cliff's recommendation, we will utilize the effect size measure to compute exploratory significance hypothesis testing. The risk that the tests are in fact measuring beyond the 95% level is not considered to be too important in these circumstances. Due to a lack of casual theory, we were unable to propose directional hypothesis for the majority of the hypotheses examined. This limitation can now be resolved. For two populations, if zero is included within the confidence interval of Cliff's delta then the populations are considered equal; if the confidence interval only includes negative numbers then UNL > SSUCD (favoring UNL subjects); if it only includes positive numbers then SSUCD > UNL (favoring SSUCD subjects).

3.3.3.1 Performed Analysis

The analysis performed investigates the effects of the treatment variables and experimental artifacts in isolation: in Section 3.3.3.2→3.3.3.5 the effect of using SSUCD vs. UNL on each system separately with respect to various quality attributes; in Section 3.3.3.6, the results obtained for the Airline Ticketing System vs. Banking System with respect to each quality attribute individually by both groups; and in Section 3.3.3.7, the performance of Group A vs. Group B with respect to each quality attribute individually, using both systems.

3.3.3.2 SSUCD vs. UNL – Inconsistencies

As stated in Chapter 2, while three types of inconsistencies are considered; neither system provides the subjects with the opportunity to commit inconsistency mistakes of types (b) and (c). Figure 3-20 shows the results for the combined count of inconsistencies for the Airline Ticketing system. Table 3-9 shows that SSUCD subjects have statistically significant (lower) inconsistency values than UNL subjects. This indicates that the embodiment of SSUCD structural constructs in the UC descriptions explicitly prompts subjects to consider and crosscheck diagrammatic and descriptive elements for consistency. This statistical significance is further confirmed as the confidence interval around $\hat{\delta}$ includes only positive values (see Table 3-10).

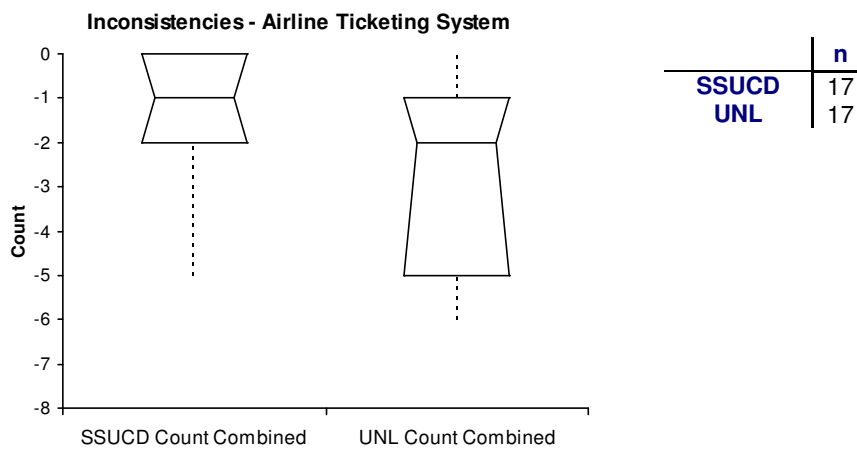


Figure 3-20: Inconsistencies - Airline Ticketing System

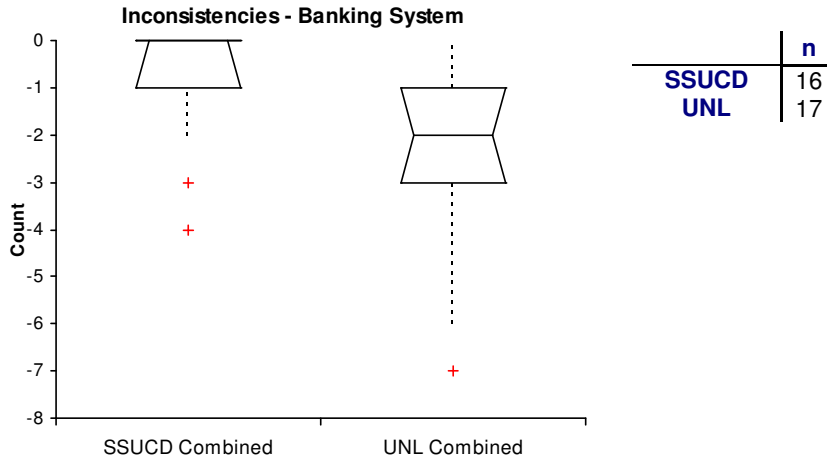


Figure 3-21: Inconsistencies - Banking System

Figure 3-21 shows the results of the combined inconsistency count from the Banking system. Once again, the results show that SSUCD subjects, statistically significantly, commit less inconsistency mistakes than their UNL counterparts (Table 3-9); this picture is re-enforced by the “large” positive $\hat{\delta}$ value and a confidence interval that only includes positive values (Table 3-10).

Table 3-9: Mann-Whitney test for the ‘Inconsistencies’ results

Alternative Hypothesis - (H_a): I (SSUCD) > I (UNL):				
System	Technique	Rank sum	Mean rank	U
Airline Ticketing System	SSUCD	362.5	21.32	79.5
	UNL	232.5	13.68	209.5
Banking System	SSUCD	337.5	21.09	70.5
	UNL	223.5	13.15	201.5
System	Difference between medians	95.2% CI	Mann-Whitney U statistic	1-tailed p
Airline Ticketing System	1.0	0.0 to $+\infty$	79.5	0.010
Banking System	1.0	0.0 to $+\infty$	70.5	0.010

Table 3-10: Cliff’s delta for the ‘Inconsistencies’ results

System	Cliff’s delta ($\hat{\delta}$)	Variance	Confidence Interval around delta ($\hat{\delta}$)	
			maximum	minimum
Airline Ticketing System	0.450	0.030	0.673	0.112
Banking System	0.764	0.028	0.783	0.435

3.3.3.3 SSUCD vs. UNL – Completeness

Figure 3-22 shows analysis with respect to ‘Completeness’ with the Airline Ticketing system. Remember, that the completeness of a UC model cannot be determined by the UC diagram, but only by the contents of the UC descriptions. The UC descriptions need to collectively state certain facts. The Airline Ticketing system collectively contains 12 distinct facts. These can be divided into subcategories according to functionality supplied by the system. Individually the subcategories contained insufficient sample sizes to allow “safe” statistical analysis to be performed. Considering the combined completeness count produced for the Airline system, Table 3-11 shows that there was a statistical significance between the performance of the SSUCD and UNL subjects. The positive range of the confidence interval around $\hat{\delta}$ (Table 3-12) indicates that SSUCD subjects have an overall higher completeness count than UNL subjects. As SSUCD explicitly promotes consistency, subjects are likely to be more inclined to reconsider the identified relationships by re-examining the Requirements Document. This in turn will incline subjects to consider and outline the contents of their UC descriptions before authoring them. Subjects have later confirmed during informal interviews that they believe that SSUCD did provoke them to undertake a more rigorous approach towards authoring their UCs.

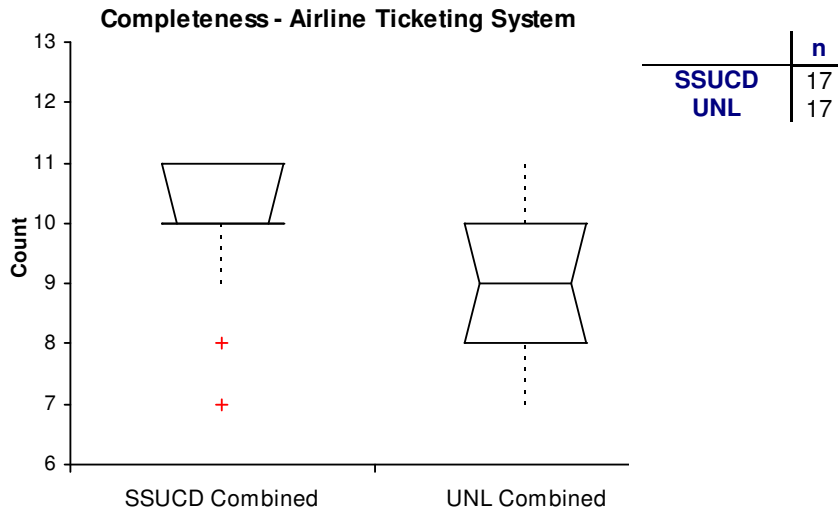


Figure 3-22: Completeness - Airline Ticketing System

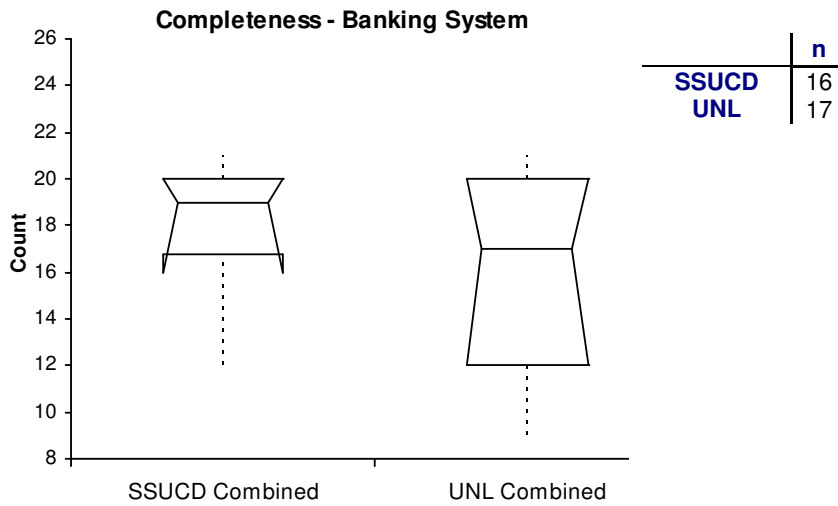


Figure 3-23: Completeness - Banking System

Figure 3-23 shows the analysis with respect to ‘Completeness’ with the Banking system. Tables 3-11 and 3-12 show that no statistically significant difference was observed within the data from the Banking System.

Table 3-11: Mann-Whitney test for the ‘Completeness’ results

Alternative Hypothesis - (H_aI): C (SSUCD) \neq C (UNL):				
System	Technique	Rank sum	Mean rank	U
Airline Ticketing System	SSUCD	364.0	21.41	78.0
	UNL	231.0	13.59	211.0
Banking System	SSUCD	305.0	19.06	103.0
	UNL	256.0	15.06	169.0
System	Difference between medians	95.2% CI	Mann-Whitney U statistic	1-tailed p
Airline Ticketing System	1.0	0.0 to 2.0	78	0.018
Banking System	1.0	-1.0 to 4.0	103	0.231

Table 3-12: Cliff’s delta for the ‘Completeness’ results

System	Cliff’s delta ($\hat{\delta}$)	Variance	Confidence Interval around delta ($\hat{\delta}$)	
			maximum	minimum
Airline Ticketing System	0.460	0.030	0.680	0.122
Banking System	0.243	0.038	0.567	-0.147

3.3.3.4 SSUCD vs. UNL – Understandability

Figure 3-24 shows the results for ‘Understandability’ with respect to the Airline Ticketing System. The results show a statistically significant difference between the performance of SSUCD and UNL subjects (Table 3-13). The positive range of the confidence interval around $\hat{\delta}$ (Table 3-14) indicates that SSUCD subjects have performed better than UNL subjects. This might be attributable to the fact that SSUCD prompts subjects to consider and plan the contents of their UCs before they start authoring them. No statistical significance was observed with respect to the Banking System (see Figure 3-25 and Table 3-13).

Table 3-13: Mann-Whitney test for the ‘Understandability’ results

Alternative Hypothesis - (H_a): U (SSUCD) \neq U (UNL):				
System	Technique	Rank sum	Mean rank	U
Airline Ticketing System	SSUCD	385.5	22.68	56.5
	UNL	209.5	12.32	232.5
Banking System	SSUCD	243.0	15.19	165.0
	UNL	385.5	22.68	56.5

System	Difference between medians	95.2% CI	Mann-Whitney U statistic	1-tailed p
Airline Ticketing System	2.0	1.0 to 3.0	56.5	<0.01
Banking System	-1.0	-2.0 to 1.0	165	0.289

3.3.3.5 SSUCD vs. UNL - Fault-Free and Non-Analytical Information

The results for the ‘Fault-Free’ quality attribute with the Airline Ticketing System and the Banking System are shown individually in Figures 3-26 and 3-27, respectively. Results for the ‘Non-Analytical’ quality attribute are shown in Figures 3-28 and 3-29, respectively. No statistical significant differences were observed for either quality attribute with either system (Tables 3-15 – 3-17).

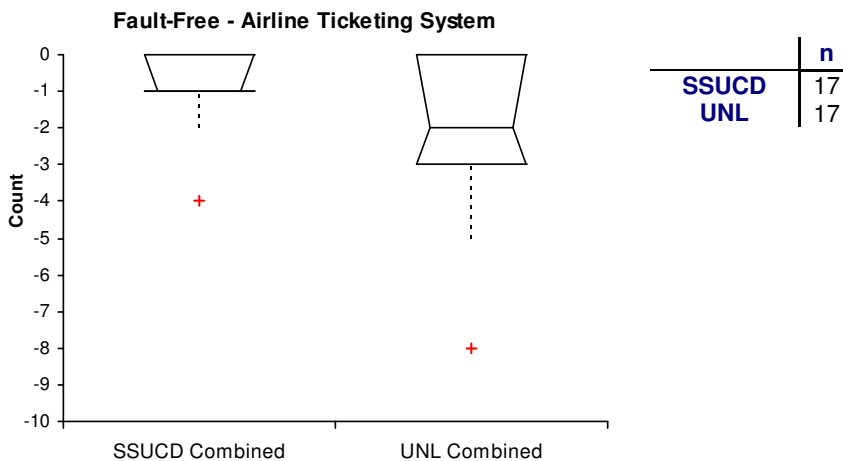


Figure 3-26: Fault-Free - Airline Ticketing System

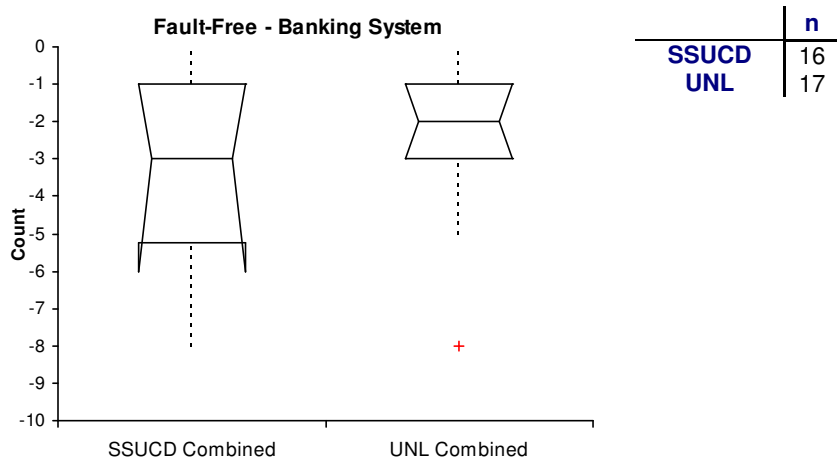


Figure 3-27: Fault-Free - Banking System

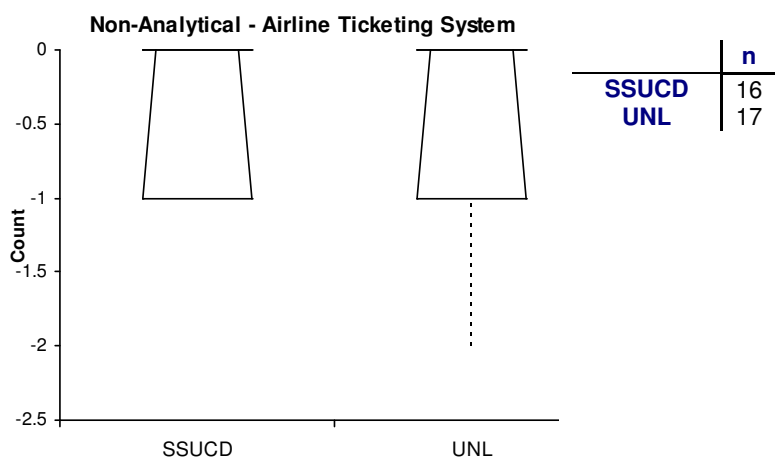


Figure 3-28: Non-Analytical - Airline Ticketing System

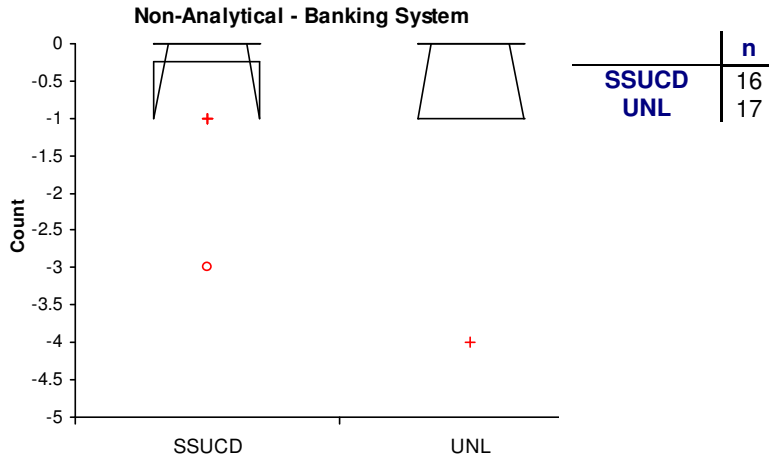


Figure 3-29: Non-Analytical - Banking System

Table 3-15: Mann-Whitney test for the ‘Fault-Free’ results

<i>Alternative Hypothesis - (H_a): P (SSUCD) \neq P (UNL):</i>				
System	Technique	Rank sum	Mean rank	U
Airline Ticketing System	SSUCD	330.5	19.44	111.5
	UNL	264.5	15.56	177.5
Banking System	SSUCD	297.5	18.59	110.5
	UNL	263.5	15.50	161.5
System	Difference between medians	95.2% CI	Mann-Whitney U statistic	1-tailed p
Airline Ticketing System	1.0	0.0 to 2.0	111.5	0.236
Banking System	1.0	-1.0 to 2.0	110.5	0.348

Table 3-16: Mann-Whitney test for the ‘Non-Analytical’ results

<i>Alternative Hypothesis - (H_a): C (SSUCD) \neq C (UNL):</i>				
System	Technique	Rank sum	Mean rank	U
Airline Ticketing System	SSUCD	330.5	19.44	111.5
	UNL	264.5	15.56	177.5
Banking System	SSUCD	258.5	16.16	149.5
	UNL	302.5	17.79	122.5
System	Difference between medians	95.2% CI	Mann-Whitney U statistic	1-tailed p
Airline Ticketing System	0.0	0.0 to 1.0	111.5	0.186
Banking System	0.0	0.0 to 0	149.5	0.546

Table 3-17: Cliff’s delta for the ‘Fault-Free’ and ‘Non-Analytical’ results

Attribute	System	Cliff’s delta ($\hat{\delta}$)	Variance	Confidence Interval around delta ($\hat{\delta}$)	
				maximum	minimum
Fault-Free	Airline	0.242	0.040	0.573	-0.157
	Banking	-0.188	0.040	0.202	-0.526
Non-Analytical	Airline	0.228	0.029	0.520	-0.111
	Banking	0.099	0.026	0.391	-0.211

3.3.3.6 Airline Ticketing System vs. Banking System

In this section, we investigate the relationship between the two documents; or to be more precise, either the relationship between the documents or the possibility of a learning effect. Unfortunately, due to the limited partial factorial nature of the experimental design, we are unable to distinguish between these two factors. However, for simplicity, this section will only refer to the evaluation with respect to the documents. While, it is believed that many formulations are possible, we will simply compare the rescaled (or weighted) performance of the subjects with respect to only the documents. This rescaling is implemented by subtracting the minimum score achieved by any subject, regardless of technique, from the current score under consideration and dividing this result by the maximum score achieved by any subject, regardless of technique. Note that the maximum score achieved by a subject is similar to the theoretical maximum score for “Completeness”; and that the other quality attributes do not possess a theoretical maximum score.

Table 3-18: Descriptive statistics of the results

Normalized	System	n	Median	IQR	95% CI of Median
Inconsistencies	Airline	34	0.833	0.375	0.667 to 0.833
	Banking	33	0.857	0.429	0.714 to 1.000
Completeness	Airline	34	0.909	0.250	0.818 to 1.000
	Banking	33	0.857	0.238	0.762 to 0.905
Fault-Free	Airline	34	0.875	0.250	0.750 to 1.000
	Banking	33	0.625	0.500	0.625 to 0.875
Non-Analytical	Airline	34	1.000	0.500	0.500 to 1.000
	Banking	33	1.000	0.250	1.000 to 1.000
Understandability	Airline	34	0.667	0.333	0.500 to 0.833
	Banking	33	0.375	0.375	0.250 to 0.500

Table 3-19: Mann-Whitney test for all quality attributes

Normalized	System	Rank sum	Mean rank	U
Inconsistencies	Airline	1005.0	29.56	712.0
	Banking	1273.0	38.58	410.0
Completeness	Airline	1291.0	37.97	426.0
	Banking	987.0	29.91	696.0
Fault-Free	Airline	1368.0	40.24	349.0
	Banking	910.0	27.58	773.0
Non-Analytical	Airline	1068.0	31.41	649.0
	Banking	1210.0	36.67	473.0
Understandability	Airline	1375.0	40.44	342.0
	Banking	903.0	27.36	780.0
Normalized	Difference between medians	95.2% CI	Mann-Whitney U statistic	2-tailed p
Inconsistencies	-0.071	-0.19 to 0	712	0.052
Completeness	0.048	0.0 to 0.100	426	0.088
Fault-Free	0.125	0.0 to 0.250	349	0.007
Non-Analytical	0.000	-0.25 to 0	649	0.191
Understandability	0.250	0.083 to 0.333	342	0.006

Table 3-20: Cliff's delta for all quality attributes

Normalized	Cliff's delta ($\hat{\delta}$)	Variance	Confidence Interval around delta ($\hat{\delta}$)	
			maximum	minimum
Inconsistencies	0.269	0.020	0.520	-0.024
Completeness	-0.241	0.022	0.058	-0.499
Fault-Free	-0.378	0.019	-0.090	-0.608
Non-Analytical	0.157	0.016	0.389	-0.095
Understandability	-0.390	0.021	-0.086	-0.627

Table 3-18 shows the results of the subjects' performances for both systems with respect to each quality attribute. A statistically significant difference was observed for the 'Fault-Free' and 'Understandability' attributes (Table 3-19). Table 3-20 further indicates that subjects performed better with the Airline Ticketing system with respect to those two quality attributes. Further examination of the subjects' performances was conducted to shed more "light" into this situation. The 'Fault-Free' category consists of two subcategories: 'Incorrect Facts' and 'Incorrect Diagrammatic Elements'. It was revealed that the Banking system subjects committed more mistakes in both subcategories. The Banking system subjects performance was "very poor" in the 'Incorrect Facts' subcategory as they committed 57 errors in comparison to only 24 by the Airline Ticketing system subjects. This might be attributable to the fact that the Banking system requires subjects to state almost twice as many correct facts (21) than the Airline Ticketing system (11). The 'Understandability' category consists of four subcategories: 'Ambiguous Information', 'Repeated Facts', 'Very Big UCs' and 'Very Small UCs'. It was revealed that the Banking subjects consistently scored poorer in all subcategories with the exception to the "Ambiguous Information" subcategory. In particular, the Banking system subjects committed 68 "Repeated Facts" mistakes compared to only 18 for the Airline Ticketing system subjects. This can also be attested to the fact that the Banking system would ideally contain three *include* relationships compared to only ideally one *include* relationship in the Airline Ticketing system. Upon examining the subjects' UC models, it was found that subjects have a tendency to restate facts at the *base* UC and the

inclusion UC, leading to a much higher count of ‘Repeated Facts’ by the Banking system subjects. Another finding is that the Airline Ticketing system subjects had no instances of ‘Very Big UCs’ compared to 14 for the Banking system subjects. The Banking system subjects’ UC models proved that many subjects merged the three main services: Transfer Funds, Withdraw Funds and Query Account, into one *base* UC. As stated in (Gomaa 2000), each of these three main services should ideally be described in a separate UC. Another common mistake was that subjects merged the behavior required to validate the card’s PIN into each transaction oriented UCs. Once again, (Gomaa 2000) shows that procedure of PIN validation should ideally be contained in a separate UC. These issues can be avoided in the future by training subjects to better identify individual goals that warrant an individual UC, and to identify tasks that are common between multiple UCs that should be contained in a separate UC. The differences in the two remaining subcategories of ‘Understandability’ were marginal.

3.3.3.7 Group A vs. Group B

Table 3-21 shows the results of the subjects’ performances in each group with respect to each quality attribute. It is important to evaluate the performance of each group as significant differences in performance can bias the results towards a particular technique. For this analysis, the score for each subject in each group for each quality attribute were added for both systems. In this section we are comparing the rescaled performance of the subjects with respect to each quality attribute. As shown in Tables 3-22 and 3-23, no statistically significant difference

was observed between the groups for any of the quality attributes, which indicates that no evidence exists which suggests that subject allocation has introduced a significant bias into the experiment.

Table 3-21: Descriptive statistics of the results

Normalized	Group	n	Median	IQR	95% CI of Median
Inconsistencies	A	34	-0.800	0.407	-1.000 to -0.600
	B	33	-0.833	0.500	-1.000 to -0.500
Completeness	A	34	0.909	0.190	0.810 to 0.952
	B	33	0.857	0.182	0.762 to 0.909
Fault-Free	A	34	-0.750	0.250	-0.875 to -0.750
	B	33	-0.750	0.500	-0.875 to -0.625
Non-Analytical	A	34	-1.000	0.250	-1.000 to -0.750
	B	33	-1.000	0.500	-1.000 to -0.667
Understandability	A	34	-0.500	0.473	-0.750 to -0.286
	B	33	-0.500	0.417	-0.667 to -0.375

Table 3-22: Mann-Whitney test for all quality attributes

Normalized	Group	System	Rank sum	Mean rank	U
Inconsistencies	A	Airline	1144.5	33.66	572.5
	B	Banking	1133.5	34.35	549.5
Completeness	A	Airline	1225.5	36.04	491.5
	B	Banking	1052.5	31.89	630.5
Fault-Free	A	Airline	1091.5	32.10	625.5
	B	Banking	1186.5	35.95	496.5
Non-Analytical	A	Airline	1127.5	33.16	589.5
	B	Banking	1150.5	34.86	532.5
Understandability	A	Airline	1153.5	33.93	563.5
	B	Banking	1124.5	34.08	558.5
Normalized	Difference between medians		95.2% CI	Mann-Whitney U statistic	2-tailed p
Inconsistencies	0.000		-0.167 to 0.14	572.5	0.883
Completeness	0.004		-0.043 to 0.091	491.5	0.379
Fault-Free	0.000		-0.250 to 0.125	625.5	0.411
Non-Analytical	0.000		0.000 to 0	589.5	0.672
Understandability	0.000		-0.167 to 0.167	563.5	0.975

Table 3-23: Cliff’s delta for all quality attributes

Normalized	Cliff’s delta ($\hat{\delta}$)	Variance	Confidence Interval around delta ($\hat{\delta}$)	
			maximum	minimum
Inconsistencies	0.020	0.023	0.303	-0.265
Completeness	-0.124	0.023	0.172	-0.399
Fault-Free	-0.031	0.024	0.263	-0.320
Non-Analytical	0.051	0.016	0.291	-0.196
Understandability	0.004	0.024	0.293	-0.284

3.3.4 Threats to Validity

In this section we present threats to the validity of the study in accordance with the standard classification (Wohlin et al. 2000).

3.3.4.1 Conclusion Validity

Heterogeneity exists in any student-based experiment. If a large degree of heterogeneity exists within the subjects, there is a serious validity threat that the variations in the observed results can be more attributed to individual differences rather than the prescribed techniques. In order to increase homogeneity, our experiment was conducted with graduate Electrical/Computer/Software Engineering students as subjects who were novice UC modelers before this experiment; and who all underwent the same seminars and practice. In addition, our analysis of the relative performance between the groups provides additional evidence that this threat is not a significant concern.

Choosing subjects who are novice UC modelers also aided in assuring that the subjects applied the prescribed methods instead of techniques they might have learned previously. At the start and during each session, subjects were reminded

to use the prescribed technique for the given session and were given a brief review of the technique and how it is applied.

3.3.4.2 Internal Validity

To combat any fatigue or maturation threats, the experimental tasks were scheduled to take place in 3 hour sessions. Subjects were allotted 3 hours in order to complete experimental tasks that would usually last approximately 1 hour (± 15 minutes). Therefore, the subjects did not feel any significant time pressure to complete the tasks.

Population selection was based on availability sampling as subjects participated in this experiment on a voluntary basis, which raises the issue of self-selection. It is only possible to mitigate this threat by conducting this experiment as a mandatory component of a course, which is not a feasible idea since this will affect the learning value that the subjects were originally intended to receive by the course. On the other hand, the fact that subjects participated on a voluntary basis mitigates against morality threats as subjects are self motivated to learn from and participate in this experiment.

3.3.4.3 Construct Validity

The design of this experiment aimed to minimize the construct validity of the dependent variables. Subjects were chosen randomly to form two groups; and effects of individual capabilities, system differences and ordering effects were minimized through a traditional 2×2 fractional factorial design. In addition, the

experimental data is analyzed with respect to these effects. Biasness towards SSUCD or UNL with respect to the systems was eliminated by using the Requirements Documents of two UC models provided by two different authors, who have no connection with this experiment.

3.3.4.4 External Validity

Another inherent external threat is that this experiment utilized students as subjects; hence, it is unsafe to generalize these results to software professionals, specifically analysts. However, in general, the difference between students and professionals is not always clear cut with respect to software engineering related activities, as reported by Höst *et al.* (Höst et al. 2000) and Arisholm *et al.* (Arisholm et al. 2003) in other controlled experiments. In fact in Arisholm *et al.* (Arisholm et al. 2006), the authors argue that students are better representatives in controlled experiments than professionals. The authors argue that professionals have a tendency to stray away from the techniques they were instructed to apply and resort to techniques they developed from previous experiences in industry. In industry, users of SSUCD will be experienced professionals rather than inexperienced students. The results presented by this experiment are only valid when the subjects are inexperienced. Another experiment will be required to evaluate the effect that previous experiences might have on the application of SSUCD.

As is often the case with controlled experiments, our experiment was conducted on a relatively small artifact; and hence it is unsafe to generalize it to

full-scale industrial settings and artifacts. Full-scale industrial UCs are of a significantly larger scale than those that were considered in this study. Larger UC models will inherently require more time to develop compared to those developed in our experiment. Users of SSUCD will naturally become less conformant to its formal syntax when using SSUCD over a longer period time. This issue can however be alleviated through tool support (SAREUCD (STEAM 2009)). Moreover, larger UC models contain more actors, UCs and relationships connecting them, all of which should be textually described. The larger number of elements in a UC model increases vulnerability to “inconsistencies”. Industry-like UC models represent larger and more complex systems, where the interactions between actors and the system are more intricate and the functionalities offered by the system are more sophisticated. For such UC models, there is a greater vulnerability towards missing required “correct” information and inserting “incorrect” information. As the number of functionalities increase, there is a greater possibility to combine too much functionality into one UC creating ‘Very Big UCs’, and vice versa, creating ‘Very Small UCs’. Naturally, with more to describe, there will be more instances of repetitive information and ambiguous information. All these are elements that hinder the understandability quality of a UC model. As SSUCD is however designed to increase the qualities of UC models, it is believed that it is ideally positioned to perform well under these circumstances, but clearly another study is required to start exploring this conjuncture.

Chapter 4

Improving the Quality of Use Case

Models Using Antipatterns

4.1. Introduction

While UC models are simple to create and read; this simplicity is often misconceived, leading practitioners to believe that creating high quality models is straightforward. Therefore, many low quality models that are inconsistent, incorrect, contain premature restrictive design decision and contain ambiguous information are produced. To combat this problem of creating low quality UC models, this Chapter presents a technique that utilizes antipatterns as a mechanism for remedying quality problems in UC models. The technique, supported by the tool ARBIUM, provides a framework for developers to define antipatterns. The feasibility of the approach is demonstrated by applying it to a real-world system. The results indicate that applying the technique improves the overall quality and clarity of UC models.

4.2. Related Work

Many researchers and practitioners have devised techniques to improve the quality of UC models. The following is a brief summary of their approaches:

4.2.1. Computer-supported Verification of UC Models – State of the art

Berenbach (Berenbach 2004) describes a set of software-supported (DesignAdvisor) heuristics to create large verifiable analysis models. This approach can be highly restrictive as many organizations only use a subset of UML; moreover, many organizations have procedures that utilize in-house design heuristics. These restrictions are resolved by ARBIUM, the tool presented in this Section. ARBIUM provides support for analysts to define and verify their own heuristics in addition to being equipped with a set of predefined rules that are applicable to any UC model. The antipatterns defined in this Chapter encompass all of the heuristics presented in (Berenbach 2004) that pertain to UC modeling. It is believed that this approach, Berenbach (Berenbach 2004), presents the current state of the art in computer-supported verification of UC models; and hence, this approach will be compared against our approach in the case study presented in Section 4.6. The heuristics in (Berenbach 2004) will be presented in Section 4.5 and the antipatterns that embody these heuristics will be identified.

4.2.2. Other Approaches

The work presented in this Chapter should be regarded as building upon foundations laid by others. However, most of these pre-existing guidelines are

informal and are provided at a very abstract level. In this Section, we will briefly outline other related work which tackles the identified problem.

The UC modeling inspection technique presented in (Anda et al. 2002) is based upon recommendations provided in (Armour et al. 2000; McCoy 2003; Schneider et al. 1998), is focused on textually-oriented domain-dependent defects in UC models. In order to effectively apply these guidelines and inspection techniques, a great deal of UC modeling expertise is required and therefore these techniques will not be evaluated in Section 4.6. Linguistic techniques (Fantechi et al. 2002; McCoy 2003) and tools (McCoy 2003; Ren et al. 2004; Ryndina et al. 2004) do not perform any verification upon the semantics of the UCs and their relationships. However, UC modeling semantics are carefully considered when developing antipatterns and applying our technique. UC refactorings (Butler et al. 2002; Ren et al. 2003, 2004; Rui et al. 2003; Xu et al. 2004) were developed to address simple defects in UC models. The refactorings are based on simple heuristics which can be found in a small subset of our antipatterns. Ryndina et al. (Ryndina et al. 2004) developed a computer-supported approach to verify UC models. However, the approach does not support the basic UC modeling syntax defined in (OMG 2005); specifically (a) all types of relationships amongst UCs, (b) the generalization relationship between actors and (c) multiple actor associations with a single UC. ARBIUM is designed to support these basic UC modeling notations.

It should be noted that it is not necessary to apply the antipatterns technique exclusively. In fact, we recommend that other approaches should be

used in addition to using antipatterns. The resulting UC models will be of higher quality in comparison to using any approach exclusively.

4.3. UC Modeling Antipatterns

The technique presented in this Chapter focuses on deficiencies that require human cognition to verify. Therefore, the approach can be characterized as “risk-based”, meaning that a “poor” UC modeling structure does not necessarily indicate that a defect certainly exists; rather it indicates that the structure in question may lead to potential defects. In this section, we describe a new technique to find these situations in UC models. The final judgment, with regard to correctness, can only be taken by a domain expert. The proposed quality improvement technique is based on identifying modeling practices that are likely to lead to harmful consequences. While it is impossible to formally analyze the UNL found in textual descriptions, UC diagrams can be formally analyzed due to their adherence to a rigorous syntax (OMG 2005). Therefore, an informal review process will be required to analyze textual descriptions, while inappropriate design decisions in UC diagrams can be formally detected.

To effectively apply this approach, a repository of (anti)patterns which articulate poor UC modeling habits and decisions is required; our initial repository is described in Section 4.3.6. An advantage to this approach is that it can be applied in the early phases of the development cycle where UC models are often incomplete.

4.3.1. Advantages of Using Antipatterns: What Can Antipatterns Do?

Learning from previous experiences and mistakes is the main concept behind using antipatterns. An antipattern explains why a given structure may cause deficiencies in a UC model. An antipattern will also provide a detection mechanism to guide modelers to areas in the UC model where an antipattern may exist, be it in the UC diagram, the descriptions, or both. Most importantly, an antipattern will explain why such a debatable structure seemed appropriate in the first place. Finally, an antipattern provides suggestions upon improving the current structure to avoid potential consequences. Basically, an antipattern provides key information to guide modelers from a fallacious solution to a superior solution (Coplien 2007). Table 4-1 shows the antipattern template used in this Section. The purpose of each field is described briefly in Table 4-1 and in more detail in Section 4.3.2.

Table 4-1: Antipattern template

Antipattern Name: The title of the antipattern.

Description: A description of the faulty decisions or techniques.

Rationale: A list of the deceptive or seductive reasons as to why the fallacious solution seemed to be appropriate.

Consequences: A list of the harmful consequences that could be sustained from applying the fallacious solution.

Detection:

Where – A guide to the areas where the antipattern can exist.

How – Instructions that are used to positively identify a match for the

antipattern.

Improvement: A list of actions that can be performed to convert a fallacious solution into a superior solution or avoid the fallacious solution.

4.3.2. Matching Antipatterns With UC Models

As mentioned earlier, poor modeling decisions may exist in the UC diagram, the descriptions, or both. The “Detection” section in an antipattern contains detailed guidelines to match the antipattern. For poor modeling decisions that exist in UC diagrams, an antipattern will outline a set of diagrammatic elements that represent a debatable structure. Detecting a match for such antipatterns can be achieved by juxtaposing the antipattern’s stated unsound diagrammatic structure with the actual UC diagram. As for poor decisions that exist in textual descriptions, the “Detection” section will guide analysts to particular field(s) of a UC template where an antipattern match can be detected. If an antipattern is matched; the analysts are then required to verify the correctness of the UC model.

Upon reviewing an antipattern match, corrective measures may be required. If corrective measures were undertaken, this may consequently eliminate previously detected antipattern matches that have not been reviewed. Alternatively, undertaking corrective measures may cause new antipatterns to surface. Therefore, the antipattern matching process must be performed iteratively until all antipattern matches have been addressed.

4.3.3. Using OCL to Describe Unsound Diagrammatic Structures

Unsound structures described in NL are inherently ambiguous. Ambiguity can be eliminated by describing unsound diagrammatic structures referred to by antipatterns using OCL constraints (Warmer et al. 1998). During the matching process, if the constraints were not satisfied, then an antipattern match is detected. Wherever possible, antipatterns will be augmented with OCL statements to automate or semi-automate their detection.

Traditionally, OCL statements are used to describe constraints in class diagrams or object models. In order to describe diagrammatic UC structures using OCL, the UC diagram must be transformed to an object model. This is possible since every instance of a UC diagram conforms to the metamodel provided by OMG (OMG 2005). Each element in a UC diagram maps onto one or more metaclasses. However, it is clearly impractical to expect analysts or domain experts to study hundreds of pages of documentation explaining thousands of metaclasses, most of which are not exclusive to UC diagrams, in order to construct their OCL statements. To increase the accessibility of our approach, a simplified metamodel was created (see Figure 4-1), which contains only four classes and a limited number of associations linking these classes together. All these metaclasses are exclusive to UC diagrams. The simplified metamodel does not need to support the entire notational set of UC diagrams. The smaller metamodel will encourage the adoption of the metamodel by analysts and minimize the learning curve while supporting the notational subset most

commonly used, and which encompasses most UC diagrams. The metamodel can easily be extended to support any additional notation required.

The metaclasses shown in Figure 4-1 represent actors, UCs, the association relationship, the *generalization* relationship (both between actors and UCs), abstraction, the *include* relationship, the *extend* relationship and extension points. The following is a brief description of the metamodel elements:

- Instances of the **UseCase** and **Actor** classes are assigned names using the **name** attribute, and a Boolean **abstract** attribute that indicates whether they are *abstract* or *concrete*. The **extensionLocation** attribute of the **ExtendsAt** association class is used to state the extension point to which an *extend* relationship link is referring.
- For the **ExtendsAt** association class, the **extensionUC** role indicates that the *extension UC extends the base UC*. The *extended* extension point is referenced by the **extensionLocation** attribute. The **ExtendsAt** relationship is required since the extension point referred to is a property of the *extend* relationship. The *base UC* is specified using the **base** role.
- An *extend* relationship that does not refer to an extension point is indicated using the **Extends** association. For the **Extends** association, the **extension** role indicates the *extension UC*. Meanwhile, the *base UC* in turn is specified using the **base** role.
- An *include* relationship is specified using the **Includes** association. For the **Includes** association, the **inclusion** role indicates the *inclusion UC* being included by a *base UC* which is in turn indicated by the **base** role.

- The *generalization* relationship between UCs is supported by the `Specializes_use_case` association. The `Specializes_use_case` association has one `UseCase` object assigned the `parent` role, while another `UseCase` object is assigned the `child` role.
- The *generalization* relationship between actors is supported by the `Specializes_actor` association. The `Specializes_actor` association has one `Actor` object assigned the `parent` role, while another `Actor` object is assigned the `child` role.
- The `Associated_With` association represents an association relationship between an actor and a UC. The actor end of the association relationship is assigned the `actorEnd` role, while the UC end of the relationship is assigned the `useCaseEnd` role.
- Directed associations are represented with the `DirectedAssociation` association class, the `directedActorEnd` role indicates the actor involved in the association, while the `directedUCEnd` role indicates the UC involved in the association. The String attribute `directTowards` can be set to either “UC” or “Actor” to indicate where the association link is directed towards.

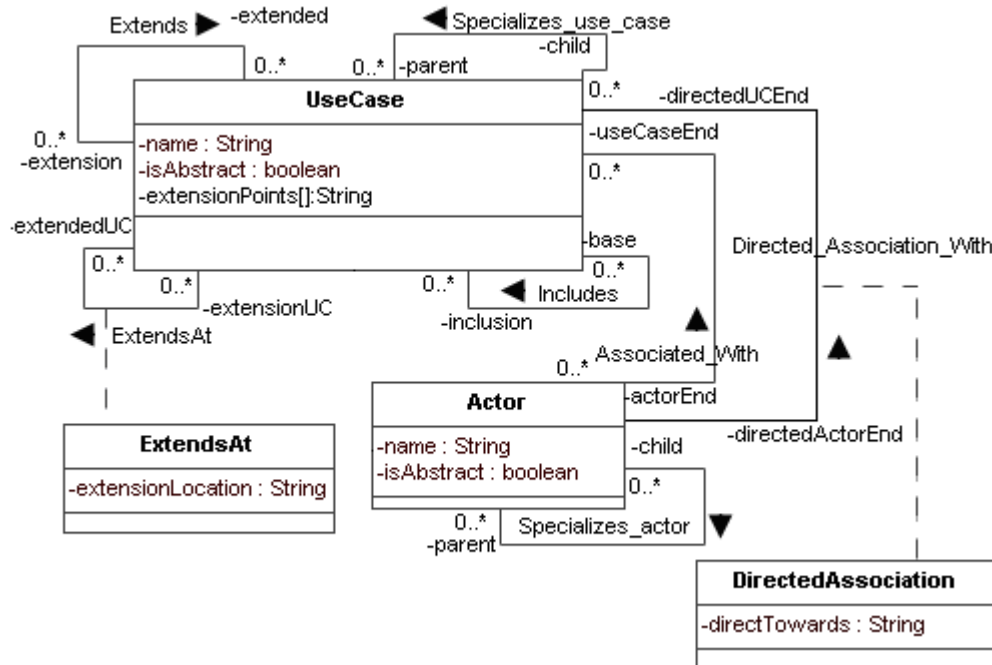


Figure 4-1: The simplified version of the UC metamodel used in ARBIUM³

Automated support is available to examine diagrammatic constructs using OCL and the above metamodel. Unfortunately, examination of textual descriptions remains a manual process. The tool supported antipatterns shown in Section 4.4 are augmented with OCL statements whenever possible to automate or semi-automate their detection.

4.3.4. Domain Independent vs. Domain Dependent Antipatterns

Antipatterns can either be domain-independent (DI) or domain dependent (DD). DI antipatterns make no assumptions about the underlying domain and hence are applicable to any UC model. Researchers can derive DI antipatterns by understanding the semantics of the UC modeling notation and the purpose behind

³ Some elements of the metamodel are present to more effectively benefit from the features provided by USE (UML-based Specification Language), which is used in addition to ARBIUM to automate the detection process (Section 4).

each component of a UC model. DD antipatterns represent additional, specialized antipatterns which seek to encode an organization’s specific objectives for a specific project or domain. Analysts should collaborate with domain experts to develop DD antipatterns. Using OCL and the simplified metamodel, analysts can quickly define DD antipatterns.

Antipatterns can be further subdivided – with respect to their suitability of being machine readable (see Table 4-2). The principal advantage of antipatterns that are machine readable is that they can be (semi-)automatically matched. Diagrammatic structures are described using OCL as a set of constraints, which is used by ARBIUM to perform the matching process. The process of matching textual patterns cannot be automated as UNL cannot be formally analyzed. Therefore, a review process is required to detect matches for textual patterns described by antipatterns. The availability of tool support to match an antipattern is dependent on the information provided in its “Detection” section.

Table 4-2: Types of antipatterns

Situation	Full Automation Support Available	Semi-automation Support Available	No Automation Support Available
Domain-independent Antipatterns	(1)	(2)	(3)
Domain Dependent Antipatterns	(4)	(5)	(6)

Type (1): This Section mainly focuses on this type of antipattern. A large number of Type (1) antipatterns are presented in detail in Section 4.3.6. The detection process of these antipatterns can be fully automated as they only require analysis of UC diagrams.

Type (2): For this type of antipattern, ARBIUM can be used to detect the diagrammatic structure described by an antipattern; subsequently a review process is required to analyze the corresponding textual descriptions.

Type (3): For this type of antipattern, the review process needs to be conducted manually since these antipatterns require the examination of textual descriptions. No automation support can be provided to detect this type of antipattern. A number of Type (3) antipatterns are described in Section 4.3.6.

Type (4): These DD antipatterns are machine-readable. Analysts use the simplified metamodel to compose OCL statements that describe the debatable construct. Analysts will need to collaborate with domain experts to develop these antipatterns.

Type (5): Similar to Type (4) antipatterns, analysts will need to collaborate with domain experts to develop this type of antipatterns. Unlike Type (4) antipatterns, Type (5) antipatterns require a review or inspection of textual descriptions in addition to the UC diagrams.

Type (6): DD antipatterns are again developed through a collaborative effort between analysts and domain experts. A manual review process needs to be performed in order to match this type of antipatterns as they require examination of textual descriptions.

4.3.5. A Systematic Review Process for Antipattern Development

The antipatterns developed in this Chapter are based on widely accepted guidelines and best practices, as well as a thorough understanding of the UC modeling notational syntax and semantics. A systematic review process was deployed in order to obtain such relevant information. The review process used is a light-weight adaptation of the systematic literature review proposed in (Kitchenham 2004). In (Kitchenham 2004), the purpose of a systematic review process is defined as:

“...a means of identifying, evaluating and interpreting all available research relevant to a particular research question, or topic area, or phenomenon of interest.”

A systematic review process consists of three main steps: (a) Planning, (b) Execution and (c) Result Analysis. In the planning stage, research objectives are identified and a review protocol is created. The purpose of the review protocol is to specify the research questions that need to be answered, or a research objective

that needs to be satisfied, and the method by which the review process will be executed. During the execution phase, a broad spectrum of literature is selected, which is then subjected to preset inclusion and exclusion criterion. Before executing the review process, it is required to approve the review process protocol in order to determine its feasibility. Approval of the review process protocol can be obtained by asking subject matter experts or by conducting a trial execution of the protocol. Similar to the planning phase, the execution phase is also evaluated to determine if the literature identified for consideration appropriately yields relative information that can be used towards answering the original research questions. During the results analysis phase, the literature selected for consideration is analyzed and the information of interest is gathered and synthesized to answer the original research questions and objectives. The reason for devising a light-weight version of the systematic review process presented in (Kitchenham 2004) is that while the rewards of a full systematic literature review is greater than the review process used in this work, it was revealed that there remains unresolved issues with conducting a full systematic review (Mian al. 2005). In particular, a full systematic review process is extremely time-consuming (Woodall et al. 2006). A full review process entails quantitative analysis which is not applicable due to the qualitative nature of the information being searched for in this research. The light-weight adaptation is to use two databases (Amazon and IEEE Xplore), in addition to the official OMG UML specification (OMG 2005). The following subsections will describe the review process protocol in more detail.

4.3.5.1 Data Classification Scheme and Scope

The result of the literature review is to determine information regarding certain aspects of UC modeling. These aspects are used to categorize the analyzed information using the following data classification scheme:

- Information that explains UC modeling, its notation, syntactical rules and semantics.
- Information regarding how to properly apply UC modeling, such as best practices, recommendations, high quality attributes, patterns and blueprints.
- Information regarding what not to do in UC modeling, such as mistakes, pitfalls, drawbacks and poor quality attributes.

To gather these types of information, a scope for the review process was set. The scope of the review only considered literature available in the form of books, scientific journals, conference and workshop proceedings, as well as the OMG UML specification (OMG 2005). This criterion was chosen as it presents the most credible set of scientific sources for information related to UC modeling. Books contain valuable UC modeling industrial experience, recommendations and best practices. Scientific journals, conference and workshop proceedings provide cutting-edge research results and solutions in the field of UC modeling. Finally, the OMG UML specification represents the definitive source for the UC modeling notation, syntactical rules and semantics.

4.3.5.2 Search Strategy

The search for books was conducted using the Amazon database (<http://www.Amazon.com>) as it can be argued that it is one of the largest databases for books available on the Internet. The search for scientific journals, conference and workshop proceedings was performed using the IEEE Xplore database (<http://ieeexplore.ieee.org>), since the IEEE Computer Society is considered a leading venue for Software Engineering research results (Glass et al. 1993). The OMG UML specification is available online at (OMG 2005).

The correctness of the search results returned by search engines is dependent on the search terms used to execute the search. The online search process was conducted as follows:

1. Derive the most relevant terms from the research objective.
2. Derive the most relevant terms from literature already reviewed prior to this research work.
3. Identify any alternative synonyms and spellings for the set of terms derived.
4. Derive as many search term combinations as possible.

Based on this strategy, a large number of search terms were developed, such as:

Use Cases *OR* Use Case Antipatterns *OR* Use Case Models *OR* Use Case Modeling *OR* Use Case best practices *OR* Use Case pitfalls *OR* Use Case mistakes *OR* Use Case suggestions *OR* Use Case drawbacks *OR* Use Case authoring *OR* Use Case descriptions *OR* Use Case diagrams *OR* Use Case

recommendations *OR* Use Case warning *OR* Use Case syntax *OR* Use Case metamodel.

4.3.5.3 Filtering the Results

An inclusion and exclusion criteria is required to filter the results returned by the search matches. The purpose of the inclusion criteria is to ensure that only literature that discusses UC modeling itself is included in the analysis. The purpose of the exclusion criteria is to avoid literature that only mentions UC modeling in a contextual manner. The following inclusion and exclusion criteria were applied:

- **Inclusion criteria:** Can this literature resource be categorized to discuss UC modeling, including its application, best practices, syntax, pitfalls, recommendations, quality improvement, notation, syntactical rules or semantics?
- **Exclusion criteria:** Can this literature resource be considered to only mention the terms UC modeling or only provide a UC model without discussing the practical aspects of UC modeling or its notation?

For each book, the inclusion and exclusion criterion were applied by reading the title, preface and its short description if available (and if necessary). For books that satisfy the inclusion criteria while not satisfying the exclusion criteria, its table of contents is examined to determine which chapters relate to UC modeling. Upon determining the relative chapters in the book, the inclusion and exclusion criteria are applied once again to exclude irrelevant chapters. For each journal,

conference and workshop proceedings, the inclusion and exclusion criteria were applied by reading the title and abstract (if necessary). For the OMG UML specification, the inclusion and exclusion criteria were applied by reviewing the table of contents and determining the chapters relevant to UC models. The search strategy was successfully piloted and its results were verified before executing a full search process.

4.3.5.4 Filtering the Results

The following process was applied to execute the search for the information required to satisfy the research objective:

1. Apply the search strategy outlined in Section 4.3.5.2.
2. Record bibliographical details of matches returned by search engines as a result of executing the search strategy.
3. Apply the inclusion and exclusion criteria outlined in Section 4.3.5.3.
4. Remove bibliographical details of matches that do not satisfy the inclusion criteria and do satisfy the exclusion criteria.
5. For each book in the bibliography after applying step (4), the inclusion and exclusion criterion are applied to record relevant chapters.
6. The resulting set of relevant chapters in books and relevant research papers are fully read. The relevant chapters in the OMG UML specifications are fully read. The contents of relevant chapters and papers are analyzed and categorized according the data classification scheme described in Section 4.3.5.1.

The set of references yielded from applying the filtering process are shown below:

- Books: (Adolph et al. 2002; Armour et al. 2000; Bittner et al. 2002; Booch et al. 2005; Cockburn 2000; Gomaa 2002; Kroll et al. 2003; Kruchten 1998; Kulak et al. 2000; Overgaard et al. 2005; Schneider et al. 1998)
- Journal papers: (Cockburn 1995; Constantine et al. 1999; Jaaksi 1998; Medvidovic et al. 2002)
- Conference proceedings: (Anda et al. 2001a; Anda et al. 2002; Anderson et al. 2001; Ben Achour et al. 1999; Berenbach 2004; Butler et al. 2002; Chandrasekaran 2008; Fantechi et al. 2002 Firesmith 1999; Gogolla et al. 2002; Lilly 1990; McCoy 2003; Ren et al. 2003, 2004; Rui et al. 2003; Xu et al. 2004)
- Workshop proceedings: (Fabbrini et al. 2001; Gomaa 1997)
- Formal syntax specifications of Use Case models: (OMG 2005)

4.3.5.5 Results Analysis

Upon executing the search process, the resulting literature matches were read and a great deal of qualitative information was gathered and categorized. Information in the literature that is presented in the form of UC modeling best practices, patterns and blueprints, is analyzed to develop antipatterns that are based on not following these recommendations. Information regarding poor UC modeling techniques, patterns and poor quality attributes, is analyzed to develop antipatterns that are based on committing such mistakes and pitfalls. Finally,

information that explains the UC modeling approach, its notational syntactical rules and semantics, is analyzed to develop antipatterns based on modeling structures that will violate the intended semantics. The set of antipatterns developed are presented in Section 4.3.6. The set of antipatterns presented in this Chapter do not encompass every possible antipattern that may exist; it is however the most comprehensive set of anti-patterns which can be derived from the current literature on the topic.

4.3.6. Examples of UC Modeling Antipatterns

➤ Antipattern Name

a1. Accessing a *generalized concrete* UC - **Automation Support: Type (1)**

▪ Description

A family of UCs that represent a framework of services offered by a system can be defined using the *generalization* relationship. The services offered by these UCs are very similar and share a common theme. Modelers can define a hierarchy between the UCs using the *generalization* relationship. The general behavior shared by these services is contained in a *generalized* UC. Meanwhile, specific behavior tailored to cater to some requirements of the system's users, are contained in *specialized* UCs. To access this framework of services, an actor is associated with the *generalized* UCs to indirectly access all of the services offered by this family of UCs.

- **Rationale**

An association can be created between an actor and a *generalized* UC for two reasons:

- (1) The *generalized* UC contains behavior that individually can be useful to that actor.
- (2) The operational mechanisms of the *generalization* relationship in UC diagrams are similar to that of class diagrams. Therefore, modelers may utilize the concept of polymorphism in their UC model. Hence, when an actor initiates a *generalized* UC, the service request can be delegated to one of its *specializing* UCs.

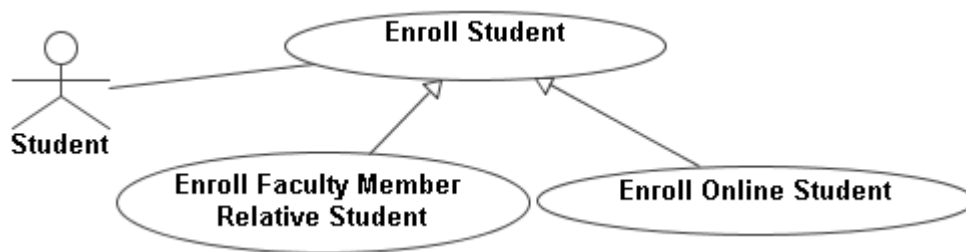


Figure 4-2: A good scenario of an actor being directly associated with a *generalized* UC

For example, in Figure 4-2, the UC Enroll Student is a *concrete* UC that describes the procedure of enrolling a regular student. Special types of students enrolling into the University receive special consideration. A student who is a relative of a faculty member is entitled to a tuition discount, in addition to free access to the University's health services. This type of student will be enrolled using the Enroll Faculty Member Relative Student UC. On the other hand, the university offers several online programs. Students enrolled in such programs are considered off-campus students and thus they

are relieved from paying the University’s health services. Moreover, since online courses are virtual, the system does not need to check for availability inside the classroom. Students enroll into online programs using the Enroll Online Student UC.

In summary, special types of students are enrolled using one of the *specializing* UCs. Meanwhile, a regular student will be enrolled using the *generalized* UC.

- **Consequences**

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. Figure 4-3 provides a visual overview of the operational mechanisms of the *generalization* relationship.

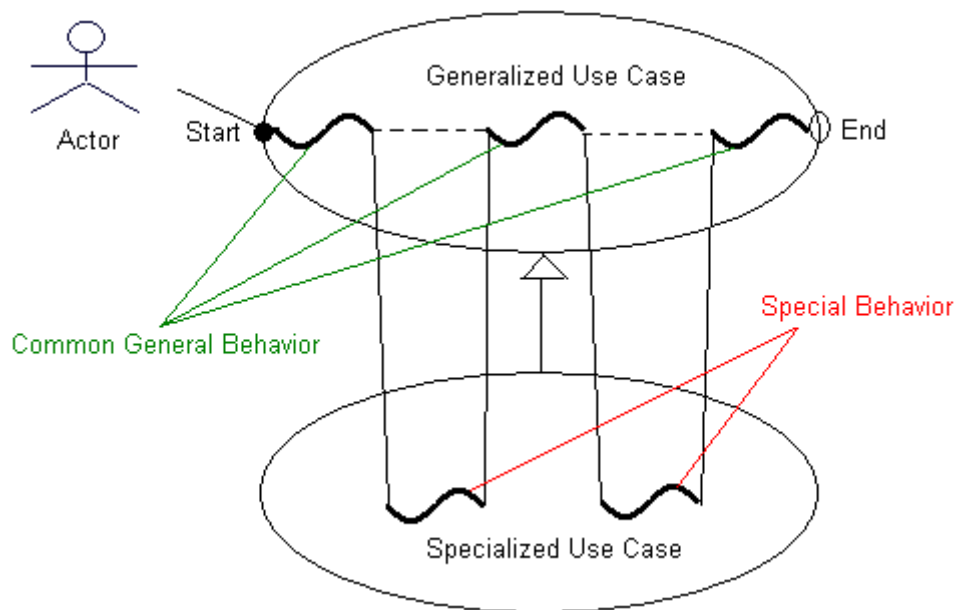


Figure 4-3: The execution flow of a *generalization* relationship

If the *generalized* UC is concrete, it can stand alone as a complete UC which can be exclusively initiated. However, if an actor makes an exclusive initiation request to such *generalized* UC, incomplete meaningless behavior will be executed. Figure 4-4 shows a shoe store system that exposes this pitfall.

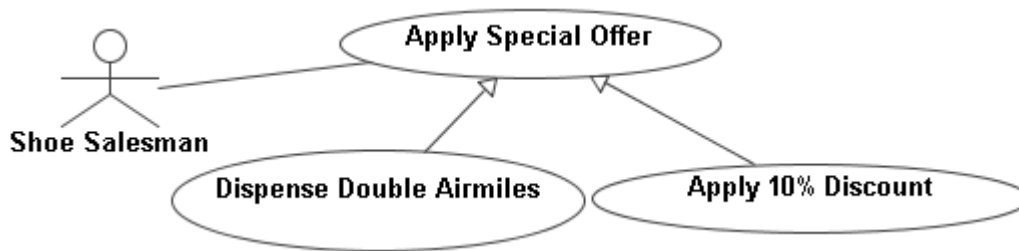


Figure 4-4: A bad scenario of an actor being directly associated with a *concrete generalized* UC

The shoe salesman may apply one of two promotional offers to a shoe purchase. The first offer allows customers to get double the airmiles they normally would get on their purchases. This offer is applied by performing the Dispense Double Airmiles UC. The other offer entitles customers to a 10% discount on their purchases. This offer is applied by executing the Apply 10% Discount UC. The *generalized* UC Apply Special Offer is *concrete*, and it contains general behavior responsible for applying any promotional offer. Since this *generalized* UC is *concrete*, it can be exclusively initiated by the Shoe Salesman. If the *generalized* UC was exclusively initiated, no particular special offer will be applied to a given purchase.

- **Detection**

Where – Search for any *generalized* UCs in the “UC Diagram”. **How** – – If a *generalized* UC is found, find out if this *generalized* UC is *concrete*. The

generalized UC must be associated with an actor. *Concrete* UCs are labeled using regular font. Meanwhile, *abstract* UCs are labeled using *italic* font.

OCL Description:

```
context UseCase
```

inv AccessingGeneralizedUseCaseByActor:

```
not ( (not (self.isAbstract)) and self.actorEnd->size  
> 0 and self.child->size >0) inv
```

AccessingGeneralizedUseCaseByActor:

```
not ( (not (self.isAbstract)) and self.actorEnd->size  
> 0 and self.child->size >0)
```

▪ **Improvement**

(1) Unlike *concrete* UCs, an *abstract* UC cannot be initiated. Setting the *generalized* UC in the shoe store system shown in Figure 4-4, which contains incomplete behavior, to be *abstract* will prevent it from being initiated (see Figure 4-5):

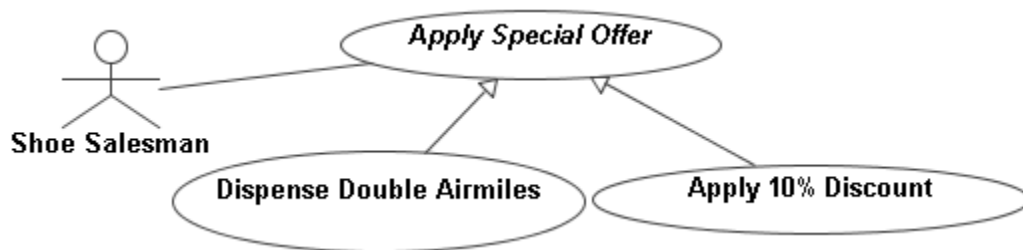


Figure 4-5: The *generalized* UC is set to be *abstract* to make sure that one of its *specializing* UCs services the actor’s request.

(2) Explicit associations between the actor and the *specializing* UCs can be created in place of the association between the actor and the *generalized* UC (see Figure 4-6). The explicit associations with the *specializing* UCs will enforce the service request to be performed through one of the *specializing* UCs. Hence, this technique will ensure that the *generalized* UC is not initiated directly and exclusively. It is worth mentioning that the improved solution presented in (1) is often superior. This approach may cluster the UC model, especially if there are many *specialized* UCs. Moreover, the solution presented in (1) preserves the semantics behind the original design.

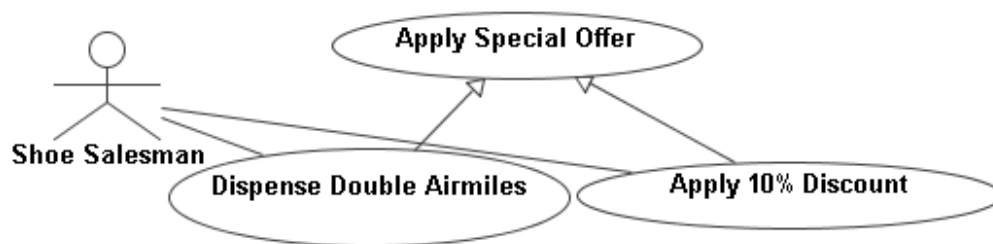


Figure 4-6: Direct access to the *generalized* UC is avoided and replaced with direct access to its *specializing* UCs

➤ **Antipattern Name**

a2. UCs containing common and exceptional functionality - Automation

Support: Type (1)

▪ **Description**

The reuse of a preexisting UC is achieved by making it both an *extension* UC and also an *inclusion* UC. For example, in a car dealership system (see Figure 4-7), when a new car arrives at the dealership, it is recorded into the database of the

dealership using the Add New Car UC. A precondition to adding the new car to the dealership's database is that the car must not already exist at the dealership. Therefore UC Add New Car *includes* UC Car Not Found to check for that precondition. UC Update Car's Information is responsible for updating the information related to a particular car, such as, its current mileage or where it is located (assuming several branches). In order to update a particular car's information, this car must exist in the dealership's database. An error is generated if the given car does not exist in the database. Therefore, UC Update Car's Information is *extended* by UC Car Not Found to handle this error generated.

This eventually leads to the discouraged construct where a UC (Car Not Found), is an *extension* and *inclusion* UC.



Figure 4-7: UC Car Not Found was incorrectly used for the purposes of containing common functionality and exception-handling behavior.

- **Rationale**

Object-oriented modeling and design strongly promotes the concept of reuse. When modelers are constructing their UC models, they are keen to reuse much of the functionality contained preexisting in UCs. UC modeling offers mechanisms through its *extend*, *include* and *generalization* relationship to allow this reuse. Reusing UCs also prevents the cluttering of the UC with many redundant UCs. 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.

- **Consequences**

The shared UC currently contains common and exceptional behavior required by the two *base* UCs. Therefore, when either of the *base* UCs initiate the shared UC, additional undesired functionality is performed. To further elaborate, during the operation of the *including base* UC Add New Car, UC Car Not Found is initiated to check that the given car does not exist in the database. However, UC Car Not Found will unnecessarily also perform the procedure of trying to update a car's information that does not exist. On the other hand, if the UC Update Car's Information is performed to update the information of a given car that does not exist in the database, UC Car Not Found is initiated to handle the generated error. In this situation, the UC Car Not Found will unnecessarily check if the given car does not exist in the system.

- **Detection**

Where –Search for any *included* UCs in the UC diagram. **How** – If an *inclusion* UC is found, check if this *inclusion* UC is *extending* other UCs.

OCL Description:

```
context UseCase
```

```
inv ExtendingMoreThanOneUseCase:
```

```
not (self.extended->size + self.extendedUC->size > 1)
```

- **Improvement**

Check if the shared UC contains functionality suitable for only one of the *base* UCs. This can be achieved by examining the contents of the shared UC.

(1) If the shared UC contains functionality suitable for only the *base* UC that *includes* it, the *extend* relationship should be removed. A new *extension* UC should be created to handle the exceptional situation generated by the other *base* UC. The resulting model is illustrated in Figure 4-8.

(2) If the shared UC contains functionality suitable only for the *base* UC that it *extends*, the *include* relationship should be removed. A new UC should be created and *included* by the other *base* UC, again resulting in the model shown in Figure 4-8.

In both cases (1) and (2), the UCs should be renamed to be more indicative of their respective purposes.

(3) In the case that the shared UC does indeed contain both common behavior and exception handling behavior. The shared UC should be split into two separate UCs. Each of the newly created UCs should only contain functionality appropriate to the *base* UC. Once again, resulting is the same model (see Figure 4-8).

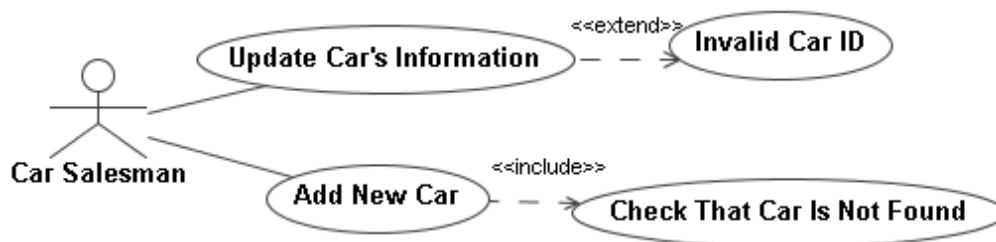


Figure 4-8: The shared UC is broken into two separate UCs, each serving a different purpose.

➤ **Antipattern Name**

a3. Functional decomposition of UCs: Using the *include* relationship -

Automation Support: Type (1)

▪ **Description**

Functional decomposition most commonly occurs due to the misuse of the *include* relationship. The *inclusion* UCs are set to describe tasks that are required to perform a complete service that is offered by their *base* UC. The tasks described by the *inclusion* UCs represent functions in a program, or menu options. For example, in the espresso machine system shown below (see Figure 4-9), the *inclusion* UCs together are used to prepare a cup of coffee. Such *inclusion* UCs are not used by any other UC and are not associated with any actors

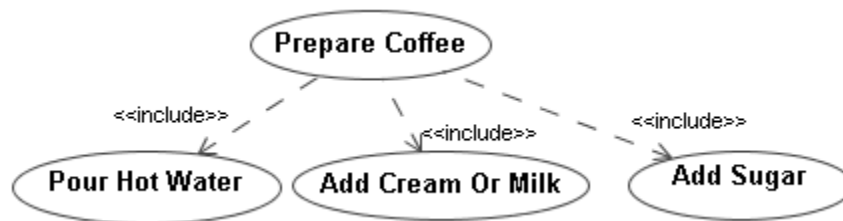


Figure 4-9: Functional decomposition of the Prepare Coffee UC

▪ **Rationale**

Dissecting analytical UCs into functions yields a set of “smaller” UCs that are naturally easier to implement. Overall, this will lead to a speedier implementation of the system. Creating “smaller” UCs is particularly attractive to modelers since they are easier to understand and code. Consequently, in later development phases, the “smaller” UCs will be easier to test and maintain. Functional

decomposition can be use to embody design decisions that analysts would like to enforce throughout the development of a system.

- **Consequences**

UC modeling is used to model the behavior of a system at the conceptual level. UCs should represent services that a system offers to its actors. An actual UC describing a complex service can easily be decomposed to hundreds of collaborating functions. The “smaller” UCs offer no value to the system’s users if executed individually. Being able to abstract the actual service offered by these numerous functions by examining many “smaller” UCs is a very difficult task. One can at best guess what service these UCs will offer when performed together. Therefore, the “smaller” UCs created as a result of functional decomposition obscure the real purpose of the system. For complex systems, it is more likely that this “guess” will be incorrect. At that point, it is up to the designers’ domain knowledge to design the correct system. Moreover, functional decomposition of UCs may lead to more complex descriptions of the interactions between the actors. Functional decomposition embodies premature design decisions which severely limits the creativity of designers and enforces them to abide to these decisions.

- **Detection**

Where – Look for an *inclusion* UC inside the UC diagram. **How** – Upon finding an *inclusion* UC, count the number of UCs that *include* it. If the *inclusion* UC is

included once, then the antipattern is matched. It is important to note that in order to positively match this antipattern; the *inclusion* UC must not be associated with any actors or UCs in the UC model.

OCL Description:

```
context UseCase
```

```
inv NotJustOneInclude:
```

```
not (self.base->size = 1)
```

- **Improvement**

The behavior described in *inclusion* UCs must be combined into UCs that individually offer a complete and meaningful service to a system's user.

For the espresso machine system shown in Figure 4-9, the behavior described by UCs Pour Hot Water, Add Cream Or Milk and Add Sugar should be merged into the UC Prepare Coffee. It can be deduced that the user will not benefit from pouring hot water only, or having a cup with only sugar in it. The real value offered to the user is the preparation of a cup of coffee. Hence, from a conceptual point of view, a single UC called Prepare Coffee (which already exists) should be individually responsible for preparing a cup of coffee.

➤ **Antipattern Name**

a4. Functional decomposition of UCs: Using the *extend* relationship -

Automation Support: Type (2)

▪ **Description**

Another form of functional decomposition is the improper use of the *extend* relationship. An *extension* UC inserts additional behavior to a *base* UC that it *extends*. *Extension* UCs are required to know the exact locations, known as “extension points”, inside a *base* UC where their additional behavior will be inserted. Naturally, this additional behavior is very specific to the respective *base* UC. If an *extension* UC contains general behavior that would be useful to more than one *base* UC, this would be a strong indication that the *extension* UC has degraded into a function. For example, in the following racquet sports store system (see Figure 4-10), the UC Equipment Damaged *extends* both the Sell Racquet and Sell Ball UCs.



Figure 4-10: Improper use of the *extend* relationship to promote functional decomposition

It is the employee’s responsibility to ensure that any merchandise being sold is not damaged. Whenever the employee encounters faulty merchandise, the *extension* UC Equipment Damaged is initiated. In the case of a damaged ball, the defective ball is discarded and a new ball is handed to the customer.

Meanwhile, an in-store technician can fix defective racquets, however if the racquet is severely damaged, it is send back to the manufacturer for an exchange. Hence it can be deduced that there are two different procedures for handling defective balls and racquets, yet the structure shown in Figure 4-10 would indicate a single procedure for handling any type of damaged merchandise.

- **Rationale**

Similar to what is described in the “Rationale” Section of the “Functional decomposition of UCs using the include relationship” antipattern. Moreover, there might be times where the *extension* UC is used to provide general functionality that is specialized by the UCs it *extends*.

- **Consequences**

Similar to what is described in the “Consequences” Section of the “Functional decomposition of UCs using the include relationship” antipattern. Moreover, when functional decomposition is applied using the *extend* relationship; it is often the case that the *extending* UC does not properly handle the exceptional situations caused by the *base* UCs. This situation can be easily detected in the racquet sports store system, since the procedure of handling a damaged racquet differs significantly from the procedure of the handling a damaged ball. Therefore, it can be easily deduced that more than one *extending* UC is required to handle the different exceptional situations occurring.

- **Detection**

Where – Search for *extension* UCs in the UC diagram. **How** – Upon finding an *extension* UC, count the number of UCs that the *extension* UC *extends*. The antipattern is matched if the *extension* UC *extends* more than one UC. It is then required that the analyst examines the behavior described by the *extension* UC to check if it is too generic. If the *extension* UC was found to contain specific behavior, it is then required by the analyst to ensure that this specific behavior is in fact suitable for all the *extended* UCs.

OCL Description:

```
context UseCase
```

```
inv ExtendingMoreThanOneUseCase:
```

```
not (self.extended->size + self.extendedUC->size>1)
```

- **Improvement**

The behavior described in the *extension* UCs must be combined into UCs that individually offer a complete service to a system's user.

For the racquet sports store system illustrated in Figure 4-10, the *extension* UC Equipment Damaged should be divided into two separate *extension* UCs (see Figure 4-11). Each of the newly created *extension* UCs will be specifically designed to more appropriately handle the exceptional situations arising at their respective *base* UCs.

For the case when *extended base* UCs are used to specialize general behavior described by their respective *extension* UC, a *generalization* relationship

would be more appropriate than an *extend* relationship to describe such a relationship.

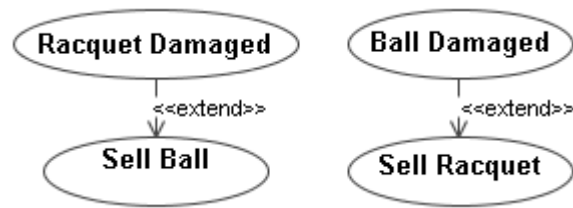


Figure 4-11: Extending UCs disjointed to properly handle different exceptional situations.

➤ **Antipattern Name**

a5. Functional decomposition of UCs: Using pre and postconditions -

Automation Support: Type (3)

▪ **Description**

If a *base* UC is decomposed into “smaller” *base* UCs, it is often the case that the “smaller” UCs need to be performed in particular sequence to properly execute the intended complete service. UC modeling does not provide a “calling” mechanism between UCs whereby UCs can “invoke” or “call” each other. UC modeling deliberately does not provide such mechanisms, since UCs are expected to provide complete services without the need to communicate with each other. To workaround this limitation, modelers misuse the pre and postconditions in UCs to explicitly declare a virtual call sequence between the UCs. It can be deduced that the virtual sequence would most likely be the result of UCs degrading into functions. For example, in a telecommunications system, a *base* UC that is intended to make a phone call is instead decomposed into the “smaller”

UCs shown in Figure 4-12. The “smaller” UCs: Get Dial Tone, Retrieve Phone Number Dialed, Establish Connection and Ring Destination Phone need to execute in the sequence to properly make a phone call.

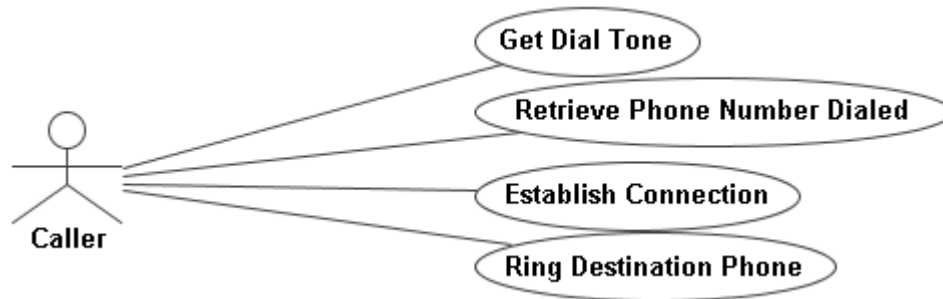


Figure 4-12: Sequencing a set of UCs to make a phone call

There are two methods that can be used to apply this concept (see Figure 4-13):

- (1) Assuming two UCs named “N” and “N+1” respectively. A virtual call sequence can be enforced between these two UCs by stating as a postcondition for UC “N” that UC “N+1” must start and stating as precondition to UC “N+1” that UC “N” must successfully be completed. Recalling the telecommunications system from Figure 4-12, initiation of Retrieve Phone Number Dialed UC is stated as a postcondition of UC Get Dial Tone. Similarly, the successful termination of UC Get Dial Tone is stated as precondition of UC Retrieve Phone Number Dialed.
- (2) Implicitly, by restating the postconditions of UC “N” as the preconditions UC “N+1”. Even though this method may be valid at times, it most likely will lead to the over-specification of the conditions in one of the UCs.

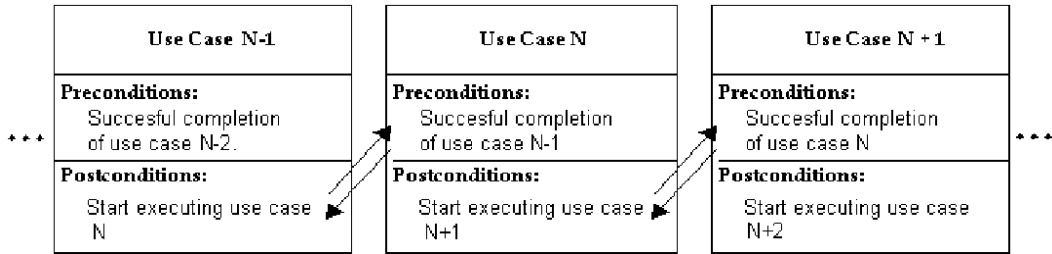


Figure 4-13: Creating a virtual call sequence between UCs using pre and postconditions

- **Rationale**

Similar to what is described in the “Rationale” Section of the “Functional decomposition of UCs using the include relationship” antipattern.

- **Consequences**

Similar to what is described in the “Consequences” Section of the “Functional decomposition of UCs using the include relationship” antipattern.

- **Detection**

Where – In the preconditions and postconditions of each *base* UC. **How** – (a) If it is stated as a postcondition for a UC that another UC needs to be initiated, then the antipattern is matched. (b) If it is stated as a precondition of a UC requires that another UC needs to be successfully completed, then the antipattern is matched. (c) The antipattern is matched if is found that the precondition of a UC and a postcondition of another UC, state similar requirements for a particular variable value. Naturally, it is more difficult to detect this situation as it requires further examination of the descriptions of the respective UCs.

- **Improvement**

UCs that represent portions of a complete behavior must be reformed into UCs that individually offer a complete meaningful service to a system's user. Therefore, for the telecommunications system illustrated above, since carrying out a phone call is the real purpose behind the existence of the "smaller" *base* UCs, the functionality of the sequences UCs should be combined into a single UC called Make A Phone Call.

- **Antipattern Name**

a6. Accessing an *extension* UC - Automation Support: Type (1)

- **Description**

Similar to *base* UCs, *extension* UCs can be initiated by actors. Therefore, modelers may associate an *extension* UC with any actor.

- **Rationale**

Extension UCs differ from regular *base* UCs in that they contain behavior that is of an exceptional or optional nature. Modelers may need an actor to access such behavior contained in an *extension* UC for a number of different reasons:

- (1) If the *extension* UC contains optional behavior relative to the *base* UC, this optional behavior may be exclusively useful to an actor. Therefore, an explicit association is created between the actor and the *extension* UC to allow the actor to execute this optional behavior without needing to initiate the *base* UC

first. This scenario is illustrated using the bookstore system shown below (see Figure 4-14):



Figure 4-14: A good scenario of directly accessing an *extending* UC to allow independent initiation of an optional service

The bookstore may occasionally put on a promotion that entitles a customer to a free book sleeve with every purchase of a book. Hence, when the UC Sell Book is performed to complete a sale transaction, the *extension* UC Give Away Free Bookmark is initiated to carryout the promotional offer. At other times, a bookstore employee may choose to give away a free book sleeve as a courtesy gesture or for advertisement purposes, without a preceding book purchase. Therefore, the Bookstore Employee actor needs to explicitly be associated with the Give Away Free Bookmark UC, to be able to give away free bookmarks without initiating the Sell Book UC.

- (2) When an *extension* UC is handling an exceptional situation, it may be desired to notify a particular actor that such an exceptional situation has occurred. An association is created between the actor and the *extension* UC to allow the *extension* UC to notify the actor of the occurrence of such an exceptional situation. For example, in the Internet Service Provider system shown in Figure 4-15, every time a customer's account is updated, the system automatically checks if there is any balance due on the customer's account. If

there is a balance due, the *extension* UC Notify Customer of Balance Due is initiated. The main purpose behind this *extension* UC is to notify the Customer of the balance due on their account. The *extension* UC can be configured to send an email or a statement letter to the corresponding Customer.

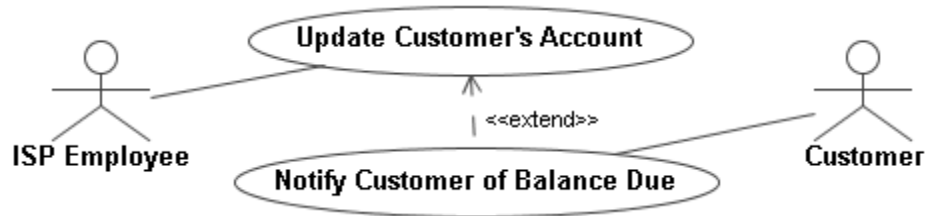


Figure 4-15: The *extension* UC is used to notify the actor of an exceptional situation.

(3) The operation of the *extension* UC may require certain information to be able to operate. If an actor is the source of this required information, modelers create an association between the actor and the *extension* UC to allow that actor to convey this required information. The required information was already provided by the actor when the *base* UC was being performed. This scenario is illustrated in Figure 4-16.



Figure 4-16: The actor is allowed to directly access the *extension* UC since the actor is the source of information required by the *extension* UC.

If a Customer attempts to buy a music video that is not available, the *extension* UC CD Out of Stock is initiated. An association is created between the Customer and the *extension* UC since it is the Customer that knows the

title of the information. The UC records the title of the unavailable CD to notify the store's manager at a later point. The UC also stores the email address of the Customer to notify that customer when the required CD eventually arrives at the store.

(4) When an *extension* UC is initiated, it may be desirable to communicate with an actor to retrieve decisions or other information from that actor with regard to the sequence of actions required to handle the exceptional situation at hand. Therefore, an association is created between the actor and the *extension* UC to form a communication link to allow the decision making process to take place. Unlike situation (3), the required information was not provided by the actor when the *base* UC was being performed. The bakery shop system presented in Figure 4-17 illustrates this scenario.

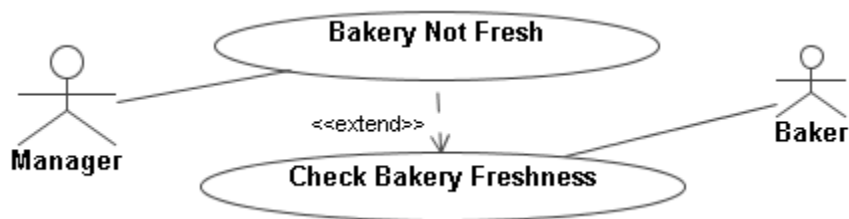


Figure 4-17: The *extending* UC communicates with the actor decide how to deal with an exceptional situation.

If the Baker decides that particular bakery merchandise is not fresh, the *extension* UC Bakery Not Fresh is initiated. The operation of this *extension* UC mainly consists of prompting the Manager for a decision with regard to what to do with the expired bakery merchandise. The Manager may opt to:

- Discard the expired merchandise;
- Give it away to shelters and charity;

- Give it away free to customers as a courtesy gesture; or
- Sell it at half price.

▪ **Consequences**

- (1) The scenario presented by the bookstore system (see Figure 4-14) is appropriate because the *extension* UC contains behavior that is complete and optional. Moreover, it is desirable to execute this optional functionality without initiating the *extended base* UC.
- (2) In the scenario presented by the ISP system shown in Figure 4-15, the association presented between the actor and the *extension* UC allows a bi-directional flow of messages between the two entities. This means that the Customer may initiate the *extension* UC Notify Customer of Balance Due, regardless of an exceptional situation did occurring. Moreover, the Customer may also interfere with the operation of the *extension* UC when it is initiated.
- (3) The scenario presented by the music store system (see Figure 4-16) is inappropriate because the Customer can directly initiate and interfere with the operation of the *extension* UC CD Out of Stock. This is an undesired effect because the Customer may convey a different title to the *extension* UC than was requested by performing the *base* UC.
- (4) In the scenario presented by the bakery shop system shown in Figure 4-17, the Manager is supposed to communicate with the *extension* UC Bakery Not Fresh only if the Baker finds bakery merchandise that is not fresh. However, since the navigability of the association between the Manager and the

extension UC is not specified, the **Manager** may initiate the *extension* UC. This is an undesired effect since the *extension* UC may be performed even if all the bakery merchandise is currently fresh.

- **Detection**

Where – Search for any *extension* UCs in the UC diagram. **How** – If the *extension* UC detected is directly associated with any actor in the model.

OCL Description:

```
context UseCase
```

inv AccessingExtensionUseCaseByActor:

```
not( ( self.extended -> size > 0 or self.extendedUC->size>0) and self.actorEnd->size > 0 )
```

- **Improvement**

- (1) Since this situation is deemed acceptable, no corrective actions required.
- (2) The modelers need to explicitly state that the association between the Customer and the *extension* UC Notify Customer of Balance Due, is a one-way communication link. Unfortunately, UML lacks the required notation to depict this type of association between actors and UCs. To workaround this limitation, a UML note can be connected to the association link between the two entities to explicitly state that this association is not bi-directional (see Figure 4-18). Moreover, the navigation direction of the association link can be specified to ensure that interaction between the two entities is started by the *extension* UC.

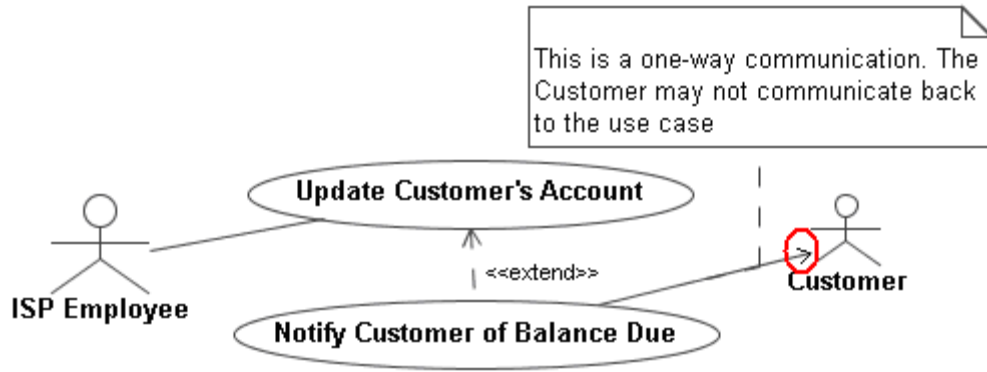


Figure 4-18: Setting the navigation allows the UC to initiate communication with the actor

(3) The *base* UC should be used to convey information to the *extension* UC.

Therefore, the association between the actor and the *extension* UC should be removed. Instead, the *extension* UC CD Out of Stock should retrieve the title of the unavailable CD from the *base* UC that it *extends*. The *base* UC Buy Music CD should have the title of the unavailable CD since the Customer would have provided the title when performing the *base* UC (see Figure 4-19).



Figure 4-19: The *base* UC should be the one conveying the *extension* UC its required information instead of the actor.

(4) The bakery shop system requires the Manager to communicate with the

extension UC Bakery Not Fresh, in order to convey the Manager's decisions. The current association between the two entities satisfies this need.

However, it is also required that *extension* UC initiates the interaction between the two entities. This can be achieved by setting the navigation direction of the association link as shown in Figure 4-20.

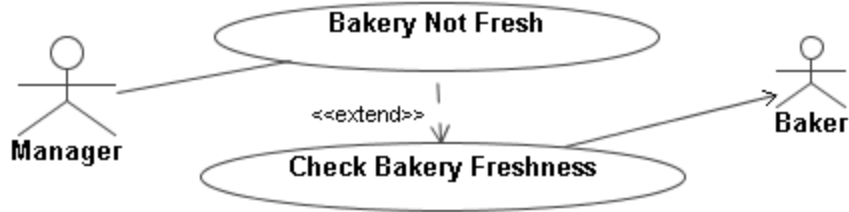


Figure 4-20: Setting the navigation direction to ensure that the actor does not start the interaction with the extension UC.

➤ **Antipattern Name**

a7. Multiple actors associated with one UC - Automation Support: Type (1)

▪ **Description**

A UC is associated with more than one actor.

▪ **Rationale**

The situation described above may occur due to different reasons:

- (1) The actors associated play a similar role when performing the shared UC. In other words, the actors will communicate with the shared UC in a similar fashion. For example, the procedure of performing the UC Perform Transaction in Figure 4-21, is the same when performed by the Manager or Employee actors.



Figure 4-21: Two actors that play a similar role when executing a UC

- (2) The modelers incorrectly depict instances of the system's users instead a class of the system's users. This situation is illustrated in Figure 4-22; actors Adam, Jane and Mary are all students who would like to enroll into the university.

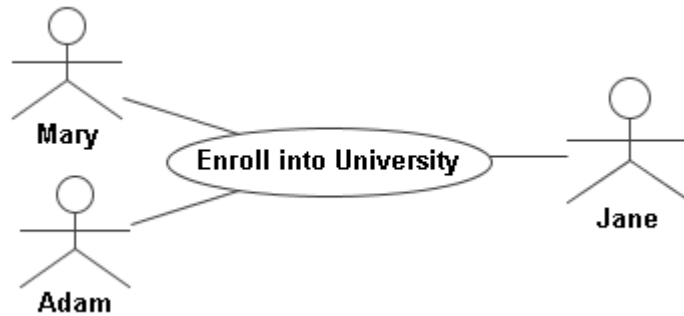


Figure 4-22: A model representing instances of an actor

- (3) The functionalities performed by the shared UC with each actor are very similar in a general sense.
- (4) The proper execution of the shared UC requires communication with two different actors. For example, the Withdraw Cash UC in Figure 4-23, requires the actor ATM Customer to input the amount of money to be withdrawn, meanwhile the actor Bank System verifies that the requested amount is available in that ATM Customer's account.



Figure 4-23: Two actors appropriately associated with a UC

▪ **Consequences**

- (1) Actors should communicate with a UC if they are playing unique roles while a UC is being performed. Therefore, in the scenario shown in Figure 4-21,

- designers will assume that the **Manager** and the **Employee** actor play different roles when executing the **Perform Transaction UC**. Hence, the implementation of the actors with respect to the execution of the UC will be different even though they should be the same.
- (2) The model illustrated in Figure 4-22 violates the true semantics of an actor. This will yield to similar consequences as described above in (1). Moreover, the model will need to be changed frequently as instances of a type of the system's users are frequently created and removed.
 - (3) The actual functionality developed will only cater to one of the actors, or perhaps neither.
 - (4) The model shown in Figure 4-23 is appropriate as both actors **Customer** and **Bank System** have different roles when the shared UC is performed.

- **Detection**

Where –Search for any UCs associated with actors in the UC diagram. **How** – If the UC is associated with more than one actor.

OCL Description:

```
context UseCase
```

```
inv MultipleActorsAssociatedWithUC:
```

```
not (self.actorEnd->size > 1)
```

- **Improvement**

- (1) The scenario illustrated in Figure 4-21 can be fixed by extracting the overlapping roles between the associated actors and creating a new actor that

represents these roles, such as Sales Clerk. The involved actors will *generalize* the newly created actor. This solution is illustrated below in Figure 4-24.

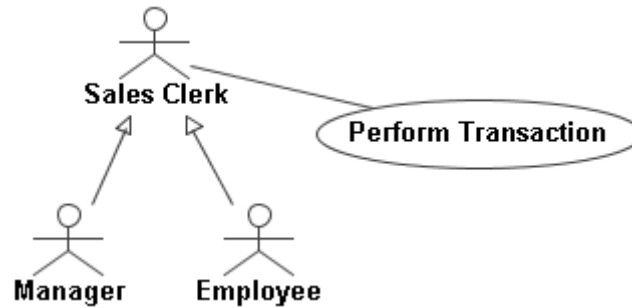


Figure 4-24: The overlapping roles between the two actors should be *generalized* into a separate actor

(2) In the scenario shown in Figure 4-22, actors Adam, Jane and Mary represent the role of a student. Therefore, an actor called Student should be created that will represent all instances of a student. This solution is illustrated below in Figure 4-25.

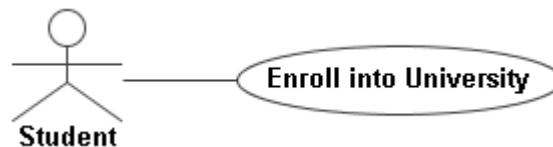


Figure 4-25: The model should represent the role of a class of users not instances of them

(3) The shared UC should be split into separate UCs which accurately represents the behavior of the system when interacting with each actor.

➤ **Antipattern Name**

a8. A description of an actor that is not depicted in the UC diagram - **Automation**

Support: Type (2)

▪ **Description**

The UC model contains a description of an actor; however, the actor is not depicted in the UC diagram.

▪ **Rationale**

- (1) Even though the behavior of the actor might be known, it is not clear at the time how the actor will interact with the system.
- (2) The shared UC should be split into separate UCs that accurately represent the behavior of the system when interacting with each actor.
- (3) The actor described is associated with many UCs. Therefore, depicting the actor and its association with the UCs will clutter the UC diagram.

▪ **Consequences**

- (1) It is acceptable to describe an actor before deciding how it interacts with the system. However, it is essential that the actor's association with the system be eventually defined. If actor remains missing from the UC diagram, the UC diagram will provide a false representation of the actor's involvement with the system.

- (2) Timers and other input/output devices do not constitute actors. This issue is discussed in the description of the “Representing devices as actors” antipattern.
- (3) This situation can be a result of having too many UCs. The “Too many UCs” antipattern “a25.” (see Appendix A) describes the consequences of this situation. If actor remains missing from the UC diagram, the UC diagram will provide a false representation of the actor’s involvement with the system.

- **Detection**

Where – For every actor described. **How** – (a) If the actor is not depicted in the UC diagram.

- **Improvement**

- (1) The association of the actor with the system must be defined and appropriately depicted in the UC diagram.
 - (2) The behavior of these devices should be included in the behavior of the UCs they are associated with. If it is necessary to describe the behavior of a device, such a description should be available in the supplementary requirements document.
 - (3) First, the improvements stated by the “Too many UCs” antipattern should be undertaken. If there remains too many UCs which one actor is associated with, then reorganizing the layout of the UC diagram can be beneficiary. In any event, the actor and its associations with the system must be depicted.
-

➤ **Antipattern Name**

a9. Using *extension/inclusion* UCs to implement an *abstract* UC - **Automation**

Support: Type (1)

▪ **Description**

An actor is directly associated with an *abstract* UC that is not implemented through a *specializing* UC. The implementation of the *abstract* UC is done through *extension* or *inclusion* UCs instead.

▪ **Rationale**

The scenario described above may occur for different reasons:

(1) Modelers find that the *inclusion* UCs contain subroutine behavior. On the other hand, the *extension* UCs contain exceptional or optional behavior. Therefore, the *inclusion* or *extension* UCs do not contain specialized behavior with regard to the *abstract* UC and thus should not be modeled using the *generalization* relationship. Figure 4-26 illustrates an example of this scenario.



Figure 4-26: an *abstract* UC including subroutine behavior and being extended by a UC containing exceptional or optional behavior

(2) *Extension* or *inclusion* UCs represent specialized behavior with respect to an *abstract* UC. For example, in Figure 4-27, the *abstract* UC Make a Trade

can be implemented in the context of making a bonds trade, using the *inclusion* UC Make a Bonds Trade, or a stocks trade, using the UC Make a Stocks Trade.



Figure 4-27: An *abstract* UC including UCs that contain specialized behavior

(3) The model is so far incomplete. At a later point, *specializing* UCs will be added to implement the *abstract* UC.

- **Consequences**

In first two scenarios described above, the *extension/inclusion* UCs are used to directly implement the *abstract* UC. However *extension/inclusion* UCs contain behavior different from the behavior specified in the *abstract* UC. To further elaborate, the behavior contained in the *extension/inclusion* UCs does not realize the behavior described in the *abstract* UC. Therefore, when the actor initiates a service request, the behavior specified by the *abstract* UC will never be performed since it is never realized by any UCs. Only *specializing* UCs may implement *abstract* UCs.

- **Detection**

Where – Search for any *abstract* UCs. **How** – (a) the *abstract* UC is associated with an actor, and (b) the *abstract* UC is *extended* by other UCs or *including* other UCs, and (c) the *abstract* UC does not have child UCs.

OCL Description:

```
context UseCase
```

```
inv UsingIncludeAndExtendToImplementAbstractUC:
```

```
not((self.isAbstract) and (self.inclusion->size > 0 or  
self.extension->size > 0 or self.extensionUC->size>0))
```

▪ **Improvement**

(1) In the scenario illustrated in Figure 4-26, the type of relationships between the *abstract* UC and the other UCs Oil System Damaged and Check Oil Level is appropriate and should remain unchanged. Unless the model is incomplete, the *abstract* UC Perform Oil Maintenance should be set as *concrete* as shown in Figure 4-28.

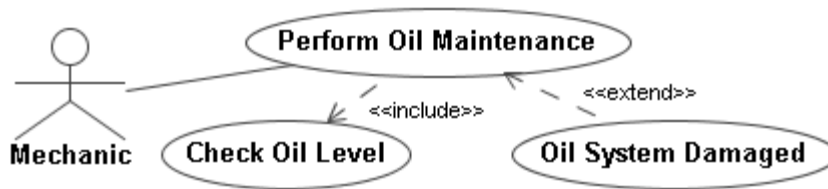


Figure 4-28: The *abstract* UC now set to be *concrete*

(2) In the scenario illustrated in Figure 4-27, the usage of the *include* and *extend* relationships is incorrect because the *extending/included* UCs Make a Bonds Trade and Make a Stocks Trade represent specialized behavior with respect to the *abstract* UC Make a Trade. In this case, the *specialization* relationship is considered to be the appropriate relationship between the UCs, and therefore this model can be fixed as shown in Figure 4-29.



Figure 4-29: The *abstract* UC is associated with its *specializing* UCs using the *generalization* relationship

(3) The modelers should review and consider adding the missing *specializing* UC(s) whenever possible.

➤ **Antipattern Name**

a10. Multiple generalizations of a UC - **Automation Support: Type (1)**

▪ **Description**

A single UC *specializing* two or more UCs.

▪ **Rationale**

Modelers extract common behavior between two or more UCs and create a new *specializing* UC that will contain this common behavior. For example, preparing either a cargo or a passenger aircraft for a trip requires the cleaning of the aircraft. As shown in Figure 4-30, the common behavior is contained in the UC Clean Aircraft, which *specializes* the UCs Prepare Passenger Aircraft For Trip and Prepare Cargo Aircraft For Trip.



Figure 4-30: Multiple *generalizations* of one UC

- **Consequences**

The behavioral semantics of the model is violated. The UC Clean Aircraft is not a specialized version of the Prepare Passenger Aircraft For Trip and the Prepare Cargo Aircraft For Trip UCs, which may lead to an incorrect implementation of the system.

- **Detection**

Where - Search for a *specializing* UC. **How** – If that UC is *specializing* more than one UC.

OCL Description:

```
context UseCase
```

```
inv MultipleGeneralizationsOfOneUC:
```

```
not (self.parent->size > 1)
```

- **Improvement**

The shared UC Clean Aircraft contains subroutine behavior required by the two other UCs. Therefore, the *specialization* relationship should be replaced with an *include* relationship. The *include* relationship is considered more appropriate

since the shared UC contains common behavior not specializing behavior. This solution is illustrated below in Figure 4-31.

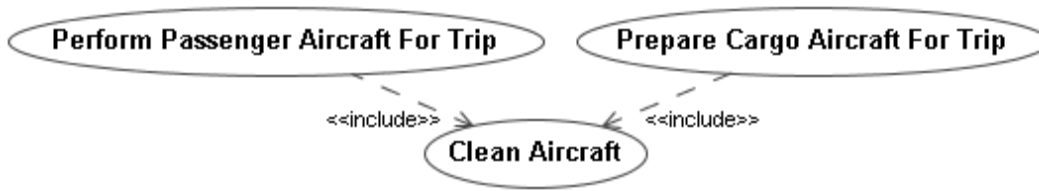


Figure 4-31: The *generalized* UC should be *included* by the other UCs that need it

➤ **Antipattern Name**

a11. Duplicating functionalities for the *generalized* and *specializing* UCs -

Automation Support: Type (1)

▪ **Description**

The relationships that a *generalized* UC has with other UCs are duplicated for the *specializing* UC. As shown in Figure 4-32, the UC Authenticate User is *included* by both the Perform Transaction and Withdraw Cash UCs. Moreover, the UC Insufficient Funds is an *extension* UC for both the Perform Transaction and Withdraw Cash UCs.

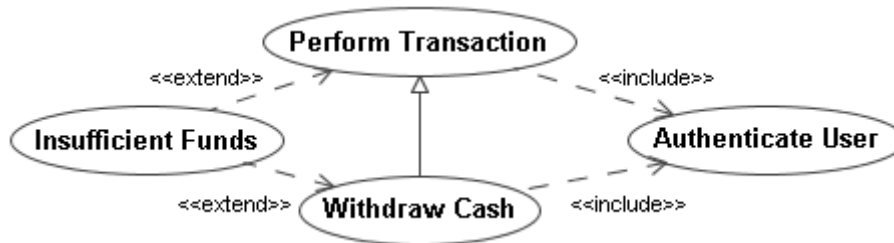


Figure 4-32: Duplicating functionalities for the *generalized* and *specialized* UCs

- **Rationale**

Modelers try to establish that the behavior contained in the *inclusion* and *extension* UCs are applicable to both the *generalized* and the *specializing* UCs.

- **Consequences**

This situation will probably lead to the creation of duplicated or redundant code at the implementation phase. This redundant code will be the implementation of the *inclusion* and the *extension* UCs.

- **Detection**

Where - Search for a *generalization* relationship between two UCs in the UC diagram. **How** – If both the *generalized* and *specializing* UCs have similar relationships with other UCs in the model.

OCL Description:

```
context UseCase
```

inv DupFuncAtChildAndParentUCUsingInclude:

```
not ( UseCase.allInstances->forAll
(u1 , u2 | ((self <> u2) and (u1 <> u2) and (self <>u1
))implies (self.inclusion->includes(u2) and
u1.inclusion-> includes(u2))))
```

inv DupFuncAtChildAndParentUCUsingExtend:

```
not ( UseCase.allInstances->forAll
(u1 , u2 | ((self <> u2) and (u1 <> u2) and (self <>u1
))implies ((self.extended->includes(u2) or
```

```
self.extendedUC-> includes(u2)) and(u1.extended->
includes(u2)or self.extendedUC->includes(u2))
```

- **Improvement**

The modelers need to determine whether a given *included* or *extending* UC is applicable to all of the *specializing* UCs or only a subset of them. For example, the Authenticate User UC must be executed to allow for any transaction to be performed. Therefore, the Authenticate User UC is applicable to all of the *specializing* UCs and thus there should be one *include* relationship in the model between the Authenticate User and the Perform Transaction UCs as shown in Figure A1-32. On the other hand, the *extension* UC Insufficient Funds describes exceptional behavior responsible of handling a situation where a user requested to withdraw a cash amount that is larger than what is available in the user's account. Hence, UC Insufficient Funds is only applicable to the Withdraw Cash UC, and therefore there should only be one *extend* relationship between the Withdraw Cash UC and the Insufficient Funds UC as shown in Figure 4-33.

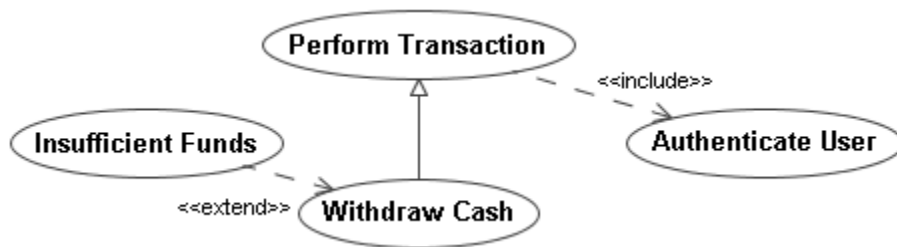


Figure 4-33: Only the appropriate *include/extend* relationships remain

➤ **Antipattern Name**

a12. Accessing an *abstract* UC that is not implemented. - **Automation**

Support: Type (1)

▪ **Description**

An actor is directly associated with an *abstract* UC that is not implemented using *specializing* UC(s) as shown in Figure 4-34.

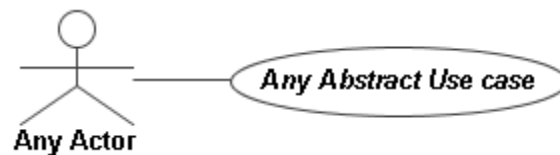


Figure 4-34: An actor directly association with an unimplemented *abstract* UC

▪ **Rationale**

- (1) This situation is most likely to occur when the model is incomplete. The *abstract* UC will be implemented by *specializing* UCs in a later phase.
- (2) Modelers may incorrectly assume that an *abstract* UC may be initiated to offer a service to the initiating actor.

▪ **Consequences**

- (1) This is an acceptable modeling practice as long as modelers eventually insert the missing *specializing* UC(s) that will implement the *abstract* UC.
- (2) *Abstract* UCs cannot be initiated and hence no behavior will be performed.

- **Detection**

Where - Search for an *abstract* UC in the “UC Diagram”. **How** – If that *abstract* UC is associated with an actor and is not *specialized* by at least one UC. If the *abstracted* UC is *including* other UCs or being *extended* by other UCs then review the “Using *extension/inclusion* UCs to implement an *abstract* UC” antipattern.

OCL Description:

```
context UseCase
```

inv AccessingUnimplementedAbstractUC:

```
not ((self.actorEnd->size > 0) and self.child->size =  
0) and self.isAbstract))
```

- **Improvement**

- (1) Modelers should review and consider adding the missing *specializing* UC(s) whenever possible.
 - (2) The *abstract* UC should be set as *concrete*. This will enable that UC to be initiated by the actor.
-

➤ **Antipattern Name**

a13. UC initiated by two actors - Automation Support: Type (1)

▪ **Description**

Two actors are associated with one UC. The associations point towards the UC meaning that the UC was initiated twice. As shown in Figure 4-35, UC *Withdraw Funds* is initiated by the actors *Customer* and *Bank System*.



Figure 4-35: Two actors initiating one UC

▪ **Rationale**

- (1) This situation usually occurs because association links are mistaken for information flow links. When the two actors are providing the UC with information, then the associations with the UC are directed towards that UC. In the ATM system shown in Figure 4-35, the *Customer* actor provides the PIN for the UC, meanwhile the *Bank System* actor provides the UC with this *Customer's* current balance.
- (2) Both actors play a similar role when the UC is being performed.

▪ **Consequences**

- (1) It is not possible to determine which actor initiates the UC. Usually the primary actor is the one that initiates a UC since the primary actor is the

- primary beneficiary of the service provided by the UC. Therefore, it will be not be intuitive to determine the primary actor.
- (2) Unless extreme care was taken during the authoring of the UC description. Designers will not be able to determine the correct sequence of interactions between the actors and the UC.
 - (3) See the “A UC is associated with more than one actor” antipattern.

- **Detection**

Where - Search for a directed association relationship in the UC diagram that is pointing towards a UC. **How** – If the UC has another association relationship pointing towards it.

OCL Description:

```
context UseCase
inv UseCaseInitiatedBy2Actors:
not ( self.directedActorEnd->size > 1 )
```

- **Improvement**

- (1) The sequence of interactions between the actors and the UC should be reconsidered. Upon determining which actor is the one responsible for initiating the UC, all other association relationship links connected with the UC should be set to be bi-directional or directed towards the non-initiating actors instead. For the ATM system, it is the Customer that initiates the Withdraw Funds UC and the interaction with the Bank System is always initiated by the UC to determine the Customer’s current balance. Therefore,

the association relationship with the Bank System should be directed towards the Bank System, as shown in Figure 4-36 below.



Figure 4-36: Only one actor should initiate a UC

(2) See the “A UC is associated with more than one actor” antipattern.

➤ **Antipattern Name**

a14. Two actors with the same name - Automation Support: Type (1)

▪ **Description**

Two actors existing within a UC diagram with identical names. For example, the system shown in Figure 4-37 has two sets of UCs. Each set of UCs carries out a certain category of administrative duties. The UC sets are associated with two actors named Administrator.

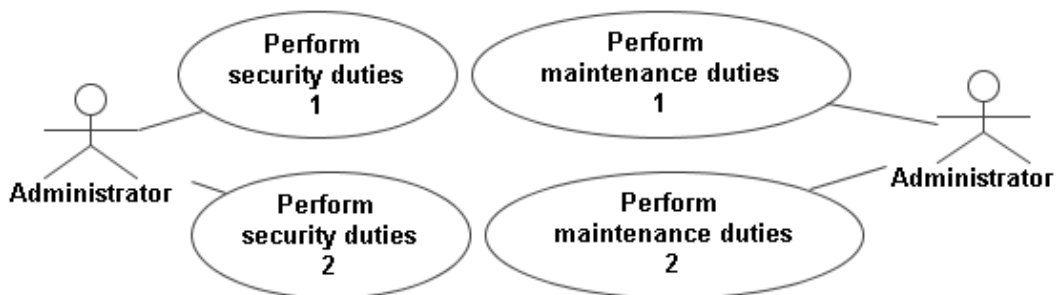


Figure 4-37: Two different actors with the same name

- **Rationale**

- (1) This situation may occur if the actors' roles are carried out by different personnel with similar job titles.
- (2) To enhance the layout of the UC diagram. Using several instances of the same actors may prevent the cluttering of the diagram.
- (3) Each instance of the actor is associated with a set of UCs that represents a certain category of services.

- **Consequences**

For all three cases presented above, using two actors with similar names may cause confusion as to whether the actors are actually similar or is there any subtle differences between them. Designers may account for two distinct actors in their design while the actors were actually the same entities and vice versa, leading to redundant or incorrect implementation respectively.

- **Detection**

Where – For every actor in the UC diagram. **How** – If there exists another actor with an identical name.

OCL Description:

```
context Actor
```

```
inv TwoActorsWithSameName:
```

```
not ( a,b | a<>b and a.name = b.name )
```


- **Improvement**

- (1) The actors should be given names that further distinguishes between them and represents their duties more accurately (see Figure 4-38).

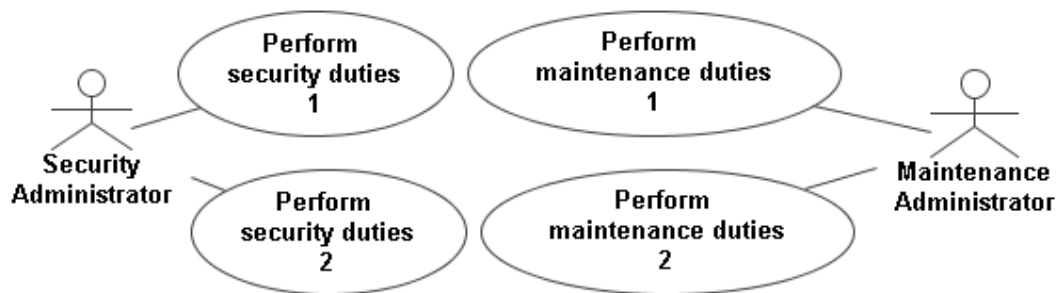


Figure 4-38: The actors are given more distinguishing names

- (2) The consequences of depicting the same actor twice outweigh the layout benefits that can be achieved by doing so. The layout of the UC diagram can be improved by reconsidering the positioning of each diagrammatic entity without the need to depict the same actor twice. However, if the diagram's layout and readability will radically improve, then this modeling practice can be warranted. In this situation, the actors should be annotated with UML notes to explicitly state that they are the same.
- (3) Each set of UCs associated with one instance of the given actor represents a separate subsystem. Subsystems can be considered as separate UC diagrams which warrants the depiction of an actor present in a different UC diagram (see Figure 4-39).

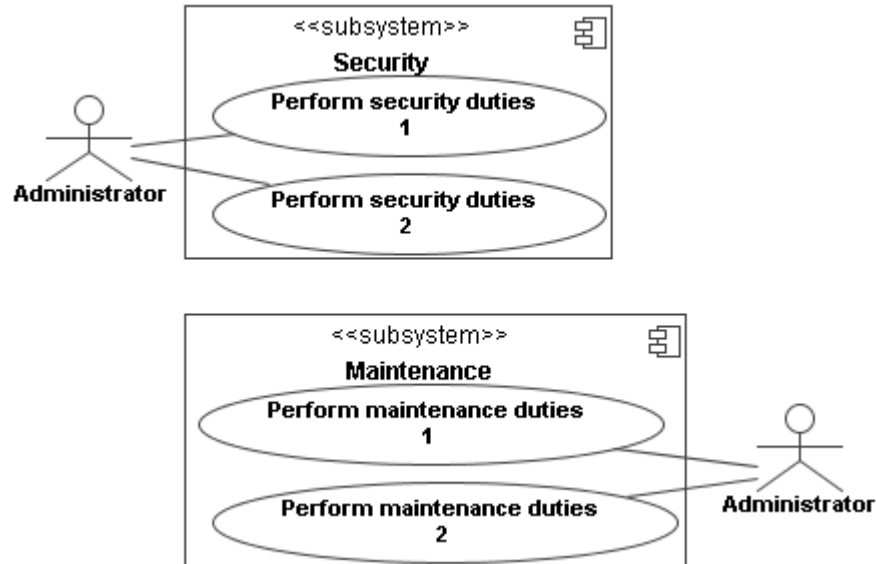


Figure 4-39: Two subsystems are presented as two UC diagrams, both containing the name actor

➤ **Antipattern Name**

a15. An actor inside the system boundary - **Automation Support: Type (1)**

▪ **Description**

An actor is depicted within the system boundary.

▪ **Rationale**

- (1) The actor represents the system itself and is depicted inside the system boundary for labeling purposes.
- (2) The actor represents an internal device which the system depends upon such as a timer.

(3) The actor plays the role of the secondary actor for all UCs, whereby that actor only aids in providing a service to the primary actors but is never the beneficiary actor.

▪ **Consequences**

(1) This situation may cause confusion since the system itself is depicted as an actor. Actors are external entities to a system, therefore depicting the system as an actor may lead the model reader to believe that the system itself is an external entity. Therefore, it would seem that the actual system represented by the UC model does have a name.

(2) Timers and other input and output devices are not systems. Such devices only facilitate the input and output of information. However, if it is necessary to represent a device as an actor, the actor should remain outside the system boundary. Leaving the actor inside the system boundary may lead designers to believe that the device is a part of the system which needs to be designed and implemented. The unnecessary additional effort spent designing such entities (actors) causes the system to be more complex and often causes the project to be behind schedule and over budget.

(3) Even if the given actor is always a secondary actor, the actor should remain outside the system boundary. The consequences of leaving the actor inside the system boundary are similar to that described previously in (2).

- **Detection**

Where – Within the system boundary in the UC diagram. **How** – If an actor can be detected inside the system boundary.

- **Improvement**

- (1) The system boundary should be labeled with the system's name only and without using an actor icon. The actor icon used to represent the system itself should be removed.
 - (2) If the device is merely an input/output device, then the device should not be represented as an actor. If it is necessary represent the device as an actor, the actor should be located outside the system boundary.
 - (3) Secondary actors as any other type of actors should be located outside the system boundary.
-

- **Antipattern Name**

a16. An unassociated UC - Automation Support: Type (1)

- **Description**

A UC is depicted in the UC diagram that is not associated with any other entity. This means that the UC does not have an association relationship with the any actor and does not have any *include*, *extend* or *generalization* relationship with any other UC.

- **Rationale**

- (1) The UC represents an internal functionality that the system needs to perform.
- (2) The association relationship notation between the UC and an actor in the UC diagram, or another entity such as a package or a subsystem, is missing.
- (3) The *include*, *extend* or *generalization* relationship notation between the UC and other UCs is missing.

- **Consequences**

- (1) The purpose of UC modeling is to show the interactions that occur between a system and its external entities in order to provide a service to an actor. Internal functionalities do not offer an immediate service to actors.
- (2) This situation may lead to a faulty design since the involved actors are not accounted for during the design.
- (3) This situation may lead to unnecessary additional design effort since the behavior of the UC is dependent on the behavior of other UCs which have already been accounted for in the design process.

- **Detection**

Where – For every UC in the UC diagram. **How** – If the UC does not have an association relationship with the any actor and does not have any *include*, *extend* or *generalization* relationship with any other UC.

OCL Description:

```
context UseCase
```

inv UnassociatedUC:

```
not( self.actorEnd -> size + or self.directedActorEnd  
-> size = 0)
```

▪ **Improvement**

- (1) UCs representing internal functionalities that do not provide any service to any actor should be removed.
 - (2) Association relationships should be depicted between the UCs and any actors involved with it.
 - (3) The correct type of relationship should be depicted between the UCs and any other UCs that are involved with it.
-

➤ **Antipattern Name**

a17. A UC without a description - Automation Support: Type (2)

▪ **Description**

A UC is depicted in the UC diagram which does not have a corresponding textual description.

- **Rationale**

(1) The UC is too simple and intuitive that it does not require the extra effort of describing it.

(2) The authoring of the UC description was postponed to a later phase. This situation usually occurs when the given UC is “unstable”. An “unstable” UC is one that represents behavior that changes frequently, or one that has a high probability of being removed from the system. The behavior of a UC may change frequently if it represents a functional requirement that was not completely and precisely defined.

- **Consequences**

For both situations described above, skipping the UC authoring process will lead to assumptions about the UC’s behavior. For complex systems, assumptions are often incorrect or inaccurate leading to a system that does not satisfy its requirements. The UCs within the system will also be unclear. This means that it is unclear how the given UC will be associated with other UCs and actors.

- **Detection**

Where – For every UC in the UC diagram. **How** – If there does not exist a corresponding UC description.

- **Improvement**

- (1) Even though a UC maybe simple and its behavior maybe fairly common and well understood, a corresponding UC description must be written to explicitly describe its behavior. This removes any ambiguity and prevents the introduction of inaccurate and incorrect assumptions.
 - (2) Upon detecting a UC that is not described, a UC description must be authored immediately. If the given UC is considered “unstable”, at least a simple outline of its intended behavior should be written.
-

➤ **Antipattern Name**

a18. A described UC that is not depicted in the UC diagram - **Automation**

Support: Type (2)

▪ **Description**

A UC is described but is not depicted in the UC diagram.

▪ **Rationale**

For any given domain, there are common functionalities (UCs) that are expected to exist. At times, it is not easy to determine how such UCs will integrate with the rest of the system.

▪ **Consequences**

- (1) The purpose of a UC diagram is to provide a visual summary of the system’s functional requirements and its environment. Examining the UC diagram is

sufficient to gain an overview of the services that the system offers. If a UC is not depicted in the UC diagram, stakeholders who were not directly involved in creating the UC model may believe that the service is not offered by the system.

(2) It will be unclear how the given UC will be associated with other UCs and actors. Designers will be forced to make assumptions about the UC's role within the system. For complex systems, such assumptions are likely to be incorrect or inaccurate, causing the production of a faulty system.

- **Detection**

Where – For every UC described. **How** – If the UC is not depicted in the UC diagram.

- **Improvement**

The given UC must be depicted in the UC diagram. Moreover, its relationships cases must be considered and appropriately depicted in the diagram.

- **Antipattern Name**

a19. Representing devices as actors - **Automation Support: Type (1)**

- **Description**

Devices such as printers, keyboards, scanners...etc, are represented in the UC diagram as actors. This antipattern is a specialized version of the “An actor inside the system boundary” antipattern and should be searched for after searching for the “An actor inside the system boundary” antipattern.

- **Rationale**

(1) An input/output device is believed to be actor since it is the actual means of I/O into and out of the system. As shown in the Figure 4-40, the Enter billing information UC is associated with the actor Keyboard.



Figure 4-40: A Keyboard actor is used to enter billing information

(2) Often the fundamental goal of a UC is to utilize a given input/output device. For example, a UC that is responsible for printing statements to customers is associated with a Printer actor. This association stems from the fact that main goal of the UC is to produce hardcopies of statements (see Figure 4-41). Therefore, a printer device will always be required to perform this UC.

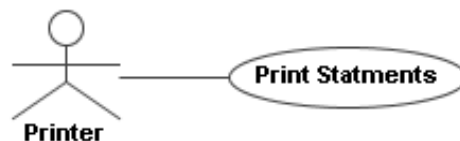


Figure 4-41: A Printer actor is required to print statements.

- **Consequences**

(1) Input and output devices are not systems or actors. They only facilitate the input and out of data. Input devices are not the source of information; and output devices are not the actual beneficiary of the information produced. Therefore, the true source of information provided to the system, and the true beneficiary of the information produced by the system, is unclear.

(2) Even though a particular device is required to perform the objective of a UC as depicted in Figure 4-41, the UC model should not detail these devices. The system should only be responsible for producing and accepting required information, regardless of the mechanism used to achieve these purposes. This decreases the flexibility and modularity of the system and adds unnecessary constraints in the design, as the system is internally designed to deal with particular devices. For example, if the system was required to send statements to customers via email instead of mailing hardcopies to them, then significant changes are required to accommodate the new requirement.

- **Detection**

Where – For every actor present in the UC diagram. **How** – If the actor’s name resembles an input/output device.

OCL Description:

In order to compose an OCL description of this antipattern, the devices’ names must be entered into OCL as a predicate.

```
context Actor
```

```
inv ActorsAsDevices
```

```
not ( self.name = <device name1> or self.name =  
<device name2> or...)
```

- **Improvement**

(1) The actual source of information should be the actor instead of the input device. Similarly, for generated information, the beneficiary of the information should be the actor instead of the output device. Therefore, in Figure 4-40, the billing information required by the Enter billing information UC was provided by the Customer. A keyboard was only used to facilitate the input of the billing information (see Figure 4-42).



Figure 4-42: The Keyboard actor is replaced with the actual actor Customer.

(2) Input/output devices should not be considered in the UC model. This requires that input/output devices should not be depicted as actors in the UC diagram. Input/output devices should only be considered in the description of the UCs

➤ **Antipattern Name**

a20. Very large alternative flows - **Automation Support: Type (3)**

- **Description**

The description of a UC contains a very large alternative flow. The alternative flow spans several pages and describes very complex behavior.

- **Rationale**

Firstly, modelers may not realize that *extension* UCs can be used to describe very large and complex alternative flows. Secondly, modelers may want group all the information related to the operation of one UC inside that UC. As an alternative, modelers resort to writing large alternative flows.

- **Consequences**

Alternative flows are used to describe small deviations from the basic nominal flow of a UC. A very complex or large alternative flow may obscure the real purpose of the UC. Such alternative flows significantly reduce the readability of the UC.

- **Detection**

Where – For every UC described. **How** – If the UC is not depicted in the UC diagram.

- **Improvement**

Extension UCs can be used to describe large and complex alternative flows. This in turn allows the original (which then would be a *base*) UC to be more readable and its main purpose to be clearer.

➤ **Antipattern Name**

a21. Using the term “actor” in textual descriptions - **Automation Support:**

Type (3)

▪ **Description**

The description of a UC makes a reference to one of its involved actors by using the term “actor”.

▪ **Rationale**

(1) The UC is associated with one actor. Therefore, the “actor” in this case is known since there is only one.

(2) Several actors involved with a UC play the same role. Therefore, it is redundant to state each actor by name. It is easier to use the term “actor” to refer to any of the involved actors.

▪ **Consequences**

(1) Using the term “actor” reduces the clarity of the UC description as supposed to using the name of the actor. A larger issue can also occur as the UC model evolves, if at a later stage another actor was associated with the UC, then the term “actor” will become ambiguous.

(2) This situation is a result of another antipattern match. The consequences of this situation are described in the “A UC is associated with more than one actor” antipattern. More specifically, this situation is similar to the first

situation (1) stated in the “A UC is associated with more than one actor” antipattern.

- **Detection**

Where – In the description of every UC. **How** – If the term “the actor” was used anywhere throughout the description of a UC.

- **Improvement**

(1) The term “actor” should be replaced with the actor’s real name.

(2) This situation can be corrected by addressing the originating “A UC is associated with more than one actor” antipattern match. The appropriate actor names should then be used in the UC description.

- **Antipattern Name**

a22. Using incorrect stereotypes - **Automation Support: Type (2)**

- **Description**

A relationship is depicted in the UC diagram that is annotated with an incorrect stereotype. For example, a *generalization* relationship link that is annotated with the stereotype <<include>>, or an *extend* relationship that is annotated with the stereotype <<uses>>.

- **Rationale**

This type of mistake only occurs due to a lack of understanding of the underlying semantics of the types of relationships available in UC modeling.

- **Consequences**

This will lead to a great deal of confusion and misinterpretation with regard to the relationships that exist between a model's entities, which in turn could lead to a faulty design of the system.

- **Detection**

Where – For every relationship depicted in the UC diagram. **How** – (a) If the relationship link between two entities is annotated with an incorrect stereotype.

- **Improvement**

It is recommended to first review the textual descriptions of the two given entities to determine the true type of relationship that exist between them. The relationship link should be corrected if necessary and annotated with the correct stereotype.

- **Antipattern Name**

a23. An association between two actors - **Automation Support: Type (1)**

- **Description**

Two actors in the UC diagram are associated with an association relationship.

- **Rationale**

- (1) The actors need to communicate and exchange information in order to perform one or more UCs.
- (2) One actor is a specialization of the other.

- **Consequences**

- (1) A system needs only to account for the interactions between itself and its actors. The design and implementation of the system will be based solely on these interactions. Accounting for communications between actors or other external systems will add unnecessary complexity to the design. Moreover, assumptions made with regard to the interactions occurring between external entities can be incorrect or inaccurate.
- (2) A generalization relationship between two actors shows the hierarchical relativity between the roles played by these actors. On the other hand, an association between two actors shows that the actors are communicating. Misrepresenting a generalization relationship as an association relationship will lead to similar consequences to those described in (1).

- **Detection**

Where – For every actor in the UC diagram. **How** – If an actor is associated with another actor by an association relationship.

OCL Description:

An association relationship between actors is a fairly uncommon practice. The original metamodel will need to be extended to support this notation. The extension will be in the form of an association relationship, named `Associated_With_Actor`, stemming from the `Actor` class (see Figure 4-43). This extension was not made part of the original metamodel since it is fairly uncommon and hence it does not warrant the additional complexity.

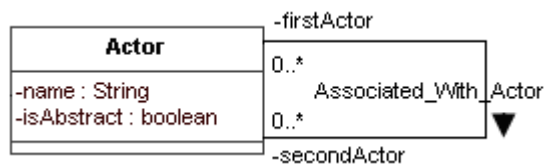


Figure 4-43: Extending the metamodel to support actor to actor associations

```
context Actor
```

inv AssociationBetween2Actors

```
not ( self.firstActor -> size + self.secondActor -  
>size > 0)
```

▪ **Improvement**

- (1) The association relationship should be removed. This allows designers to focus only on the interactions between the actors and the system. Designers should not worry about interactions that occur outside the system.
- (2) The association relationship should be removed and replaced with a *generalization* relationship to reflect the actual relationship between the actors.

➤ **Antipattern Name**

a24. An association between UCs - **Automation Support: Type (1)**

▪ **Description**

Modelers require two UCs to communicate in order to carryout a meaningful and complete service. The two UCs involved are then linked to each other using an *association* relationship.

▪ **Rationale**

The scenario described above may occur due to different reasons:

- (1) Each UC needs to provide information in order to carryout the required service for the user. This scenario is shown in Figure 4-44. The UC Count Shaft Rotations in Trip is responsible to calculate the distance traveled during a given trip. Meanwhile, the UC Measure Time of Trip keeps track of the time elapsed during that trip. The UCs Count Shaft Rotations in Trip and Measure Time of Trip each provide necessary information in order to calculate the average speed of the trip.



Figure 4-44: Two UCs trading information to provide a service

- (2) One of the UCs need to initiate the other UC in order to carry out a subroutine. This scenario is shown in Figure 4-45. The UC Calculate Average Trip Speed initiates the UCs Count Shaft Rotations in Trip and Measure Time of Trip in order to retrieve information necessary to calculate the average speed of the current trip.

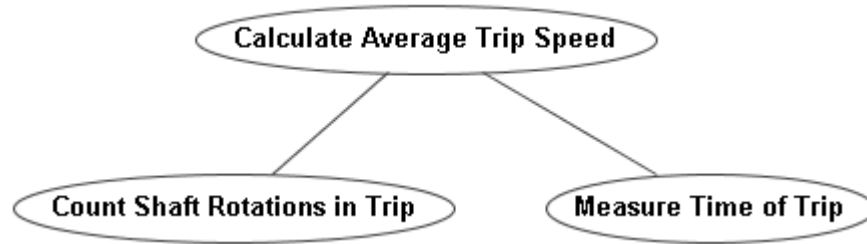


Figure 4-45: A main UC is communicating with other UCs by “calling” them to retrieve necessary information

(3) One of the UCs contains exceptional or optional behavior relative to the other UC. This scenario is shown in Figure 4-46. The UC Trip Has Not Started is responsible for handling the error resulting from attempting to calculate the average trip speed (when UC Calculate Average Trip Speed is being performed) before a trip has even started. The error is caused since the distance and time elapsed are both 0.



Figure 4-46: A UC communicating with the other to provide exceptional behavior

- **Consequences**

(1) In the scenario presented in Figure 4-44, the UC model is used as a design tool instead of an analysis tool. The modelers used the UC model to show internal implementation decisions. This obscures the real functionality and services offered by the system to its users, leading to confusion between various stakeholders. Moreover, descriptions of the UCs will include instructions about the two UCs communicating with each other. Therefore, at the implementation phase, every time the average trip speed is calculated, the elapsed will be unnecessarily measured as well, and vice versa.

- (2) In the scenario presented in Figure 4-45, the true semantics of the relationship between the UCs are violated. The initiated UCs **Count Shaft Rotations in Trip** and **Measure Time of Trip** contain subroutine behavior and thus do not need to know about the internal behavior of their initiating UC. However, an association relationship between two UCs requires the UCs to be aware of each other's behavior. Therefore, the initiated UCs will require additional descriptions that will allow them be aware of the initiating UC.
- (3) In the scenario presented in Figure 4-46, the true semantics of the relationship between the UCs are violated, since the **Trip Has Not Started** UC contains error-handling behavior with respect to the **Calculate Average Trip Speed** UC. The **Calculate Average Trip Speed** UC provides complete behavior individually. Hence, it is not required to be aware of the **Trip Has Not Started** UC. As mentioned before in (2), an association relationship between two UCs requires both UCs to be aware of each other's behavior. Therefore, the **Calculate Average Trip Speed** UC will require unnecessary additional descriptions that will allow it to communicate with the **Trip Has Not Started** UC.

- **Detection**

Where – The Analyst needs to search for any pair of UCs in the UC diagram.

How – If the UCs are connected with each other using an association relationship.

OCL Description:

Similar to the “An association between two actors” antipattern, association relationships between UCs is also a fairly uncommon practice. The metamodel will need to be extended to support the additional notation. The extension will be in the form of an association relationship, named `Associated_With_UseCase`, stemming from the `UseCase` class (see Figure 4-47). This extension was not made part of the original metamodel since it is fairly uncommon and hence it does not warrant the additional complexity.

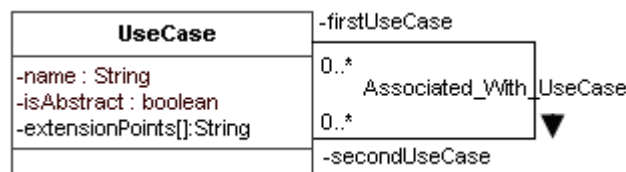


Figure 4-47: Extending the metamodel to support UC to UC associations context UseCase

inv AssociationBetween2UseCases

```

not (self.firstUseCase->size + self.secondUseCase->size > 0)
  
```

▪ **Improvement**

(1) This situation is yet another form of functional decomposition. The UCs shown in Figure 4-44 need to be merged into one UC that will individually calculate the average trip speed. The newly formed UC maybe called Calculate Average Trip Speed. Therefore this scenario can be fixed as shown in Figure 4-48.

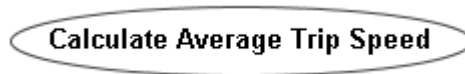


Figure 4-48: Communicating UCs merged into one

However, in the case that the UCs need to remain separated, a new UC that will be responsible of calculating the average trip speed should be created. The newly created UC may then be set to use the existing UCs using the *include* relationship as shown in Figure 4-49 to calculate the average trip speed.

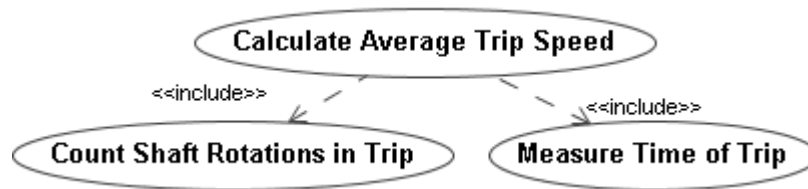


Figure 4-49: Communicating UCs are *included* by a UC that provides a separate complete service

- (2) The initiated UCs in Figure 4-45 Count Shaft Rotations in Trip and Measure Time of Trip contain subroutine behavior required to calculate the average trip speed. These subroutine behaviors should be able to execute regardless of the context they are initiated in. Therefore, the association relationship between the UCs should be replaced with the proper *include* relationship as shown in Figure 4-49.
- (3) The initiating UC in Figure 4-46 Trip Has Not Started contains exceptional behavior with regard to the Calculate Average Trip Speed UC. Moreover, the UC Calculate Average Trip Speed provides complete behavior in the case of no exceptional events occurring. Therefore, the association relationship between the UCs should be replaced with the proper *extend* relationship as shown in Figure 4-50.

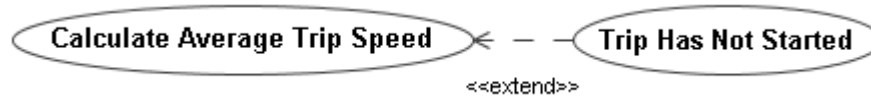


Figure 4-50: The UC containing exceptional behavior now *extends* the *base* UC

➤ **Antipattern Name**

a25. Too many UCs - Automation Support: Type (1)

▪ **Description**

The UC model contains numerous UCs. Detecting this antipattern is dependent on the problem domain. Therefore, identifying how many UCs is too many requires domain expertise and examination of UC models of similar systems. This knowledge will yield an appropriate range for the number of UCs expected. Therefore, this antipattern is matched only if the existing number of UCs far exceeds the appropriate range.

▪ **Rationale**

- (1) The system provides new functionalities and services which incorporates new technologies that were not available in older similar systems. Moreover, the system by nature is extremely complex, providing numerous services to many actors.
- (2) The UCs are designed to be simple and contain very simple behavior for easier implementation. Such UCs usually contain very short flows, and often represents GUI menu commands.

- **Consequences**

- (1) This situation is acceptable since systems evolve and become more complex, offering far more services than before.
- (2) This is another form of functional decomposition. The UCs offer no meaning individually and contain very little substance. The UCs are only useful when combined and sequenced with other UCs.

- **Detection**

Where –The UC diagram. **How** – If the number of UCs far exceeds the expected range.

OCL Description:

```
context UseCase
```

```
inv TooManyUseCases
```

```
not (UseCase -> size > <appropriate size>)
```

- **Improvement**

- (1) No corrective actions are required.
 - (2) UCs that contain very little substance should be reformed into uses that offer a complete meaningful service to a system's user. UCs that contain implementation details, such as GUI menu commands should be removed and replaced with analytical UCs that describe what the system needs to do rather than how it does it.
-

➤ **Antipattern Name**

a26. Depicted actors that do not have a corresponding description -

Automation Support: Type (2)

▪ **Description**

An actor is depicted in the UC diagram; however a textual description for the actor is missing.

▪ **Rationale**

The actor represents an external entity that is fairly well known. A textual description is then considered unnecessary.

▪ **Consequences**

Even though the external entity which an actor represents might be commonly known, a description for that actor is required to describe its capabilities and limitations with respect to interacting with the given system. It is also important to describe the services an actor seeks from the system. If such information is missing, UCs developed at a later phase that are associated with the given actor may falsely presume how the actor will interact with the system.

▪ **Detection**

Where – For every actor depicted in the UC diagram. **How** – If the actor does not have a corresponding textual description.

- **Improvement**

A brief description of the actor, its capabilities, limitations and the services it seeks from the system must be provided. The description does not need to be long. A maximum of one paragraph that contains the required information will suffice.

4.4. Tool Support Using ARBIUM

Examining the structure of a UC diagram is a process that can be fully automated. For complex systems, a UC model may contain hundreds of UCs (Berenbach 2004); in addition, these UCs are not depicted in any chronological order. Moreover, various types of relationships are depicted linking those UCs. Inherently, these relationships are not depicted in any chronological order either. Such systems also usually contain a large number of actors that are associated with UCs using association relationship links. Ultimately the UC diagram can be viewed as a large mesh of UCs, actors and relationship links. Attempting to detect a match for a given diagrammatic structure described by an antipattern can be very challenging, cumbersome and error prone. ARBIUM (**A**utomated **R**isk-**B**ased **I**nspector of **U**C **M**odels) provides automation support for detecting diagrammatic structures. The presented technique does not target deficiencies that can be detected via static analysis, such as syntax errors. ARBIUM is geared towards detecting potential deficiencies that require human validation.

Unsound structures described in antipatterns are entered into ARBIUM as OCL statements. The OCL statements adhere to the simplified metamodel

presented earlier (Figure 4-1) In addition to being able to describe and search for custom made antipatterns, ARBIUM is provided with a set of predefined antipatterns, which analysts may utilize to improve their models. The predefined antipatterns are of the DI variety so that they can be applied to any UC model regardless of its domain.

The matching process is aided by the tool USE (UML-based Specification Environment). USE is a tool that checks the integrity of information systems against constraints described in OCL (Gogolla et al. 2002). ARBIUM generates two input files for USE: a specifications file and a script file. The specifications file describes the class structure of the metamodel, and contains the set of antipatterns specified by the analyst. The script file loads an object representation of the actual UC diagram, based on the simplified metamodel. After completing the matching process, USE presents any antipattern matches for analysts to review. An overview of how ARBIUM, incorporating USE, can be used to search for antipatterns is shown below (see Figure 4-51) A more detailed discussion of ARBIUM is presented in (STEAM 2009c).

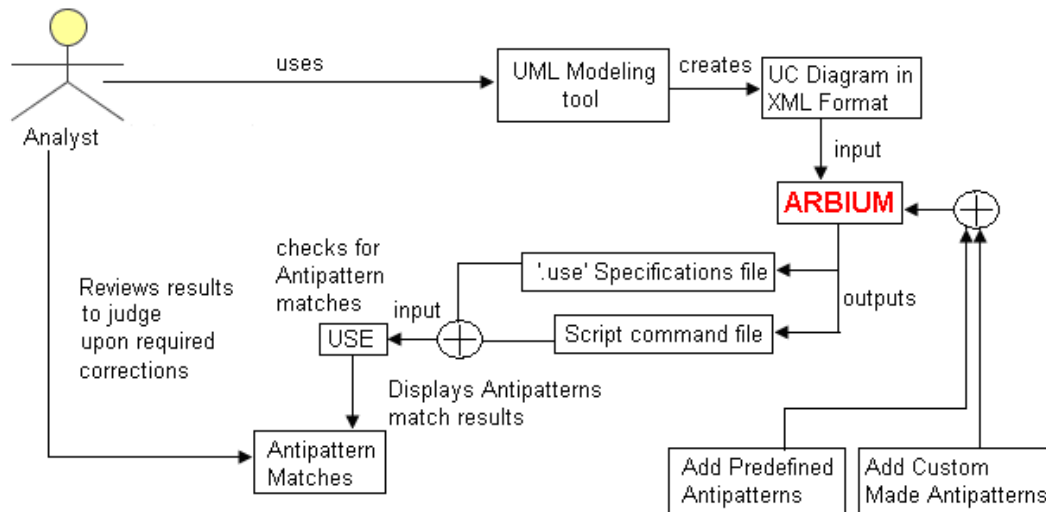


Figure 4-51: An overview of how ARBIUM and USE can automate the detection process

4.5. Evaluation

In this Section we present a real world case study to demonstrate the application of our proposed technique and to examine its feasibility. In addition, we compare the results of using ARBIUM to drive the inspection process to the results of using DesignAdvisor (Berenbach 2004).

4.5.1. Definition and Motivation

The main research question posed by this case study is whether the detection of antipatterns and analysis of the resulting matches can improve the overall quality of UC models. This is achieved on two fronts: (a) by restructuring the UC diagrams to adhere to the notational syntax rules and semantics set by OMG (OMG 2005); and (b) by changing UC descriptions to comply with recommended guidelines and widely accepted practices (Section 4.3.5). Therefore, the

effectiveness of using our proposed approach will be assessed by comparing the resulting UC model with the original UC model, with respect to the aspects mentioned in (a) and (b).

4.5.2. Case Study Formulation

The proposed approach was applied to the MAPSTEDI (**M**ountains and **P**lains **S**patio-**T**emporal **D**atabase **I**nformatics) (MAPSTEDI 2008) UC model. ARBIUM was utilized to perform the matching process. The MAPSTEDI system is being developed to allow the University of Colorado Museum (UCM), Denver Museum of Nature and Science (DMNS), and Denver Botanic Gardens (DBG) to merge their separate collections into one distributed biodiversity database. The merged collections will include over 285,000 biological specimens. The system will also be used as a research toolkit by geocoders to analyze biodiversity data in the southern and central Rocky Mountains and the northern plains both spatially and temporally. The MAPSTEDI system will be developed over three phases. Upon completion of the project, MAPSTEDI will be able to “georeference” the museum collection databases. Users’ search results will be provided by the MAPSTEDI website in GIS-linked spatial-temporal coverage.

The UC model of the MAPSTEDI system contains several UC packages that are used to model different subsystems of the target system. The UC model is accompanied with UC descriptions. The descriptions play an essential role in examining the validity of the UC diagrams and the model as a whole. The

MAPSTEDI UC model contains five UC packages which represent different aspects of the system's functionality. Each UC package contains one UC diagram:

- **Database Access** (Figure 4-52): The purpose of this UC package is to state who may access the database and how. Users of the system can search and download collections data. Users may also visualize biodiversity analysis. Only research users are permitted to access sensitive data.
- **Database Queries** (Figure 4-53): This UC package provides a hierarchal outline of the query functionalities performed by the system. The subsystem queries local and distributed databases for collections data. There are two distributed databases, the DMNS and DIGIR databases.
- **Database Integrator** (Figure 4-54): This UC package shows how the collections data from separate databases (local and remote) are integrated after being updated.
- **Database Edits** (Figure 4-55): This UC package outlines the operational mechanisms for editing and updating the databases. The geocoder edits the collections data and the databases are updated accordingly.
- **Administrative Process** (Figure 4-56): This UC package shows the administrative functionalities and responsibilities. The subsystem backups and restores collections data and application code. The subsystem also installs any new updates.

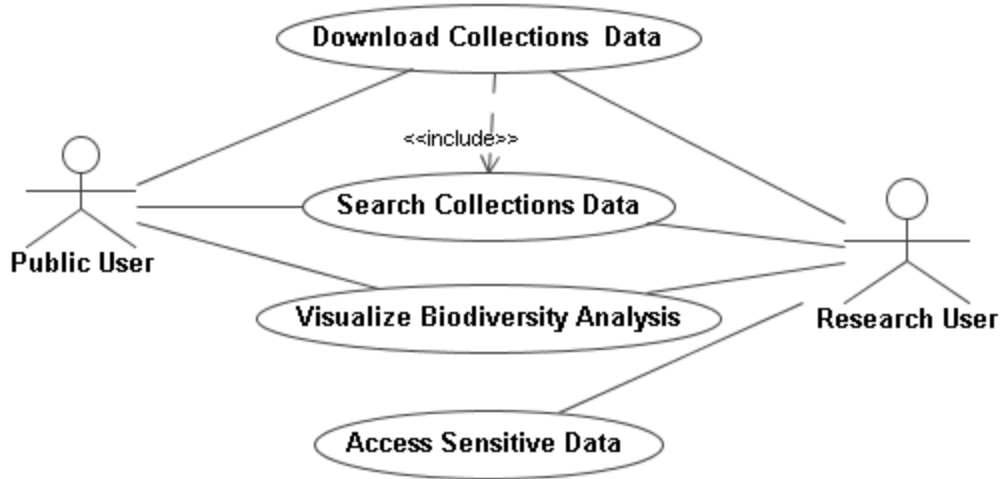


Figure 4-52: The UC diagram of the “Database Access” subsystem

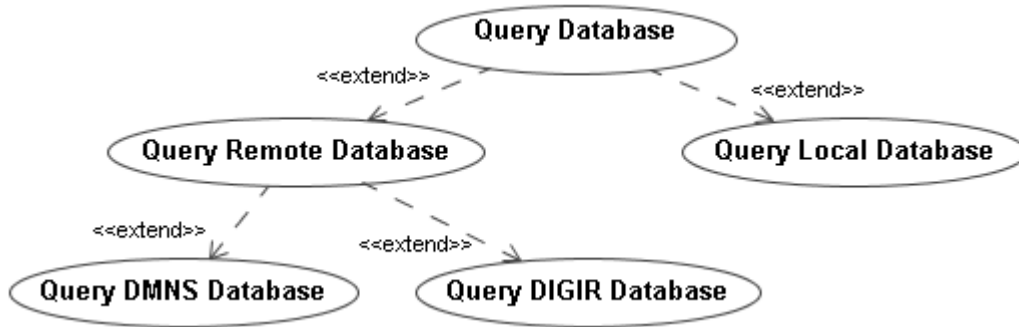


Figure 4-53: The UC diagram of the “Database Queries” subsystem

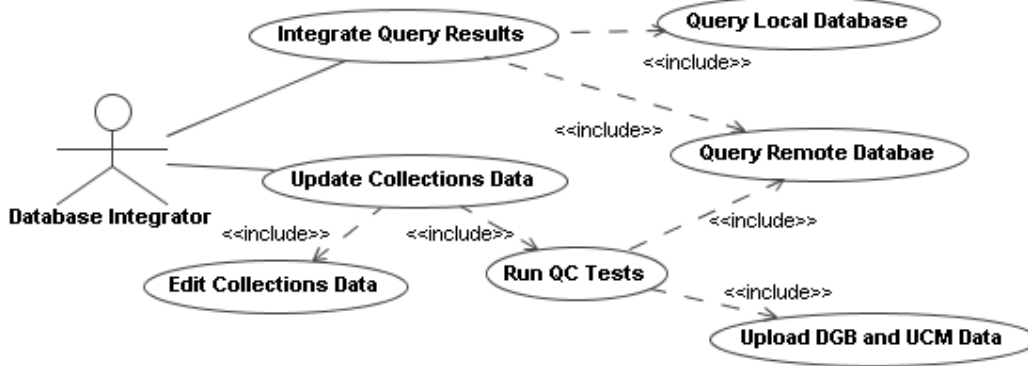


Figure 4-54: The UC diagram of the “Database Integrator” subsystem

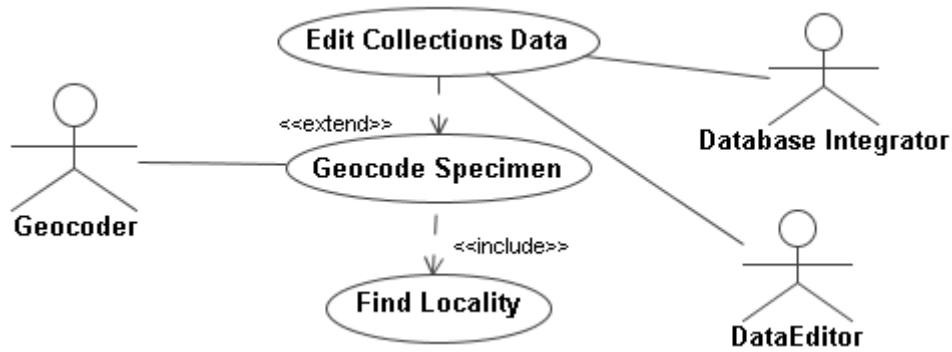


Figure 4-55: The UC diagram of the “Database Edits” subsystem

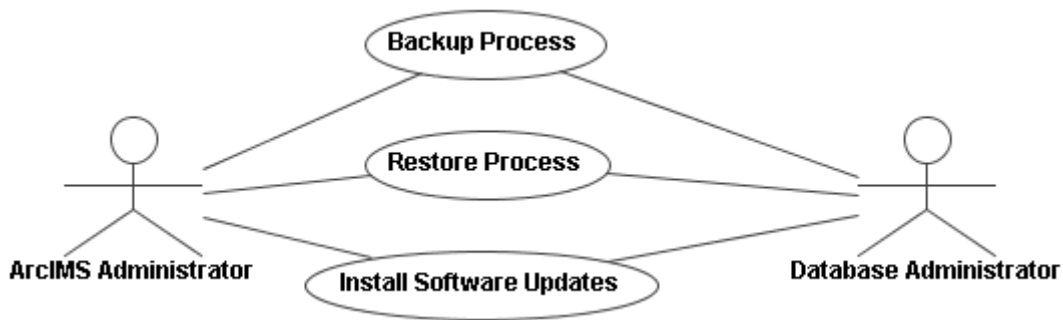


Figure 4-56: The UC diagram of the “Administrative Process” subsystem

Currently the MAPSTEDI UC model suffers from a number of issues (listed below) that decrease its quality. These issues are determined after examining the UC diagrams and the corresponding UC and actor descriptions:

1. The public and research users are shown to have different roles when accessing certain functionalities offered by the system, however they perform the same role. Moreover, the UC diagram indicates that both public and research users need to be involved with the system in order to perform certain functionalities which is incorrect.
2. A dependency is created between UCs Download Collections Data and Search Collections Data through improper use of pre and postconditions.

- (Please refer to antipattern **(a5.)** for details regarding the implications of this issue).
3. UCs in the Query Databases UC diagram are shown to *extend* each other, meaning that some UCs introduce optional or exceptional behavior to the functionality described in other UCs which is incorrect. The UCs have a hierarchical relation with respect to the query services that they offer, which is not shown.
 4. The UC model presents a number of functionally decomposed UCs, such as the Edit Collections Data, Upload DGB and UCM Data and Run QC Tests UCs. This is detrimental to the analytical quality of the UC model. Further implications of functional decomposition are presented in antipatterns **(a3.)**, **(a4.)** and **(a5.)**.
 5. The Database Edits UC diagram shows an incorrect type of dependency between the Geocode Specimen UC and the Update Collections Data UC.
 6. The UC model contains a superfluous actor: **Data Editor**.
 7. The Administrative Process UC diagram shows three UCs that are too generic to allow either of the administrator actors to perform their intended duties.
 8. The UC Model describes two actors that are system functionality not actors.
 9. UC Query Remote Database is indirectly accessed by an actor while it exclusively does not describe complete and meaningful functionality.

Many of these issues may have severe consequences downstream in the development process. It is crucial to remove these issues from the UC model. In

the following subsections, our proposed technique will be applied to the UC model in order to assess its ability to resolve these issues. All UC diagrams will be juxtaposed with the entire set of antipatterns. While performing the matching process, it is important to consider overlapping entities. That is, UCs or actors that exist in more than one diagram. Considering overlapping entities help reveal antipattern matches that may exist over multiple UC diagrams.

4.5.3. Analysis and Interpretation of the Results

The resulting antipattern matches shown in Table 4-3 require human inspection to verify the correctness of the UC model. A total of 11 antipattern matches were detected across all of the UC packages. An analysis of the antipattern matches of the first iteration is shown in Table 4-4. All antipatterns detected in the first iteration, with the exception of antipattern matches 1.2 and 6.1, are of Type (1). Therefore, they were detected automatically by ARBIUM. Antipattern match 1.2 (Type (3)) was detected by manually applying the anti-pattern template to the descriptions of the Download Collections Data and Search Collections Data UCs of the Database Access UC diagram. Meanwhile, antipattern match 6.1 (Type (2)) was detected by manually applying the anti-pattern template to the actor descriptions, while ARBIUM searched for these actors in the UC diagrams.

Table 4-3: First Iteration Matches

Match No.	UC Diagram	Antipattern Matched	Elements involved
1.1.1	Database Access	a6. Multiple actors associated with one UC	Actors: Public User and Research User UCs: Download Collections Data, Search Collections Data and Visualize Biodiversity Analysis
1.1.2		a8. Functional decomposition: Using pre and postconditions	UCs: Download Collections Data, Search Collections Data
1.2.1	Database Queries	a3. Functional decomposition: Using the <i>extend</i> relationship	UCs: All five UCs illustrated in the corresponding UC diagram.
3.1	Database Integrator	a5. Functional decomposition: Using the <i>include</i> relationship	UCs: Edit Collections Data and Update Collections Data.
1.3.2		a5. Functional decomposition: Using the <i>include</i> relationship	UCs: Upload DGB and UCM Data, Run QC Tests and Update Collections Data.
1.4.1	Database Edits	a5. Functional decomposition: Using the <i>include</i> relationship	UCs: Geocode Specimen and Find Locality
1.4.2		a4. Accessing an <i>extension</i> UC	Actors: Data Editor and Database Integrator. UCs: Edit Collections Data and Geocode Specimen.
1.5.1	Administrative Process	a6. Multiple actors associated with one UC	Actors: Database Administrator and ArcIMS Administrator UCs: Backup Process

1.5.2		a6. Multiple actors associated with one UC	Actors: Database Administrator and ArcIMS Administrator UCs: Restore Process
1.5.3		a6. Multiple actors associated with one UC	Actors: Database Administrator and ArcIMS Administrator UCs: Install Software Updates
1.6.1	System Wide	a7. A description of an actor that is not depicted in the UC diagram	Actors: Database Upload Process and Database QA/QC Process

The Database Edits, Database Queries and Database Integrator UC Diagrams were merged since they contain a number of overlapping entities. The merged UC diagram (“Merged UC Diagram”) is presented in Figure 4-58.

Table 4-4: First iteration analysis

Antipattern Match 1.1.1:

Analysis:

Upon analysis of the three UCs which the actors Public User and Research User are associated with, the actors were found to have similar roles when performing the UCs.

Corrective Actions:

The role that the actors play in correspondence to the three given UCs will be *generalized* into a separate actor (called User). The *generalized* actor is then associated with the UCs, while the Research User remains the only actor associated with UC Access Sensitive Data (Figure 4-57).

Antipattern Match 1.1.2:**Analysis:**

The precondition of the Download Collections Data UC states that the Search Collections Data UC must be initialized beforehand.

Corrective Actions:

Each UC offers a complete service individually hence they should remain separate. However, the precondition stated by the Download Collections Data UC should be removed.

Antipattern Match 1.2.1:**Analysis:**

The *extend* relationship was used to represent the hierarchy between the query services offered by the system.

Corrective Actions:

The *extend* relationships should be replaced with *generalization* relationships (Figure 4-58).

Antipattern Match 1.3.1:**Analysis:**

The Edit Collections Data UC represents subroutine type behavior that is required by the Update Collections Data UC.

Corrective Actions:

The functionality described in the Edit Collections Data UC should be merged with the description of the Update Collections Data UC and represented as a

“Sub-flow⁴”. Subsequently, the UC Edit Collections Data and its *include* relationship link with UC Update Collections Data are removed from the diagram (Figure 4-58).

Antipattern Match 1.3.2:

Analysis:

Analysis of the involved UCs show that updating the database requires the DGB (Denver Botanic Gardens) and UCM (University of Colorado Museum) data to be uploaded. Meanwhile, the task of uploading any data also requires that the data undergo Quality Control (QC) tests.

Corrective Actions:

The Upload DGM and UCM Data and Run QC Tests UCs should be merged into the Update Collections Data UC by modeling each as a separate “Sub-flow” component. Moreover, the description of the “Sub-flow” component responsible for uploading the data should indicate a requirement to execute the other “Sub-flow” that is responsible for running the QC tests. UCs Upload DGM and UCM Data and Run QC are removed from the Database Integrator diagram. Meanwhile, an *include* relationship will be directed from the Update Collections Data UC to the Query Remote Database UC, to replace the *include* relationship that was present between the Run QC Tests and Query Remote Database UCs (Figure 4-58).

Antipattern Match 1.4.1:

Analysis:

⁴ A “Sub-flow” is a component of a UC description that describes subroutine-like behavior that is exclusive only to the belonging UC.

The Find Locality UC represents subroutine type behavior that is required by the Geocode Specimen UC.

Corrective Actions:

The functionality described in the Find Locality UC should be merged and represented as a “Sub-flow” component of the Geocode Specimen UC. Hence, UC Find Locality and its *include* relationship with UC Geocode Specimen are removed from the diagram (Figure 4-58).

Antipattern Match 1.4.2:

Analysis:

The Edit Collections Data UC was merged into the Update Collections Data UC as a result of antipattern match 1.3.1 in the Data Integrator UC diagram. Therefore, actors Data Editor and Database Integrator are now associated with the UC Update Collections Data. Moreover, UC Update Collections Data now *extends* the Geocode Specimen UC.

- (a) Upon analyzing the *extended* UC Geocode Specimen, it is discovered that updating the database represents part of its required functionality.
- (b) The data-editing role played by the Geocoder actor is modeled using the Data Editor actor. However, the Geocoder is the only actor that edits this data. Moreover, the model shows that the Geocoder already has indirect access to the Update Collections Data UC, through the Geocode Specimen UC.

Corrective Actions:

- (a) The *extend* relationship between the involved UCs was used to indicate

subroutine type behavior. Therefore, this relationship should be replaced with an *include* relationship directed from the Geocode Specimen UC to the Update Collections Data UC. Hence, the Data Integrator actor is no longer directly accessing an *extension* UC (Figure 4-58).

- (b) Since the Geocoder actor already has indirect access to the Update Collections Data UC, the Data Editor actor is no longer required and should be removed (Figure 4-58).

Antipattern Matches 1.5.1, 1.5.2 and 1.5.3:

Analysis:

All three antipattern matches resulted from the same issue; the shared UCs are too general to suit either the ArcIMS Administrator or the Database Administrator actor. After reviewing the tasks of both actors, it was determined that the ArcIMS Administrator actor accesses the system to backup and restore the application code and to install code updates. Meanwhile, the Database Administrator actor accesses the system to backup and restore the *collections* data, and to install database updates.

Corrective Actions:

The three shared UCs should be split down into six UCs in order to properly represent the administrative duties of the actors (Figure 4-59).

Antipatterns Match 1.6.1:

Analysis:

Two actors Database Upload Process and Database QA/QC Process where

described but never depicted in any UC diagram. The descriptions of the actors however simply state functionality that is performed by the system itself, and hence should be part of the UC descriptions.

Corrective Actions:

No corrective actions are required since the actor tasks were already stated in the UC descriptions. The superfluous actors should be removed from the UC model.

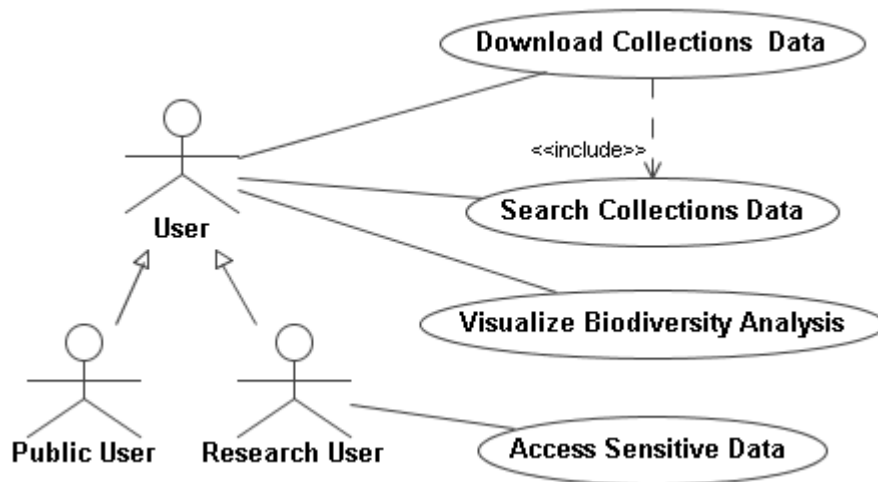


Figure 4-57: The Database Access UC diagram after the first iteration

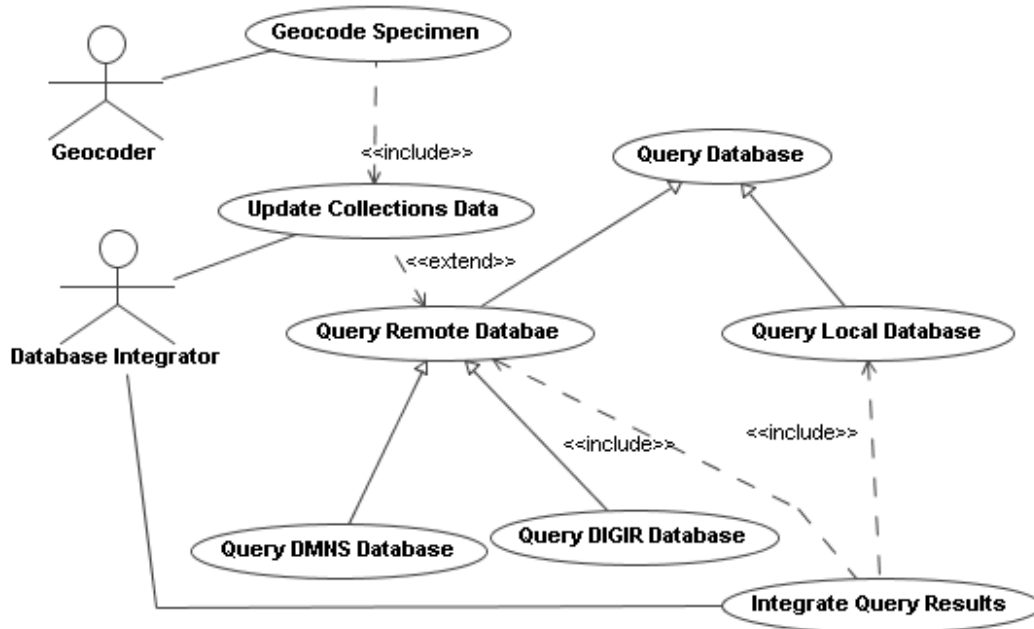


Figure 4-58: A merged view of the remaining three UC diagrams after the first iteration

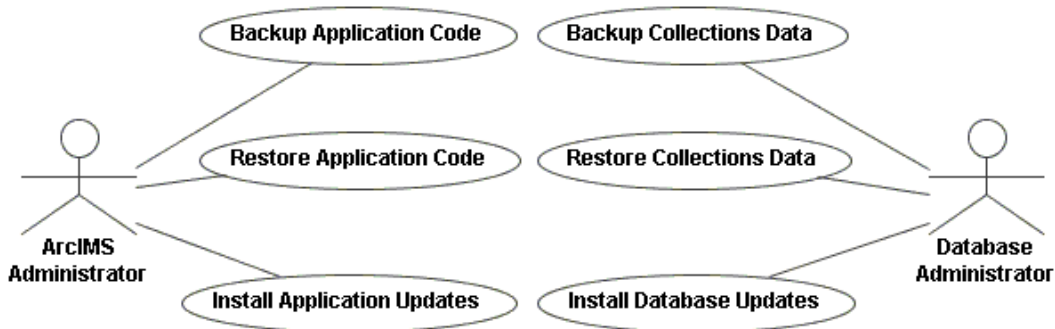


Figure 4-59: The Administrative Process UC diagram after the first iteration

As mentioned earlier, the proposed technique must be applied iteratively as corrections and changes applied upon reviewing an antipattern match might cause new antipatterns to surface. The matching process is repeated for a second iteration. Table 4-5 shows the antipattern matches detected during the second iteration, and Table 4-6 shows the corresponding analysis. All antipatterns matched are of Type (1) and hence were detected by ARBIUM. The antipattern matches were detected in the Merged UC diagram shown in Figure 4-58.

Table 4-5: Second iteration matches

Match No.	UC Diagram	Diagrammatic-Antipattern Matched	Elements involved
2.1.1	Merged UC Diagram (Figure 4-58)	a1. Accessing a <i>generalized concrete</i> UC	Actors: Database Integrator UCs: Query Remote Database and Integrate Query Results.
2.1.2		a2. UCs containing common and exceptional functionality	UCs: Query Remote Database, Update Collections Data and Geocode Specimen.
2.1.3		a4. Accessing an <i>extension</i> UC	Actor: Database Integrator UCs: Update Collections Data

Table 4-6: Second Iteration Analysis**Antipattern Match 2.1.1:****Analysis:**

This antipattern match resulted from replacing the inappropriately used *extend* relationships with *generalization* relationships. The *generalized* UC Query Remote Database is *concrete* and is indirectly accessed by the Database Integrator actor through the Integrate Query Results UC.

Corrective Actions:

According to the “Accessing a *generalized concrete* UC” antipattern (a1.), this situation may be fixed by setting the *generalized* Query Remote Database UC to be *abstract*. To conserve space, this minor change to the merged UC diagram (Figure 4-58) will not be shown.

Antipattern Match 2.1.2:**Analysis:**

The shared UC Update Collections Data contains subroutine behavior relative to the Geocode Specimen and Query Remote Database UCs.

Corrective Actions:

The *include* relationship with the UC and the Update Collections Data UC should remain intact. Meanwhile, the *extend* relationship between the Update Collections Data UC and the Query Remote Database UC should be replaced with an *include* relationship. To conserve space, this minor amendment to the merged UC diagram (Figure 4-58) will not be shown.

Antipattern Match 2.1.3:**Analysis:**

This antipattern no longer exists due to the corrective actions undertaken after analyzing antipattern match 2.1.2.

4.5.4. Discussion of Results and Validation

In this Section we assess whether the application of our technique resolved the issues that existed in the original MAPSTEDI UC model (see end of Section 4.5.2). Table 4-7 provides a summary of the issues resolved by applying our technique:

Table 4-7: Addressing issues in the MAPSTEDI UC model

Issue	Discussion and Validation	Resolved
1	The newly created <i>generalized</i> actor <i>User</i> represents the only role	✓

	<p>that exists while performing the shared UCs. Hence, the <i>generalization</i> relationship between the actors Research User and Public User, and their parent actor User, correctly indicates that they have the same role while performing the shared UCs. Having a single <i>generalized</i> actor access the previously shared UCs also eliminates the misinterpretation that both the Research User and Public User actors are required to be involved with the system simultaneously in order to perform the UCs. The only unshared UC Access Sensitive Data remains associated only with the Research User actor.</p>	
2	<p>Removal of the improper preconditions from the Search Collections Data UC complies with a widely accepted authoring guideline discussed in (Bittner et al. 2002). Now the Search Collections Data UC is appropriately dependent on the Download Collections Data UC through an <i>include</i> relationship only.</p>	✓
3	<p>The <i>generalization</i> relationships appropriately represent the hierarchy of services offered by the UCs in the Query Databases UC diagram.</p>	✓
4	<p>The analytical value of the UC model is greatly improved as the functionally decomposed UCs (Edit Collections Data, Upload DGB and UCM Data and Run QC Tests) are removed. Their respective functionalities are appropriately merged into their</p>	✓

	respective <i>base</i> UCs so that the <i>base</i> UCs describe complete and useful behavior.	
5	The Geocode Specimen UC is now appropriately set to <i>include</i> the Update Collections Data UC, representing the correct type of dependency that exists between the UCs.	✓
6	Each actor must have a distinct role. The Data Editor actor represents part of the role already performed by the Geocoder actor. The superfluous Data Editor actor is now removed from the UC model, eliminating redundancy and improving understandability.	✓
7	UCs must contain the correct level of detail in order to provide a complete and meaningful service to an actor. Each of the three overly general UCs shared by the administrator actors are now split into two separate UCs. The newly created UCs contain specific behavior to allow each administrator actor to perform their respective administrative duties.	✓
8	Every actor described in the UC model must be depicted at least once in a UC diagram. An actor is invalid if its description states functionality that is performed by the system itself. Therefore, actors Database Upload Process and Database QA/QC Process are now removed from the UC model.	✓
9	A UC should only be <i>concrete</i> if it can offer a complete service to an actor, which is not the case with the Query Remote Database	✓

	UC. Therefore, the Query Remote Database UC was set to be <i>abstract</i> to force one of its implementing UCs to carry out the specific behavior of querying a remote database.	
--	--	--

4.6. Comparison of Alternative Approaches

In order to fully evaluate the effectiveness of using antipatterns, it should be compared to alternative approaches by applying them to the MAPSTEDI UC model. As mentioned earlier in Section 4.2, only the approach presented by Berenbach in (Berenbach 2004) can be compared to our approach since it does not require significant human cognition to apply. The MAPSTEDI UC model was examined to determine if it violates any of the heuristics presented in (Berenbach 2004); these violations will then be “resolved” in the model. Table 4-8 presents the heuristics from (Berenbach 2004), and presents the number of violations found in the MAPSTEDI model. Each heuristic is stated is followed by the antipatterns that embody it; the heuristics from (Berenbach 2004) believe that the model is defect free!

Table 4-8: Examining the MAPSTEDI UC model for violations of the heuristics presented in (Berenbach 2007)

#	Heuristic	Violations Detected
1	“Every UC must be defined.” (Covered by a17.)	0
Analysis: Every UC was appropriately defined. The template used for each UC contained fields for the UC name, actors involved, preconditions, postconditions and the actual description of the		

	intended behavior. The description section of each UC stated a basic flow as well as alternative whenever applicable.	
2	“Abstract UCs must be realized with included or inheriting concrete UCs.” (Covered by a1. and a12.)	
3	“A concrete UC cannot include an abstract UC (unless it is realized).” (Covered by a3., a9. and a12.)	0
	Analysis: There was no abstract UCs in the original MAPSTEDI UC model.	
4	“Extending UC relationships can only exist between concrete UCs.” (Covered by a4., a6. and a9.)	
	Analysis: The extend relationship only existed in two diagrams: (a) the Data Edits and (b) the Database Queries UC diagrams. The Data Edits had one extend relationship between two concrete UCs. Meanwhile, the extend relationships between all UCs in the Database Queries UC diagrams are concrete.	0
5	“Use activity diagrams to show all possible scenarios associated with a UC.”	N/A
6	“The definition of a UC must be consistent across all diagrams defining the UC.”	
7	“Use sequence diagrams rather than collaboration diagrams to define one thread or path for a process.”	

8	“Avoid realization relationships and artifacts in the analysis models.”	
	<p>Analysis: Only the UC model of the MAPSTEDI was available.</p> <p>Moreover, this case study focuses on comparing approaches that improve UC models early in the development cycle where only the UC model is available.</p>	

Chapter 5

Developing Comprehensive

Acceptance Tests from Use Cases

5.1. Introduction

In agile development processes, the rewards from user acceptance testing are maximized by using the practice to drive the development process. Traditionally, User Stories are used in agile projects to describe a system's usage scenarios; and are utilized as a basis for developing acceptance tests. However, there remains a significant subset of agile projects that utilize another popular functional requirements modeling method, namely Use Case Modeling. This Chapter introduces a technique that aims to achieve the benefits of acceptance testing, and specifically acceptance test driven development, within agile approaches that utilize Use Case Models. The approach is based on utilizing a number of artifacts: Use Case Models "supported by" robustness diagrams and domain models. The feasibility of the proposed approach is demonstrated by applying it to a real-world system -- the RestoMapper system. The results show that a comprehensive set of acceptance tests can be developed based upon Use Case Models.

5.2. Developing Acceptance Tests from Use Cases

Acceptance tests are developed from requirements artifacts. In agile processes, acceptance tests are often constructed from USs (Cohn 2004); however, large-scale development projects that deploy a more rigorous development process such as the V-Model do not utilize USs. It is common for large-scale software projects within a V-Model development process to deploy a model-oriented approach throughout the development process. The UML in particular has become the de-facto modeling language for large-scale object-oriented software development, which has resulted in the widespread use of Use Case (OMG 2005) (UC) models for requirements analysis and modeling. Yet, there lacks a process that allows analysts and customers to develop acceptance tests from UC models. The focal point of this Section is to define such a process; that is, using requirements artifacts normally available during early development phases to drive, or at least support, the production process. UC models place an emphasis on system boundaries, and user-to-system expectations and interactions. This Chapter proposes a UC driven approach to developing executable acceptance tests, based on using UC models, robustness diagrams (Jacobson et al. 1992; Rosenberg et al. 1999, Rosenberg et al. 2005) and domain models. In this Chapter a tool UCAT (Use Case Acceptance Tester) is presented, which provides automation support for executing acceptance tests developed through our approach. It is important to note that our proposed approach does not attempt to replace or improve upon any other approaches that develop acceptance tests. Our approach provides a mechanism to develop executable acceptance tests based on UC models.

The concept of developing tests from UCs is not new. There have been many proposed techniques that attempt to generate system tests from UCs (Basanieri et al. 2002; Briand et al. 2002; Nebut al. 2006; Ryser et al. 1999). However, none of these techniques cater to the technical skill set of personnel who would carry out the construction of acceptance tests, namely customers and Business Analysts (BAs), whom are typically business-oriented rather than technically-oriented. Before comparing our approach with other works, it is crucial to identify the differences between user acceptance and system testing. System testing is similar to user acceptance testing in that it is concerned with testing an entire system from a black-box perspective. Both activities do not require knowledge of the underlying code structure. However, user acceptance and system testing remain distinct activities. Table 5-1 provides a detailed comparison between user acceptance and system testing.

Table 5-1: User acceptance testing vs. system testing

	System Testing	Use Acceptance Testing
What?	System testing compares the Software Under Test (SUT) with the <i>requirements specifications</i> . System testing is a validation activity to ensure that the SUT is functioning as intended. For example, considering a “simple addition” program, a system test	User acceptance testing compares the SUT with <i>end-user requirements</i> . It is a verification activity to ensure that the development team will be building the “right” system in the sense that end-product <i>will</i> satisfy the end-user requirements. For the “simple

	would be to ensure that the output <i>is</i> 7 when the inputs were 3 and 4-	addition” program, the behavior of the Software to be built is verified by an acceptance test that states that the output produced when the inputs are 3 and 4 <i>should be</i> 7.
Who?	<i>Developers</i> ideally create system tests. The rationale behind this is that upon the implementation of a functionality, it is the developers who are most knowledgeable of using (or interacting) with the software. Developers would have in depth knowledge of the input and output formats, which will enable them to create system tests, execute them and validate that the SUT has indeed behaved as intended.	<i>Customers</i> and <i>end-users</i> ideally create user acceptance tests, often with the aid of <i>Business Analysts</i> . The reason being is that it is the customers and end-users who are most knowledgeable of the problem domain and the required functionalities to improve their work processes. Meanwhile, Business Analysts are trained professionals who facilitate the process of requirements gathering, specification, analysis and verification (such as through user acceptance testing).
How?	As shown in Figure 5-1, user acceptance tests are a result of requirements analysis. User acceptance tests are developed based on various requirements artifacts. In large-scale development projects that	

	<p>deploy a v-model development process, it is common to develop UC and domain models as part of the requirements analysis process. Also as shown in Figure 5-1, system tests are based on user acceptance tests in addition to system design artifacts.</p>
--	--

Based on the purpose and properties of acceptance testing outlined in Table 5-1, there are desirable characteristics that should be present in any technique aimed at developing acceptance tests, which will typically be used by customers and BAs (Table 5-2).

Table 5-2: Desirable characteristics of a technique aimed at developing acceptance tests

1. Low technical difficulty:

Software testing tools developed for customers and BAs are intentionally designed to avoid any technical complexity, often relying on simple test tables (also known as fixtures). Such tools include FIT/Fitnesse (Mugridge al. 2005) and Selenium (Selenium 2008). Moreover, according to the duties and skills set of BAs outlined by BABOK (Business Analysts Body of Knowledge) (BABOK 2009), which is a well recognized official reference used by BAs to attain their BA certification, it can be deduced that a BA cannot be expected to perform any activities that can be considered technically highly complex. For example, BAs cannot be expected to perform complex mathematical calculations, formal methods formulations or use complex languages to specify and verify requirements. Similarly, it cannot be assumed that a customer can understand or use technically difficult analysis techniques and languages. It is crucial that a

technique aimed at developing acceptance tests must require a skill set that can be performed by BAs, whereby the developed acceptance tests are in a form that is understandable by customers.

2. Can be applied in the early phases of development:

It is important to develop acceptance tests early in the development process since according to the V-Model development process, the development of acceptance tests play a key role in developing other types of tests (system, integrations and unit tests) as the project progresses. During the early phases of development, it is common that a significant amount of requirements and design details are missing, or are only available at an abstract level. Hence, it is very important that a technique aimed at developing acceptance tests can be applied without the availability of such detailed information, while utilizing any abstract information available.

3. Help bridges the gap between analysis and design phases:

According to the V-Model development process, the process of developing acceptance tests along with requirements analysis should collaboratively guide the system design, and in turn system tests. A technique aimed at developing acceptance tests should bridge the gap between the analysis and design phases, leading to a seamless transition between the two phases.

4. Produces executable tests:

Executing acceptance tests manually is an error-prone and cumbersome process. It is desirable to develop executable acceptance tests to allow their execution to be performed effortlessly and accurately.

5. Produces reusable acceptance tests

Many acceptance are comprised of common components, for example, setup and cleanup tests. Therefore, it is highly beneficial to develop modular acceptance tests that can be reused to compose additional acceptance tests.

In (Briand et al. 2002), a testing methodology named TOTEM was introduced. TOTEM is used to develop test cases from various analysis artifacts, such as use case models, sequence, collaboration and class diagrams. However, to effectively deploy the TOTEM methodology, such analysis artifacts need to be heavily augmented with OCL (Object Constraint Language) expressions. It can be argued that learning OCL and effectively using it is an advanced skill beyond that can be expected from a customer or a BA (Table 5-2-point (1)). Inspired by the approach presented in (Briand et al. 2002), the technique introduced in (Nebut al. 2006) is also based on extending UCs with contracts to facilitate test case generation. The authors introduce a new language (similar to OCL) to express these use case contracts. Similar to the work in (Briand et al. 2002), it can be deduced that the approach presented in (Nebut al. 2006) is also beyond the capabilities of customers and BAs (Table 5-2-point (1)). In (Ryser et al. 1999), an approach named the SCENT-Method was introduced which uses scenarios embedded in UCs to develop tests. The technique requires the formalization of scenarios into statecharts. The systematically constructed statecharts are detailed with pre and postconditions, data ranges, data values and non-functional requirements. The statecharts then need to be crosschecked for any inconsistencies, incorrectness

and incompleteness. The authors have explicitly stated that a developer should perform the tasks comprising the SCENT-Method; this implies that they are too technically demanding to be performed effectively by a customer or a BA (Table 5-2-point (1)). The approach presented in (Basanieri et al. 2002) is based on two main components: (a) a strategy called Cowtest (Cost Weighted Test Strategy), and (b) a method to derive test cases from UML diagrams. The approach is tool supported with Cow_Suite. While the approach can be applied in the early development phases and based on abstract UML models, the approach is only useful when detailed UML design artifacts are constructed, mainly sequence and communication diagrams. Such a high level of detail can only be provided by developers or designers, rather than by customers and BAs. Without the availability of such detailed design artifacts, the tool can only render an abstract outline of test plan. This is due to the fact that the approach does not attempt to “understand” the requirements but rather systematically integrating scenario “steps” from use cases to generate test cases (Table 5-2-point (2)). The approach presented in this Section overcomes this limitation since it requires the customer and BA to make sense of the requirements before creating tests. The tests developed are hence more effective than tests generated by the approach presented in (Basanieri et al. 2002) as they serve the purpose of evaluating the outcome and behavior of a system rather than simply executing “steps” from use case scenarios.

Apart from the approach presented in (Basanieri et al. 2002), none of the other mentioned approaches produce executable tests (Table 5-2-point (4)). It is

also important to note that none of the approaches assist in bridging the gap between the analysis and design phases (Table 5-2-point (5)).

The approach presented in this Section is intentionally devised to account for all the desired features and characteristics outlined in Table 5-2. The approach presented in this Section provides customers and BAs with a technically simple approach to develop tests that are more comprehensive and effective than those created based on an ad-hoc approach. As such, the skill set required to perform our approach is that within the typical capabilities BAs. The artifacts involved in our approach are UC and domain models, and robustness diagrams. A BA is typically expected to be able to develop UC and domain models, while robustness analysis is a technique that a BA can learn and effectively use in a relatively very short period. A customer can be expected to understand all artifacts involved in our technique (Table 5-2-point (1)). Performing robustness analysis helps bridge the gap between the analysis and design phases (Table 5-2-point (3)). The approach can be applied in the early phases of development where a great deal of requirements and design details are missing, while utilizing abstract information available from the artifacts involved in our approach (Table 5-2-point (2)). The acceptance tests produced are executable (Table 5-2-point (4)) with tool support readily available. The acceptance tests produced are modular in nature and can be reused to compose additional acceptance tests (Table 5-2-point (5)). The compliance of our approach with the entire set of characteristics outlined in Table 5-2 is demonstrated in the case study presented in Section 5.5.

In this Chapter, the process of creating user acceptance tests from UCs consists of three principle phases (see Figure 5-1):

Phase 1: Developing High Level Acceptance Tests (HLATs) for each UC to evaluate a system's behavior when a given UC is performed.

Phase 2: Performing *robustness analysis* to identify object level information.

Phase 3: Developing Executable Acceptance Tests (EATs) for each UC using the object level information (identified in Phase 2) that will realize the HLATs previously created in the Phase 1.

High Level Analysis

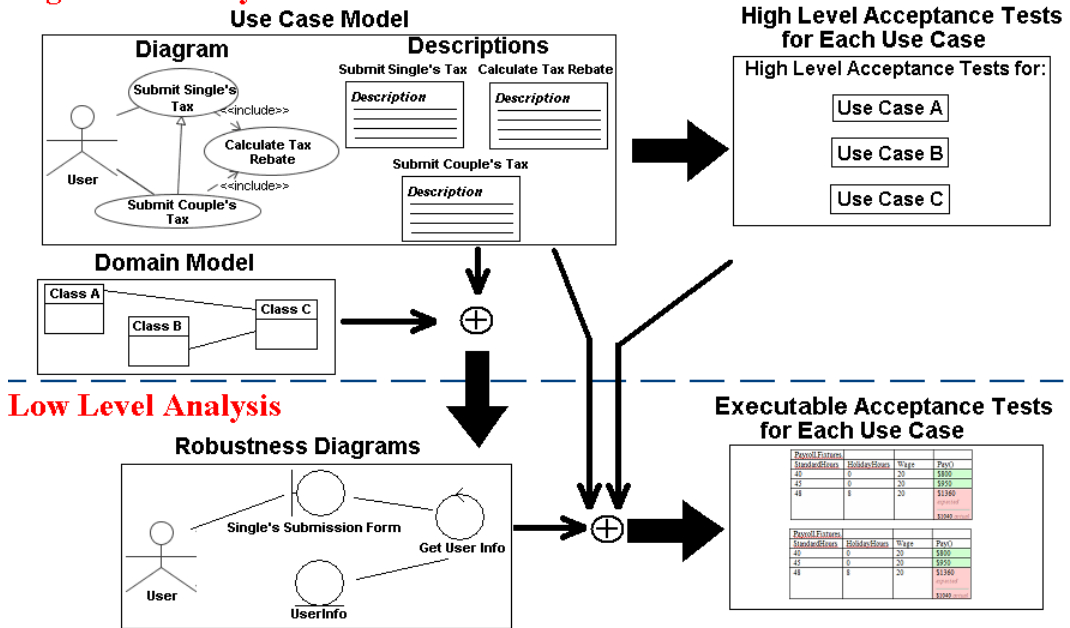


Figure 5-1: The overall process of developing high and executable acceptance tests per UC

5.2.1. Phase 1: Developing High Level Acceptance Tests for Each Use Case

HLATs provide an informal and abstract level description of acceptance tests. HLATs are composed of semi-narrative text. The purpose of HLATs is to decouple the process of identifying the required set of acceptance tests from any technical details pertaining to any particular programming language or syntax for creating EATs. For each UC, a set of HLATs are developed to account for its usage scenarios. HLATs are developed by examining a UC and its corresponding domain model. A number of UC templates exist (Cockburn 2000; Harwood 1997; Jaaksi 1998; Kulak et al. 2000; Mattingly al. 1988; Schneider et al. 1998); our proposed approach is independent of any specific UC description template as long as it appropriately defines *basic* and *alternative flows*. Ideally, a UC description should also state its name, the actors involved, any triggers, pre and postconditions, and extension points (Cockburn 2000).

Acceptance testing involves the development of a series of tests that simulate various usage scenarios of the SUT. In order to simulate a usage scenario (run a test), the SUT is provided with a series of input. This input can be data or function calls. The tests are evaluated by checking the system's resulting output. Therefore, the process of creating the HLATs is based on examining the UC descriptions and the domain model, while asking the following three key questions:

Q1: What are the usage scenarios?

The purpose of asking this question is to identify the entire set of usage scenarios from the UC descriptions. UC descriptions should ideally describe the entire set of a system's functional requirements, at least the most crucial and commonly used ones. UC descriptions describe functional requirements in the form of narrated scenarios, which makes UC descriptions an ideal source for identifying usage scenarios. It is common for a UC description to embody a number of scenarios that may occur while delivering an intended service. As a guideline, the basic flow of a UC represents an individual scenario, while each alternative flow represents another individual scenario. It should be noted that it is not required to examine the domain model in order to identify usage scenarios. The domain model does not contain any information regarding how a system will be used. The domain model only contains static information regarding real-world entities.

Q2: What are the required inputs or triggers?

Most UCs require an input or trigger in order for their embodied scenarios to be performed. An actor, another UC, or another subsystem can provide this input. For each scenario, it is required to determine the requisite inputs or triggers and pre-conditions. For example, a system that verifies credit cards will require a credit card number, expiry date and the cardholder's name for input. It can be deduced that the system cannot perform its validation functionality without first obtaining this data. Therefore, the purpose of this question is to determine the inputs, triggers and pre-conditions that are required to activate these embodied

scenarios. This information can be determined by examining the UC description itself, or the domain model. Such information would ideally be present in a UC description, since this information is essential to the business logic behind the described scenario. Otherwise, the scenario may be incomprehensible, or worse, misinterpreted. Information regarding inputs should also ideally be present in the domain model, since inputs often represent elements of the real-world domain. In the credit card example, a credit card is considered as part of the real-world domain, and hence it is expected that the domain model will contain a class representing credit cards. It is also expected that this class will contain attributes representing the credit card's number, expiry date and cardholder's name, etc.

Q3: What output is expected from the UC?

The purpose of this question is to determine the criteria that will be used to evaluate the success (or failure) of the system to deliver an intended service. The output generated by the system is compared to the expected results that are stated in UC descriptions. Every UC is expected to perform a *step* or a set of *steps* resulting in outputs. The term "step" is defined in (Cockburn 2000) as an action taken during the execution of a scenario that contributes towards the completion of the scenario. The template presented in (Cockburn 2000) is considered an industry standard. An output is intended for an actor, another UC, or another subsystem. For each scenario, it is required to determine outputs and post-conditions. Similar to Q2, information regarding outputs and post-conditions can be located in the domain model in addition to UC descriptions. It is likely that the

output represents a real-world entity or attribute. For example, if a car travelled 10 kilometers in a given trip, the value displayed by the odometer is expected to be 10 kilometers greater than the value it displayed before the trip started. The domain model representing such a system will ideally contain an attribute that represents the distance travelled.

Answering the above questions will provide the necessary information required to create HLATs pertaining to a single UC (see Figure 5-2). This process should be iterated for each UC. HLATs of a given UC are defined in the form of a table. The table is composed of three main columns, each row representing a distinct HLAT. Table 5-3 presents the format of HLATs used in this Section. It is important to note that HLAT tables do not contain any specific keywords as part of its syntax. In Table 5-3, keywords such as *precondition* and *input* are used to state pre-conditions and inputs used in a given test. However, analysts are free to use any keywords that better suits their needs. The rationale behind not mandating any particular syntax is to allow analysts the greatest flexibility while describing their HLATs.

Table 5-3: HLAT format: The first column of a HLAT defines a unique test ID. The test IDs used in this Chapter are a combination of the flow name and the belonging UC name. The second column describes the pre and postcondition(s), as well as inputs and triggers provided into the system to execute the given scenario. The third column is used to describe the expected output.

Test ID	Description	Expected Results
UC-Name -Basic Flow	<i>Precondition:</i> Preconditions for running the flow. <i>Input:</i> Data attribute-1 <i>Input:</i> Data attribute-2	The expected results in natural language form.

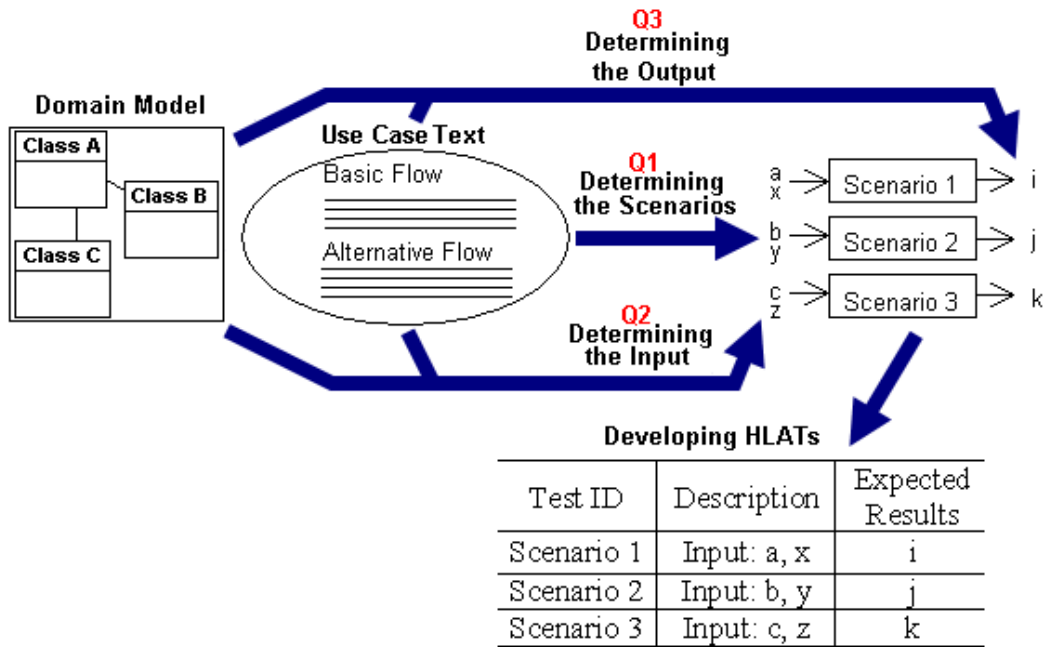





Figure 5-2: Developing HLATs by analyzing the UC text and domain model

5.2.2. Phase 2: Performing Robustness Analysis

Without automation support, analysts can run acceptance tests manually, which can be a cumbersome, time consuming and error-prone process. Therefore, the proposed approach aims to develop executable acceptance tests (EATs) to allow for automation support. The HLATs developed in the previous phase are refined into EATs. This is achieved via robustness analysis. Robustness analysis involves the analysis of robustness diagrams (Jacobson et al. 1992; Rosenberg et al. 1999, 2005) and their corresponding HLATs. Robustness diagrams provide a first-cut attempt at linking UCs with objects already present in a domain model. Robustness analysis will reveal “missing” objects from the solution domain that are required to realize UCs and provide an initial view of the object structure and behavioral aspects of a system without committing to any specific design. Ideally, there is one robustness diagram for each UC. Objects identified from the solution

domain are classified as either: a boundary, a control or an entity object. Table 5-4 provides a brief description of each type of object.

Table 5-4: Robustness diagram objects

Object	Symbol	Description
Boundary		Boundary objects facilitate communication with Actors.
Control		Control objects act as coordinators of activities implementing intended business processes. A control object utilizes inputs provided by interface objects, along with existing entity objects, to derive outputs that are “sent back” to interface objects or used to update entity objects. Control objects provide the “glue” between Boundary and Entity objects. It is common for control objects not to represent actual objects in the eventual class diagram but to represent a series of function calls that carry out the application logic.
Entity		Entity objects are typically entities that exist within the domain model.

The notational set of robustness diagrams is relatively small with a small set of syntax rules and semantics that are easy to understand, which makes them easy to create, read and update. (The interested reader should consult Jacobson et al. 1992, Rosenberg et al. 1999, or Rosenberg et al. 2005 for detailed discussions on robustness diagrams.) Actors can only be associated with Boundary objects, while

Boundary objects can only be associated with Control objects. Control objects can be associated with any type of object. An Entity object cannot be associated with another Entity object. Robustness diagrams also depict actors and UCs to illustrate the involvement of the represented robustness diagram with actors and other UCs. The three object types of robustness diagrams map well into the *Model-View-Controller* (MVC) design pattern. Where a Boundary object maps to the *View* concept, a Control object maps to the *Controller* concept, and an Entity object maps to the *Model* concept. There lacks a formal approach to develop the initial set of objects in robustness diagrams. However, the nature of UC descriptions aids the derivation of the three types of objects:

- A scenario in a UC description will narrate an interaction of an actor with the system, which prompts the need to consider the interface (Boundary objects) that the actor uses to interact with the system.
- A scenario will also state the business logic required to deliver a service, thereby prompting the introduction of Control objects.
- Entity objects (data) are required by Control objects to perform the intended business logic.

After constructing a robustness diagram, the domain model should be updated with new objects and attributes introduced while developing the robustness diagram. Moreover, the corresponding UC description should be updated, to ensure that it conforms to the robustness diagram and by implication the domain model. Therefore, the term “robustness” is used since the construction of robustness diagrams results in ensuring a consistent view between the UC and

domain models, while “filling in” information that might be missing in either artifact. This information is essential for the construction of acceptance tests.

These concepts are elaborated using the banking system UC model in Figure 5-3. The “Withdraw Cash” and “Deposit Funds” UCs describe behavior for withdrawing and depositing funds, respectively. The “Perform Transaction” UC describes behavior for logging the transactions performed by its child UCs. Finally, the “Calculate Investments” UC describes the behavior responsible for calculating the return on investments. The “Withdraw Cash” UC description and its respective domain model are shown in Figure 5-4.

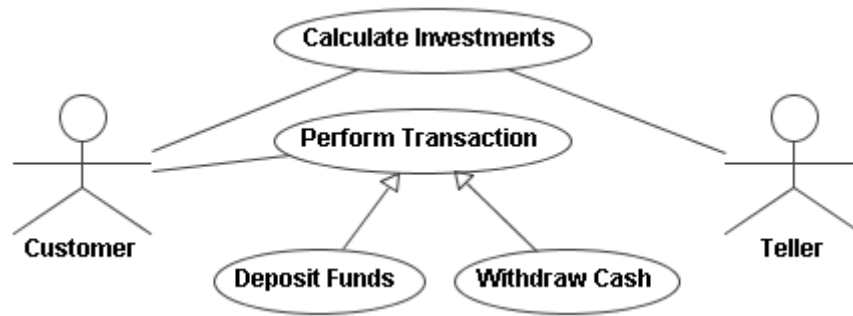


Figure 5-3: The banking system UC model

<p>UC Name: Withdraw Cash</p> <p>Actors: Customer</p> <p>Basic Flow: The Customer selects the account that he wishes to withdraw money from and the desired amount. The Customer can choose to withdraw money from a cheque-ing or savings account. The system checks if sufficient funds exists and dispenses the appropriate amount.</p>	<pre> graph TD Customer((Customer)) --- AccountViewer[AccountViewer] AccountViewer --- BankServer[BankServer] BankServer --- Account[Account] </pre>
---	--

Figure 5-4: The “Withdraw Cash” UC description (left) and domain model (right)

When analyzing the narrative text of the “Withdraw Cash” UC, it can be deduced that the Customer interacts with the system through an interface that displays the Customer’s accounts. A Boundary object is hence required to handle the interaction of the Customer with the system and to display the respective accounts. The domain model already contains such an object (AccountViewer), and therefore instead of adding a new object, the Boundary object depicted in the robustness diagram will represent the preexisting domain object. The Boundary object will be responsible for retrieving the Customer’s account selection and the desired cash amount. Before proceeding to analyze further text, it is important to represent entities that handle the different account types (“ChequingAccount” and “SavingsAccount”). The UC description then states that the system checks if there are sufficient funds in the chosen account. A Control object is required to handle this business logic; the newly created Control object “Check Sufficient Funds”. Another Control object, “Dispense Cash”, handles the dispensing. The UC description ends at this point; however, a vital logical operation is missing, that of deducting the dispensed amount from the Customer’s account. Therefore, a Control object “Deduct Amount” is introduced. The robustness diagram created is presented in Figure 5-5.

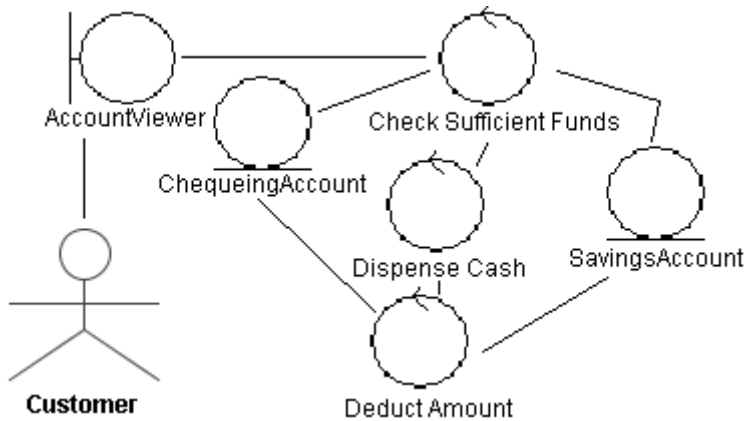


Figure 5-5: The robustness diagram corresponding to the “Withdraw Cash” UC

The actor “Customer” is depicted and associated (only) with the Boundary object “AccountViewer”. During the construction of the robustness diagram, two Entity objects were introduced (“ChequeingAccount” and “SavingsAccount”). As shown in Figure 5-5, “AccountViewer” is linked with a series of Control objects, starting with “Check Sufficient Funds” and ending with “Deduct Amount”; these carry out the business logic behind the Basic Flow of the UC. The control objects “Check Sufficient Funds” and “Deduct Amount” are both linked to the Entity objects “ChequeingAccount” and “SavingsAccount”, since they require access to these objects to read and update them. The domain model should be updated with these newly created Entity objects. Meanwhile, the UC description should be updated accordingly. The updated domain model and UC description are shown in Figure 5-6. The newly added text in the UC description is shown in *italic* font.

<p>UC Name: Withdraw Cash</p> <p>Actors: Customer</p> <p>Basic Flow: The Customer selects the account that he wishes to withdraw money from and the desired amount. The Customer can choose to withdraw</p>
--

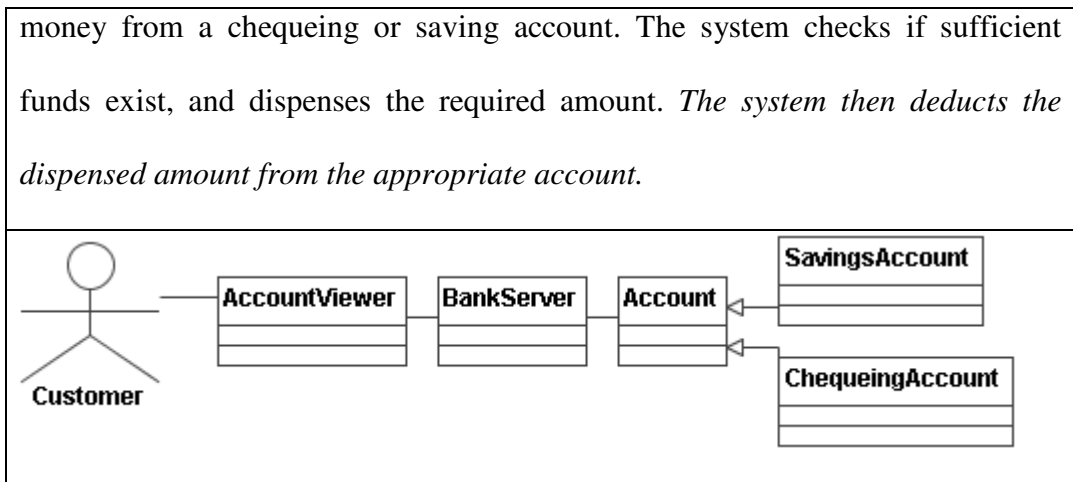


Figure 5-6: The updated “Withdraw Cash” UC description (top) and domain model (bottom)

After creating the robustness diagrams and updating the UC and domain models, robustness analysis is performed to refine each HLAT into an EAT. Acceptance testing is a black-box activity. Executing an acceptance test simply involves the injection of certain input into the system and the evaluation of the output it produces. Therefore, in order to execute an acceptance test, it is required to determine the object-level information that corresponds to the inputs and outputs. This is the principle purpose of performing robustness analysis. HLATs, UC descriptions and the domain model are used to perform robustness analysis. HLATs are used since they already contain the scenarios (extracted from UC descriptions) that require to be tested, and hence each EAT created will correspond directly to a single HLAT. Moreover, HLATs state the inputs, preconditions, outputs and postconditions of each scenario. UC descriptions are used since they contain the narrative text that corresponds to each scenario contained in an HLAT. The narrative text is used to perform tracing activities that will allow us to ultimately determine the object-level information required to produce EATs. The object-level information retrieved is that corresponding to the

inputs, preconditions, outputs and postconditions stated in HLATs. Note that ideally domain models are not required for robustness analysis since at that point the information that they contain should already be present in the UC descriptions. Robustness analysis is performed by applying the following steps:

Step 1:

Determine the set of steps that are contained within each UC flow. A scenario (a single UC flow) can be decomposed as a series of steps. However, many UC flows are authored in a paragraph-like format. In this case, the extraction of “steps” from a UC flow description will require human judgment. The narrative text is divided into “steps” the purposes of performing “Step 2”.

Step 2:

For each of the steps determined from Step 1, we trace the flow of control between the objects in the corresponding robustness diagram. This will allow us to determine the set of objects that correspond to each step’s inputs, preconditions, outputs and postconditions, previously stated in the corresponding HLATs. These objects will be evaluated to determine if the system behaved as expected. The object-level information resulting from robustness analysis is used to create EATs (Phase 3).

5.2.3. Phase 3: Developing Executable Acceptance Tests for Each Use Case

Object-level information retrieved from robustness analysis is used to create EATs that correspond to the previously developed HLATs. Even though the ultimate goal of our approach is to develop a set of EATs, our approach is independent of any implementation solution. Developers can use any automated framework to develop and execute acceptance tests. A number of such tools are readily available such as FIT/FITnesse (Mugridge al. 2005) and Selenium (Selenium 2008). For illustrative purposes, in this Section we present the EATs for the “Perform Transaction” UC, presented in Section 5.5.1.3, using the FIT/FITnesse syntax, since it is arguably the most commonly used framework for developing and executing acceptance tests. The FIT/FITnesse syntax contains a small number of command keywords (Mugridge al. 2005). Tests are created in the form of simple tables, more commonly known as FIT *fixtures*. FIT fixtures can be created using spreadsheets or an HTML editor. FIT defines three standardized fixtures to compose tests: (a) *ActionFixture*, (b) *ColumnFixture*, and (c) *RowFixture*; each fixture represents a different “style” of test case. The syntax for each type of fixture is designed to support a “style” of test case. For example, *ActionFixtures* support certain keywords (described below) that allow it to be used for step-by-step processing to check that a sequence of actions executed will result in producing expected outputs. Each row in an *ActionFixture* (apart from the fixture header) is a single “action”. A row is composed of a number of fields. The first field of each row is used to specify one of the following commands:

- The **Enter** command: this field is followed by a field containing the name of a data value entered, which is followed by a field containing the actual data value.
- The **Check** command: this field is followed by a field containing the name of a data value to be evaluated, which is followed by a field containing the expected data value.
- The **Press** command: this field is followed by a field containing the name of a function to be executed, which is optionally followed by a number of fields containing the appropriate parameter data values.

The structure of an *ActionFixture* is shown Figure 5-7; and its application is illustrated with a test that performs various transactions described by the “Perform Transactions” UC. For the interested reader, a detailed explanation of the *ColumnFixture* and *RowFixture* syntax is presented in Appendix E.

PerformTransactions		
Check	SavingsBalance	0
Check	ChequeingBalance	0
Enter	Amount	200
Press	Deposit	Amount
Enter	Amount	300
Press	Deposit	Amount
Enter	Amount	100
Press	Withdraw	Amount
Check	SavingsBalance	200
Check	ChequeingBalance	200

Figure 5-7: ActionFixture example of performing transactions. This test assumes a scenario where the balance of both saving and cheque-ing accounts are nil. A variable, “Amount”, is used to store values provided by the “Customer”. The “Amount” variable is passed as a parameter during a series of function calls to deposit and withdraw funds. The test ends by checking the balance of both accounts.

5.3. Tool Support with UCAT

The approach proposed in this Section describes the development of EATs and associating them with their respective UCs. However, the FIT/FITnesse framework is insufficient, as it is not equipped to handle UC models. To achieve this goal, a tool named UCAT (**U**se **C**ase **A**cceptance **T**ester) was developed to enable UCs to be augmented with FIT fixtures (Figure 5-8); the tool includes functionality from a previously developed tool SAREUCD (El-Attar et al. 2006a). Using UCAT, an analyst can build an entire UC Model. Figure 5-8 shows the banking system UC model built within the tool. The left-hand pane of the tool displays the UCs in the UC model. The middle pane shows the actors involved in the UC model. The right-hand pane displays all relationships that exist in the model, including association, generalization, inclusion and extension relationships. UCAT is initially based on the FIT/FITnesse framework to maximize the accessibility of our approach; however it provides a flexible interface which allows other acceptance testing frameworks to be quickly integrated. Hence, currently UCAT requires that EATs be encoded according to the FIT/FITnesse syntax (Mugridge al. 2005). Test results are also associated with UCs and can be automatically run through UCAT, which aids requirements traceability (see Figure 5-9).

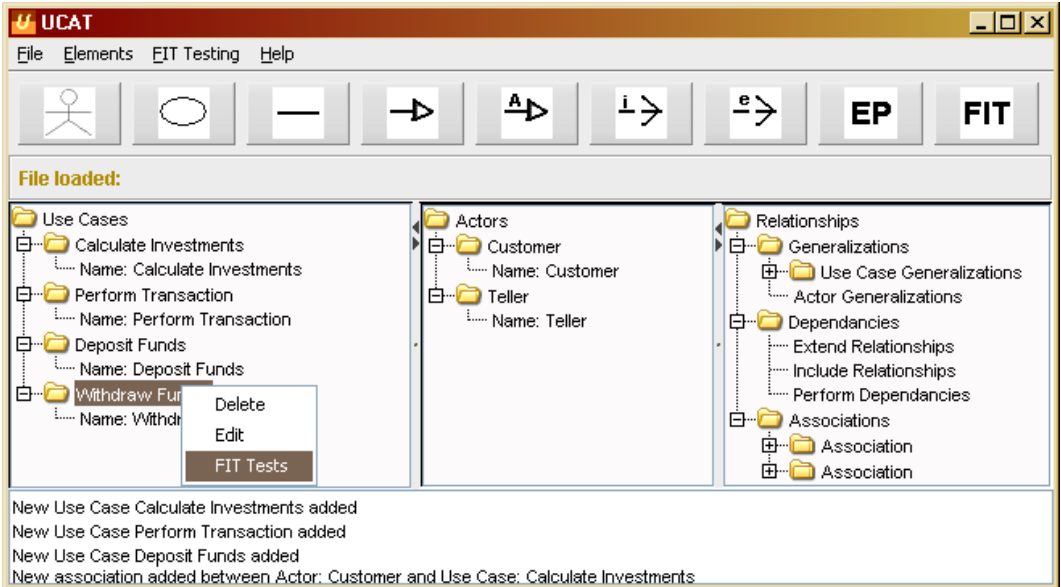


Figure 5-8: UCAT – associating FIT tests with the “Withdraw Cash” UC

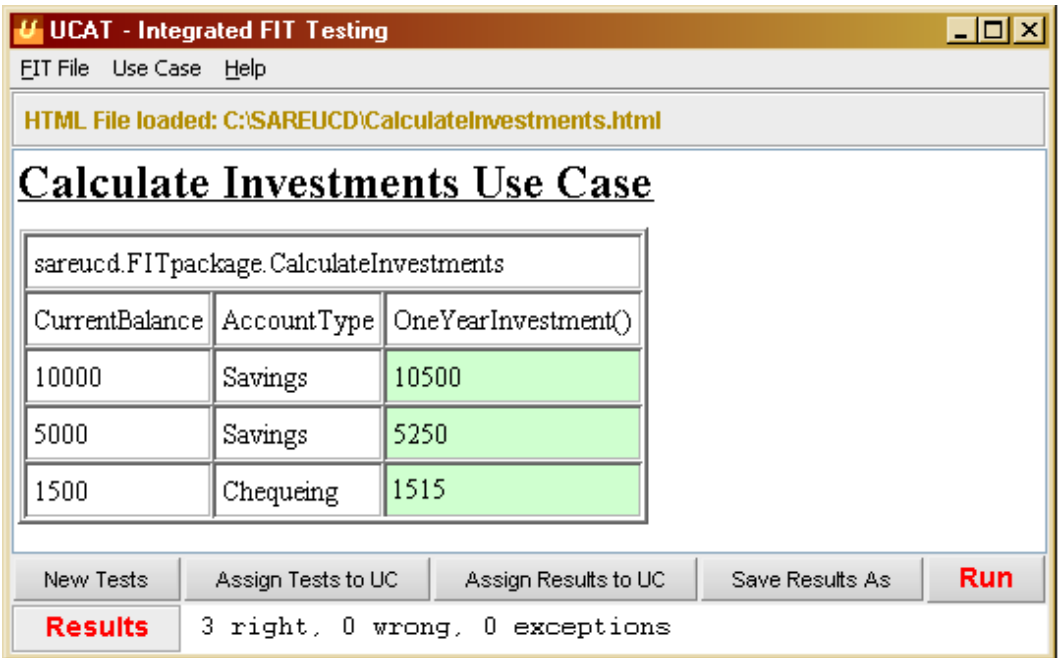


Figure 5-9: UCAT – displaying test results of the “Calculate Investments” UC

5.4. Evaluating the Efficiency of the Developed Tests

As mentioned earlier, acceptance testing is used as a validation process to ensure that the right system is being built. Therefore, the efficiency of the developed acceptance tests is dependent on the quality of the corresponding UC model. The quality attributes of UC models are specific in Table 2-2 of Chapter 2. For example, if a UC model is lacking the description of a particular functionality, it will not be possible to create acceptance tests to test this functionality by analyzing the UC model. It is the responsibility of the customer and analysts to explore the most important usage scenarios and document them into the UC model. This dependence on the quality of UC models, which heavily relies on the skill and experience of analysts and customers, significantly impedes the ability to empirically or formally validate the effectiveness of our proposed approach. Producing high quality UC models is outside the scope of this Chapter.

5.5. The RestoMapper System Case Study

This Section presents the RestoMapper⁵ system case study to demonstrate how the proposed approach can be used to produce acceptance tests using a UC model and its associated robustness diagrams. The RestoMapper system is a mapping application that locates restaurants in any city. RestoMapper provides a graphical interface that allows its user to acquire detailed information regarding their desired restaurants. The restaurant information can be displayed in pop-up or

⁵ The RestoMapper system is based on the Mapplet system which is feature and discussed in great detail in the book *Agile Development with ICONIX Process: People, Process, and Pragmatism* authored by Rosenberg et al. [Rosenberg]. The RestoMapper system has been modified to prevent any copyright infringements.

using an internet browser application. RestoMapper also contains a feature that allows users to configure a display filter that will cause the application to display only the restaurants that meet their desired criteria. The available criterion for users is valet parking availability; smoking section availability; and live music. The UC model of the RestoMapper system is presented in Figure 5-10. The UC model was used as input to UCAT (Figure 5-11). The UC model contains one actor and five UCs. A brief description of the model's elements are shown in Table 5-5.

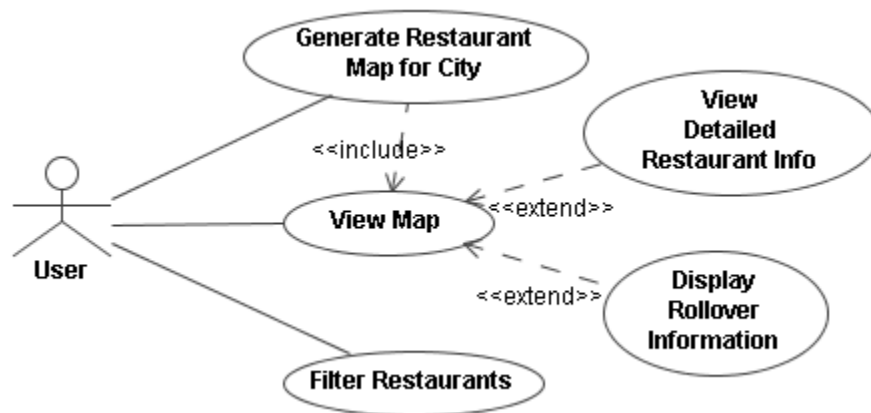


Figure 5-10: The UC diagram of the RestoMapper application

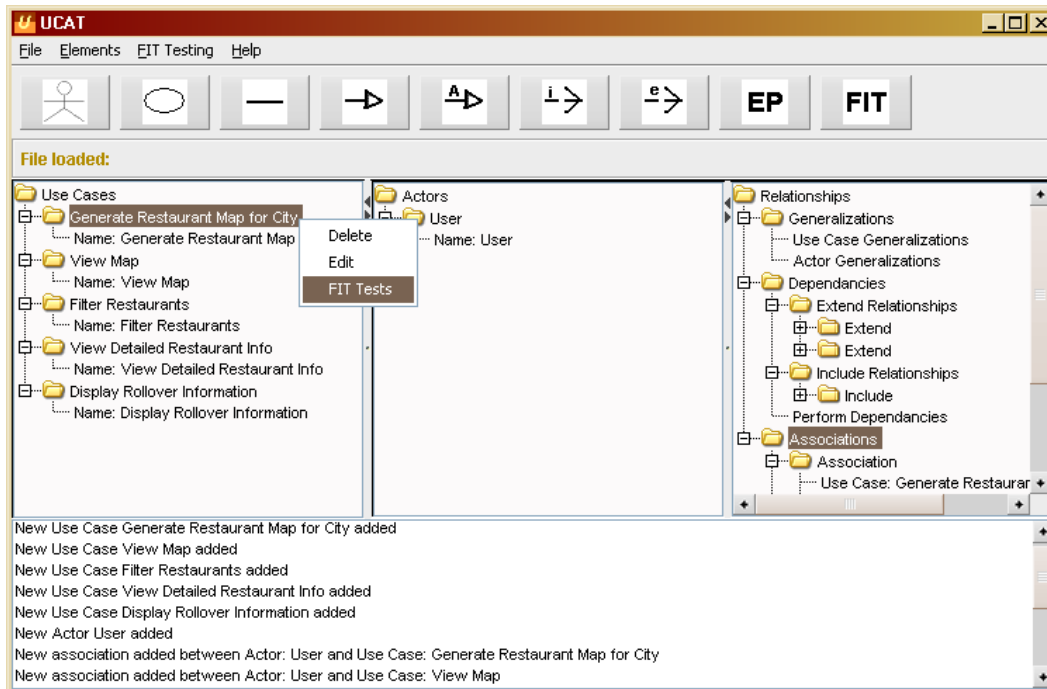


Figure 5-11: The RestoMapper UC model into UCAT

Table 5-5: Properties of the RestoMapper UC model

Element	Purpose
User	“User” is the sole actor interacting with the system and is the only beneficiary of the system’s services. These services are briefly explained below.
View Map	This UC describes the behavior required to display a map to the “User”.
Filter Restaurants	The “User” is associated with this UC to specify display settings, such as displaying restaurants with live music, a designated smoking section and available valet parking.
Generate Restaurant Map	This UC describes behavior responsible for determining the components that will be depicted on the map. The

for City	components are determined according to the city, zoom and display settings selected by the “User”. The resulting map is displayed via the “View Map” UC. The display settings are configured by the “Filter Restaurants” UC.
Display Rollover Information	This UC describes behavior that allows a “User” to retrieve information regarding a particular displayed restaurant(s) from the displayed map; and displays this information in a pop-up window.
View Detailed Restaurant Info	This UC describes behavior that allows a “User” to retrieve comprehensive information regarding a particular restaurant; and displays this information in an Internet browser.

For presentation and brevity purposes, only two UCs: “Generate Restaurant Map for City” and “Display Rollover Information”, are described and elaborated with robustness diagrams.

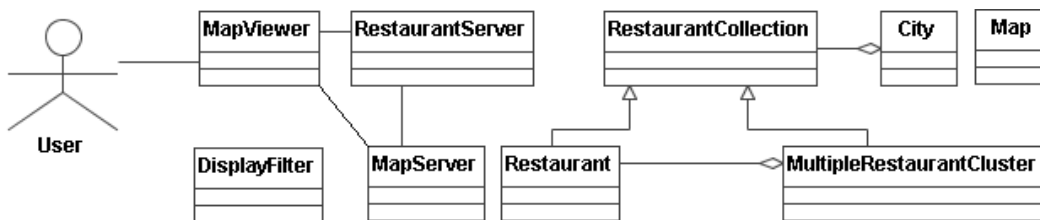


Figure 5-12: The initial domain model of the RestoMapper system

The preliminary domain model is presented in Figure 5-12. The domain model is usually built through a brainstorming process. The following is brief description of the objects contained in the initial domain model (see Table 5-6):

Table 5-6: Objects in the domain model

Object	Purpose
MapView	Responsible for interacting with the “User” and retrieving the “User’s” viewing preferences.
RestaurantServer	Responsible for managing and providing information regarding all restaurants.
MapServer	Responsible for generating visual maps.
Map	Contains data regarding the map to be displayed.
City	Contains data regarding a single city.
Restaurant	Contains data regarding a single restaurant.
DisplayFilter	Contains data regarding the “User’s” filter settings.
RestaurantCollection	A collection of all restaurants in a given city.
MultipleRestaurantCluster	A cluster containing a subset of restaurants contained in RestaurantCollection. The cluster is derived according to the “User’s” filter settings stored in DisplayFilter.

The aim of applying our approach is to create a set of executable acceptance tests that will cover usage scenarios described by UCs: “Generate Restaurant Map for City” and “Display Rollover Information”. The following is an outline of the analysis performed to produce acceptance testing for this case study:

- Sections 5.5.1: The UC description of the *Generate Restaurant Map for City* UC is presented. The UC is analyzed in the proceeding subsections.
 - Sections 5.5.1.1 (Phase 1): The textual description of the UC is analyzed to create a set of HLATs, which test every flow in the UC.
 - Sections 5.5.1.2 (Phase 2): Robustness analysis is performed using the UC description and its corresponding robustness diagram. Robustness analysis identifies the inputs and outputs of each flow and the objects (from the robustness diagram) that correspond to the identified inputs and outputs.
 - Sections 5.5.1.3 (Phase 3): The identified objects are used to create EATs to implement the HLATs previously created in Section 5.5.1.1.
- Appendix G follows a similar structure to that of Section 5.5.1 to analyze UC “Display Rollover Information”.

5.5.1. UC: Generate Restaurant Map for City

This UC generates a map for a given city with restaurant icons according to criteria set by a user. The actor “User” selects a city and the system retrieves the list of restaurants and their coordinates from the database. The current display filter settings are retrieved to determine the type of restaurants that can be

displayed. The “View Map” UC performs the displaying of the map. The UC description is shown in Figure 5-13.

Basic Flow:

The *MapView* queries the *RestaurantServer* for restaurants within the provided city. The list of restaurants retrieved are then stored in a *RestaurantCollection*. Using a specified scale, the *MapView* obtains the Map for the provided city from the *MapServer*. The *MapView* then proceeds to add an icon for every restaurant contained in the *RestaurantCollection* that also meets the criteria defined in the *DisplayFilter*. The display criteria can be used to specify restaurant characteristics such as the availability of a smoking section, valet parking and live music. UC *View Map* is then activated to display the map.

Alternative Flow: Invalid Zoom Setting

If the *MapView* determines that the map is zoomed out beyond a predefined value, the *MapView* will not query the *RestaurantServer*. Instead, the *MapView* will display a popup to the user indicating that the map is zoomed out beyond an acceptable scale.

Figure 5-13: Textual description of the *Generate Restaurant Map for City* UC

5.5.1.1 Examining the UC Descriptions and Creating its HLATs (Phase 1)

We begin the process by examining the textual description of the UC (Figure 5-13). The UC description contains two flows that require a zoom and display filter setting, plus a specified city, to allow the system to generate a city map with the appropriate restaurant icons displayed. The flows are similar with the exception of

having different zoom validity settings. The set of HLATs created are shown in Table 5-7.

Table 5-7: HLATs for the *Generate Restaurant Map for City UC*

Test ID	Description	Expected Results
GRMC ⁶ -Basic Flow	<i>Precondition:</i> Run RestoMapper <i>Input:</i> valid zoom setting <i>Input:</i> display filter setting <i>Input:</i> current city	A map with restaurant icons on it based on a criteria set by the display filter and zoom setting
GRMC-Alternative Flow: <i>Invalid zoom setting</i>	<i>Precondition:</i> Run RestoMapper <i>Input:</i> invalid zoom setting <i>Input:</i> display filter setting <i>Input:</i> current city	Popup message displayed: “Invalid zoom setting, please reduce your zoom setting”

5.5.1.2 Robustness Analysis (Phase 2)

After creating the HLATs, robustness analysis is performed to retrieve object level information to create EATs. It is beneficial, though not mandatory, to decompose a flow into steps. There does not exist systematic method to decompose a flow into steps, therefore this task requires human judgment. For the “Generate Restaurant Map for City” UC, three steps are identified in the Basic Flow, and two in the Alternative Flow. Appendix F contains a demonstration of how every step is traced through its corresponding robustness diagram to determine the objects involved. The inputs, outputs and their representative objects, identified from performing robustness analysis are shown in Tables 5-8 and 5-9 for the Basic and Alternative Flows, respectively.

⁶ GRMC is an abbreviation standing for the *Generate Restaurant Map for City UC* and it is used to create a unique test ID. Other test ID abbreviations are based on the name of the belonging UC.

Basic Flow:

1. Get Restaurants for Current City with Valid Zoom Setting.
2. Get Map.
3. Add Restaurant Icons to Map.

Table 5-8: Results of performing robustness analysis on the Basic Flow

Flow Step	Input/Output	Element or Action	Representative Objects
1	Input	Current City	MapView
1	Input	Zoom setting	MapView
3	Input	Display filter settings	DisplayFilter
3	Output	Edited Map entity object containing restaurant icons	Map

Alternative Flow: Invalid zoom setting

1. Generate Map for Current City with Invalid Zoom Setting.
2. Display pop-up.

Table 5-9: Results of performing robustness analysis on the Alternative Flow

Flow Step	Input/Output	Element or Action	Representative Objects
1	Input	Current City	MapView
1	Input	Out of scale zoom setting	MapView
1	Output	Invalid zoom popup	InvalidZoomPopup

5.5.1.3 Creating Executable Acceptance Tests (Phase 3)

Object-level information retrieved from robustness analysis is used to create EATs that correspond to the previously developed HLATs. As mentioned earlier, the EATs for the Basic Flow (Figure 5-14) and the Alternative Flow (Figure 5-15) are formatted according to the FIT/FITness syntax. For presentation purposes, only the EATs for the Basic Flow will be shown using UCAT. The remaining EATs in this Section and Appendix G will be shown as fixture tables only.

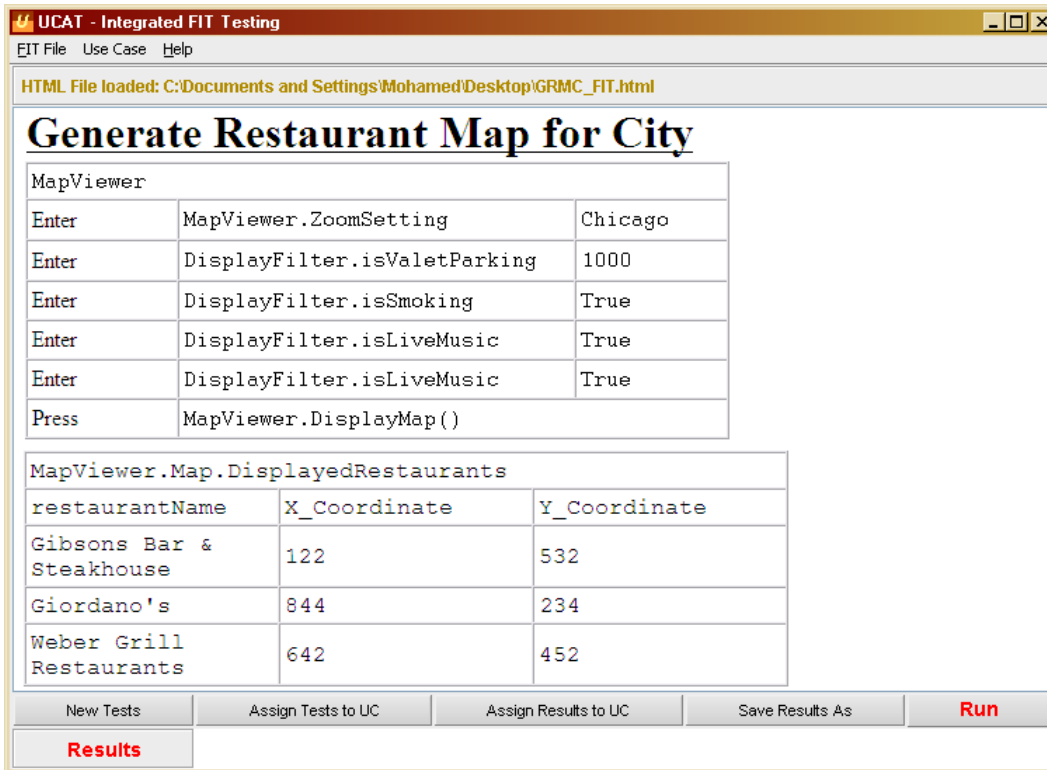


Figure 5-14: EAT of the Basic Flow

MapView		
Enter	MapView.CurrentCity	<i>Current City</i>
Enter	MapView.ZoomSetting	<i>Valid Zoom Setting</i>
Enter	DisplayFilter.isValetParking	<i>True/False</i>
Enter	DisplayFilter.isSmoking	<i>True/False</i>
Enter	DisplayFilter.isLiveMusic	<i>True/False</i>
Press	MapView.DisplayMap()	
Check	InvalidZoomPopup.isDisplay()	True
MapView.Map.DisplayedRestaurants		
restaurantName	X_Coordinate	Y_Coordinate

Figure 5-15: EAT of the Alternative Flow: Invalid Zoom Setting

The EAT corresponding to the *Basic Flow* consists of two fixtures:

- an *ActionFixture* to input the required data (the current city, zoom setting and display settings); and
- a *RowFixture* to examine that the Map is displayed the expected set of restaurants.

Meanwhile, the EAT for the Alternative flow consists of an *ActionFixture* to input the required data and to check that the `InvalidZoomPopup` has been displayed. The EAT also includes a *RowFixture* to examine that the Map does not contain any restaurants. Note that this fixture does not contain any elements because no restaurants are expected to be displayed.

Data values shown in EATs throughout this case study are presented in *italics* for the purposes of abstraction and generality. To make the tests executable, abstract data values are replaced with concrete ones. For example, in Figure 5-15, *Current City* can be substituted with the values of Chicago, New York, etc...

5.5.2. Efficacy of the Developed Acceptance Tests

The efficacy of the EATs developed is dependent on the quality of the UC model at hand. Acceptance testing is a validation process that the right system is being built, which ideally should be correctly defined in the UC model. The set of EATs created are based on analyzing the UC model of the RestoMapper system in its current form. The proposed approach is based on extracting all usage scenarios (flows) described in UC descriptions. The usage scenarios were used to create the

set of EATs shown throughout this case study, to cover the functionality described by usage scenarios. Table 5-10 shows the usage scenarios encompassed by the two UCs analyzed in this case study and their corresponding set of EATs.

Table 5-10: Coverage provided by the created EATs

Use Case	Flow	EAT
Generate Restaurant Map for City	Basic Flow	EAT shown in Figure 5-13
	Alternative Flow: Invalid zoom setting	EAT shown in Figure 5-14
Display Rollover Information	Basic Flow	EAT shown in Figure G.2 (Appendix G)
	Alternative Flow: Clicked on coordinates with multiple restaurants	EAT shown in Figure G.3 (Appendix G)
	Alternative Flow: Clicked on coordinates with no restaurants	EAT shown in Figure G.4 (Appendix G)

5.6. Role of the Developed Acceptance Tests

The technique presented in this Chapter is designed to develop a comprehensive set of acceptance tests for large-scale software development projects, in particular, those that utilize the V-Model development process. While the process of developing acceptance tests within a V-Model development process guides the

system design, the developed acceptance tests serve two additional purposes (see Figure 5-16).

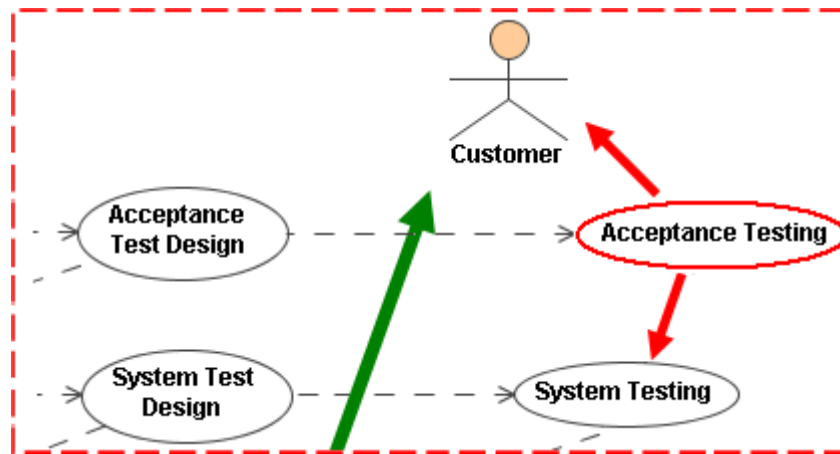


Figure 5-16: Subsection of the V-Model development process emphasizing the role of acceptance tests

Firstly, acceptance tests are used as basis for developing systems, which in turn guides the development of integration and unit tests. Secondly, after developing the intended system, acceptance tests are executed to demonstrate to the customer how the system will behave under various scenarios and that the system does indeed generate the expected output in each of those scenarios. From a contractual end, the customer uses the acceptance tests to determine whether to accept the system or not.

Chapter 6

Conclusions

6.1. Summary

UC modeling is a very powerful requirements modeling tool, providing great flexibility for requirements engineers to capture the behavioral essence of the target system. In a UC driven development process, UC models are used to create other UML artifacts leading to the eventual implementation of the target system. Thus poorly constructed UC models may yield many inconsistencies between subsequent UML artifacts, ultimately leading to many defects in the eventual code. Unfortunately, UC modeling is often misapplied resulting in significant numbers of defects. Hence, it is essential to produce high quality UC models.

A quality use case model improves every aspect of the development cycle. There are several quality attributes that should exist in every use case model. A use case model needs to be precise and unambiguous so that all stakeholders would have a common understanding of the capabilities and constraints of the system. A use case model needs to be analytical and should not contain any assumptions about the design or implementation. An analytical use case model should only describe what a system should do. Another essential quality attribute is consistency. Many researchers and practitioners warn about the harmful consequences of inconsistencies in use case models. Inconsistencies can negatively affect every aspect of the development cycle as well as the

stakeholders. Relying on heuristics and experience to manually detect inconsistencies can be cumbersome, error prone and requires a great deal of expertise to be effective. Such expertise is often not readily available.

6.2. Contributions and Results

Several major contributions have been made to tackle the above mentioned issues.

6.2.1. Improving the Understandability of Functional Requirements with AGADUC

In this thesis, a technique named AGADUC was developed that will automatically produce UCADs (which contain UML activity diagrams) that adheres to the syntax rules and notation standard of UML activity diagrams. UCADs provide a visual representation of the functional requirements embedded within the UC descriptions, allowing its stakeholders to gain an accurate understanding of the flow of interactions and scenarios that take place. This technique is supported by the tool AREUCD and resulted in the development of the SUCD structure which paved the way to future work that will tackle the issue of inconsistencies in UC models.

6.2.2. Reducing Inconsistencies with SSUCD

The potential for reducing inconsistencies in UC models resulted in the introduction of the structure SSUCD. SSUCD serves as a guideline to UC authors. The SSUCD structure along with the REUCD process enables the systematic

generation of use case diagrams and ensures consistency between the descriptions and their diagrams. The generated diagrams will be complete and provide an accurate representation of the use case descriptions. This process is automated by SAREUCD. The REUCD process may also be reversed, where the use case diagram is constructed before the use case descriptions. In that case, SAREUCD can automatically generate use case description ‘skeletons’ from use case diagrams. Analysts will then need to manually fill in the details of each use case description. After filling in the details, SAREUCD can detect any inconsistencies between the diagrams and the descriptions and notify the analysts about these inconsistencies.

This thesis presents a subject-based controlled experiment which explores a number of research questions. The first research question posed by this experiment was to evaluate the effectiveness of using SSUCD to improve the consistency level in UC models versus the use of traditional UNL. Consistency solely is a highly sought after quality attribute in UC models. Another research question posed by this experiment was to evaluate the impact of improving consistency on other UC modeling quality attributes.

This experiment was conducted in the context a voluntary mini-course which involved graduate students as subjects. Subjects applied both treatments (SSUCD and UNL) to two distinct systems. The results of this experiment showed that when SSUCD was utilized to develop both given systems, a statistically significant improvement was achieved with respect to the consistency level of the developed UC models. A statistically significant improvement was also observed

when using SSUCD with respect to the completeness and understandability levels in one of the two systems. As far as the other quality attributes are concerned; there was no statistical significance observed to support any possible conjecture.

As predicted, all subjects finished their exercises in approximately 1 hour (± 15 minutes). There was no significant difference between the times the SSUCD and UNL subjects required to finish their exercises. Informal post-interviews revealed that the subjects generally did not perceive SSUCD as an additional burden to their authoring efforts, and that it was quite simple to “learn and apply”.

6.2.3. Using Antipatterns to improve the Quality UC models

In this thesis, a technique based on antipatterns that helps improve the quality of UC models was devised. The application of the technique does not require any artifacts in addition to UC models, this allows the technique to be applied early in the development cycle, where other design artifacts are usually unavailable and the cost of removing defects is minimized. Given the “informality” of UC models, many approaches provide abstract guidelines towards improving UC models. Using antipatterns provides analysts with a more systematic approach to improve UC models, significantly reducing the dependency on skill and experience. A large repository of antipatterns was developed to guide analysts in improving their UC models. The repository contains 26 domain-independent antipatterns that can be applied to any UC model. The majority of the developed antipatterns benefit from (semi-)automation support to increase the accuracy and speed of their detection. In addition to the provided antipatterns, a framework was developed for

analysts to create their customized antipatterns based on a simplified UC modeling metamodel, where analysts can create their own antipattern descriptions using OCL. The complexity of the metamodel was intentionally designed to encourage its adoption by analysts and minimize the requisite learning curve, while supporting the basic notational subset of UC models. Automation support for detecting antipatterns is provided via the tool ARBIUM.. ARBIUM provides (semi-) automated support for 23 antipatterns presented in the repository and allows analysts to define their own antipatterns.

The effectiveness of the approach was demonstrated upon the MAPSTEDI system. Before applying the proposed process, the MAPSTEDI UC model suffered from a number of quality degradation issues. Most issues (antipatterns) were detected automatically using ARBIUM. Most antipattern matches addressed resulted in changes; however, there were also a small number of antipattern matches that are considered false positives. This indicates that real-world UC models are highly vulnerable to poor modeling habits and design decisions and often require improvements. Many of these improvements were critical as they improved the correctness and consistency of the UC models. Others enhanced the understandability of the UC models and made them more analytical. The antipattern matches revealed the issues that existed in the original UC model that had been overlooked. The issues were addressed and resolved accordingly, resulting in a higher quality UC model.

6.2.4. Producing Acceptance Tests from UC Models

The production of high quality UC models would be useless unless they were better used to improve the Software Development process. A process is presented that utilizes UC models as a basis for developing acceptance tests. The process also utilizes domain models and robustness diagrams to aid and guide analysts in developing a set of acceptance tests that cover a system's common usage scenarios, including unsuccessful scenarios. Acceptance tests are developed for each UC individually then a more comprehensive set of acceptance tests are developed that cover the functionality provided by multiple UCs. In the featured case study, nine *individual* HLATs and EATs as well as four *multiple* HLATs and EATs were developed using only three narrated UCs, their respective available robustness diagrams and a domain model.

The benefits of creating HLATs extend beyond the ability to create EATs. The customer can use HLATs for the high-level validation of the requirements. HLATs are more readable than EATs since they are composed of natural language statements. HLATs can be quickly developed without the need to wait for object level information to become available through robustness analysis.

UC models adhere to a relatively small set of syntax rules. UC descriptions are mainly comprised of UNL. It is naturally very difficult to devise systematic technical solutions that are heavily dependent on analyzing UC models. Human judgment is hence required while applying the proposed approach. Within our proposed approach, human judgment is required breakdown basic and alternative flows into scenarios, and to breakdown scenarios into steps. Human judgment is

also required to trace steps from a scenario in a robustness diagram. Furthermore, the quality of the UC model, which is the basis for our approach, is dependent on human skill and experience.

6.3. Future Work

The research presented in this thesis can be extended in several ways. This Section outlines potential future work based on each approach presented in the thesis.

6.3.1. Future Work Based on Structured UCs

To begin with, future work can be directed towards improving the SUCD structure and the AGADUC process to allow analysts to describe more complex workflows. The AGADUC process can be incorporated in leading UML modeling tools. The tool AREUCD can be upgraded to generate XML files showing complete UCADs that can be displayed on various UML modeling tools. However, this is dependent on UML modeling tools providing adequate support for the notation required to construct UCADs. Other techniques can be developed to systematically generate other types of UML artifacts, such as sequence diagrams, from UC models, thus minimizing the error injections by humans when constructing these artifacts. Future work can also be directed towards developing a semi-systematic approach that will convert SSUCD use cases into SUCD use cases (El-Attar et al. 2006). The approach will require personnel with technical expertise to carry out.

6.3.2. Future Work Based Using Antipatterns

Future work will initially be based around improving the usability (e.g. the incorporation diagrammatic construct drawing package) of ARBIUM with respect to the construction of new domain-specific anti-patterns. ARBIUM can also be upgraded to perform limited textual analysis, making use of any structure that may exist in UC descriptions, such as the actual template. Another beneficial upgrade to ARBIUM is the implementation of transformation rules written in a model transformation language such as QVT (QVT 2002) (Queries/Views/Transformation) to formalize and automate changes applied to UC diagrams.

Other future work can be directed towards creating a hierarchy of antipatterns. The hierarchy will act as an antipatterns matching strategy for analysts to apply the proposed technique more efficiently. Analysts will be able to determine which antipatterns to look for first and when to start a new iteration. This will help reduce the effort and time required to apply the technique. The antipatterns matching strategy may then be implemented in ARBIUM to further automate the technique and reduce the analyst's workload.

Finally, it will be beneficial to improve the UC modeling notation in order to prevent the occurrence of many antipatterns. For example, while analyzing a large number of UC models and applying the proposed technique, it was discovered that many antipatterns matches existed due to a notational limitation in UC modeling. The *extend* relationship is used to model both exceptional behavior and optional behavior. One of the greatest advantages of UC modeling is that it

contains a small notational set, allowing its ease of use. However, it may be advantageous to introduce two additional relationships that explicitly represent optional and exceptional behavior separately.

6.3.3. Future Work Based Developing Acceptance Tests from UC Models

The success and effectiveness of the approach presented in Chapter 5 is dependent on the experience level of the analyst(s) applying it. Therefore, future work can be directed towards modifying the approach to become more systematic by utilizing the limited formality provided by the SSUCD (El-Attar et al. 2006a) structure to create acceptance tests. SAREUCD, which already supports SSUCD, can be further upgraded to automate (or at least partially automate) the creation of EATs based on UCs described in SSUCD structure.

Bibliography

- [Adolph et al. 2002] S. Adolph and P. Bramble, *Patterns for effective use cases*. Addison-Wiley, 2002.
- [Agarwal et al. 2002] R. Agarwal and V. Venkatesh, “Assessing a firm’s web presence: A heuristic evaluation procedure for the measurement of usability,” *Inform. Syst. Res.*, vol. 13, no. 2, pp. 168–186, June 2002.
- [Ambler 2007] S. Ambler, “When is a Model Agile,” [Online]. Available: <http://www.agilemodeling.com/essays/whenIsAModelAgile.htm>. [Accessed: Nov. 2007].
- [Analyst Pro 2008] Goda Software, *Analyst Pro*, ver. 6.0., [Online]. Available: www.analysttool.com.
- [Anda et al. 2002] B. Anda and D. I. K. Sjøberg, “Towards an Inspection Technique for Use Case Models,” *Proc. 14th Int’l Conf. on Software Eng. and Knowledge Eng.*, 2002, pp. 127-134.
- [Anda et al. 2001a] B. Anda, D. Sjøberg, and M. Jørgensen, “Quality and Understandability in Use Case Models,” *Proc. 15th European Conf. Object-Oriented Programming*, J. Lindskov Knudsen, ed., 2001, pp. 402-428.
- [Anda et al. 2001b] B. Anda, H. Dreiem, D. Sjøberg, and M. Jørgensen, “Estimating Software Development Effort Based on Use Cases –

Experiences from industry,” *Fourth Int’l Conf. on the Unified Modeling Language*, 2001.

[Anderson et al. 2001] E., Anderson, M. Bradley, and R., Brinko, “Use case and business rules: styles of documenting business rules in use cases,” Addendum to the Object-oriented programming, systems, languages, and applications conference. 1997.

[Arisholm et al. 2006] E. Arisholm, L. Briand, S. Hove, and Y. Labiche, “The Impact of UML Documentation on Software Maintenance: An Experimental Evaluation,” *IEEE Transaction on Software Engineering*, vol. 32, pp. 365-381, 2006.

[Arisholm et al. 2003] E. Arisholm and D. Sjøberg, “A Controlled Experiment with Professionals to Evaluate the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software,” Simula Research Laboratory, Tech. Rep. 2003-6, 2003.

[Armour et al. 2000] F. Armour and G. Miller, *Advanced Use Case Modeling*. Addison-Wiley, 2000.

[BABOK 2009] International Institute of Business Analysts, “Business Analysts Body of Knowledge,” *International Institute of Business Analysts*, Version 1.6, 2009. [Online]. Available: http://www.theiiba.org/AM/Template.cfm?Section=Body_of_Knowledge. [Accessed February 2009].

[Basanieri et al. 2002] F. Basanieri, A. Bertolino, and E. Marchetti, “The Cow_Suite Approach to Planning and Deriving Test Suites in UML

- Projects,” *Proc. Fifth Int’l Conf. UML: Model Eng. Languages and Concepts and Tools*, 2002, pp. 383-397.
- [Beck 1999] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison Wiley Professional, 1999.
- [Berenbach 2004] B. Berenbach, “The Evaluation of Large, Complex UML Analysis and Design Models,” in *Proc. 26th Int’l Conf. on Software Eng.*, pp. 2004, pp. 232-241.
- [Belgamo et al. 2005] A. Belgamo, S. Fabbri, and J. C. Maldonado, “TUCCA: improving the effectiveness of use case construction and requirement analysis,” *Int’l Symp. on Empirical Soft. Eng.*, 2005.
- [Ben Achour et al. 1999] C. Ben Achour, C. Rolland, N. A. M. Maiden, and C. Souveyet, “Guiding Use Case Authoring: Results of an Empirical Study,” *Proc. IEEE Symp. on Requirements Eng.*, 1999.
- [Betty] B. H. C. Cheng and J. M. Atlee, “Research Directions in Requirements Engineering,” *Future of Software Engineering*, 2007, pp. 285-303.
- [Biddle et al. 2002] B. Biddle, J. Noble, and E. Tempero, “Essential Use Cases and Responsibility in Object-Oriented Development,” *Proc. of 25th CRPITS*, 2002, vol. 24, issue 1.
- [Bittner et al. 2002] K. Bittner and I. Spence, *Use Case Modeling*. Addison-Wiley, 2002.
- [Boehm] B. Boehm, *Software Engineering Prentice - Economics*. Hall, Englewood Cliffs, 1981.

- [Booch et al. 2005] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide Second Edition*. Addison-Wiley, 2005.
- [Briand et al. 2002] L. Briand and Y. Labiche, "A UML-Based Approach to System Testing," *J. Software and Systems Modeling*, pp. 10-42, 2002.
- [Butler et al. 2002] G. Butler and L. Xu, "Cascaded Refactoring for Framework Evolution," *Proc. Symp. on Soft. Reusability*, ACM Press, pp. 51-57, 2001.
- [CancerGrid 2008] "CancerGrid Tissue Tracking Database System," [Online]. Available:
<http://www.cancergrid.org/public/documents/2006/mrc/Report%20MRC-1.2.2.1%20Tissue%20tracking%20database%20requirements.pdf>.
[Accessed July 2008].
- [Chandrasekaran 2008] P. Chandrasekaran, "How use case modeling policies have affected the success of various projects (or how to improve use case modeling)," *Addendum to the Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 1997.
- [Cliff 1993] N. Cliff, "Dominance statistics: Ordinal analyses To Answer Ordinal Questions," *Psychological Bulletin*, vol. 114, pp. 494-509, 1993.
- [Cliff 1996] N. Cliff, "Answering Ordinal Questions With Ordinal Data Using Ordinal Statistics," *Multivariate Behavioral Research*, vol. 31, pp. 331-350, 1996.
- [Cliff 1996b] N. Cliff, *Ordinal Methods for Behavioral Data Analysis*. Lawrence Erlbaum Associates, 1996.

- [Cockburn 1995] A. Cockburn, "Structuring Use Cases with Goals," Tech. Rep. Human and Tech., 7691 Dell Rd, Salt Lake City, UT 84121, HaT.TR.95.1, <http://members.aol.com/acockburn/papers/usecaes.htm>, 1995.
- [Cockburn 2000] A. Cockburn, *Writing Effective Use Cases*. Addison-Wiley, 2000.
- [Cohn 2004] M. Cohn, *User Stories Applied: For Agile Software Development*. Addison Wiley, 2004.
- [Constantine et al. 1999] L. L. Constantine and L. A. D. Lockwood, *Software for Use. A Practical Guide to the Models and Methods for Usage-Centered Design*. Addison-Wiley, 1999.
- [Coplien 2007] J. Coplien, *Software Patterns*. SIGS, 1996.
- [El-Attar et al. 2006] M. El-Attar and J. Miller, "AGADUC: Towards a More Precise Presentation of Functional Requirement in Use Case Models," *Proc. 4th ACIS International Conference on Soft. Eng., Research, Management & Applications*, 2006.
- [El-Attar et al. 2006a] M. El-Attar and J. Miller, "Producing Robust Use Case Diagrams via Reverse Engineering of Use Case Descriptions," *Journal of Software and Systems Modeling*, vol. 7, no.1 , pp. 67-83 , 2006.
- [Fabbrini et al. 2001] F. Fabbrini, M. Fusani, S. Gnesi, and G. Lami, "The Linguistic Approach to the Natural Language Requirements Quality: Benefits of the Use of an Automatic Tool," in *Proc. 26th Annual NASA Goddard Software Workshop*, 2001, pp. 97 – 105.

- [Fagan 1976] M. E. Fagan, “Design and Code Inspections to Reduce Errors in Program Development,” *IBM Systems Journal*, vol. 15, no. 3, pp. 182-211, 1976.
- [Fantechi et al. 2002] A. Fantechi, S. Gnesi, G. Lami, and A. Maccari, “Application of Linguistic Techniques for Use Case analysis,” *Proc. of IEEE Joint Int’l Conf. on Requirements Eng.*, 2002, pp. 157–164.
- [FAIN 2008] “FAIN Active Network Enterprise Model UC model,” [Online]. Available: www.ee.ucl.ac.uk/lcs/papers2000/lcs026.pdf. [Accessed July 2008].
- [Firesmith 1999] D.G. Firesmith, “Use Case Modeling Guidelines,” *Proc. of Tech. of Object-Oriented Languages and Systems*, 1999.
- [Gilb et al. 1993] T. Gilb and D. Graham, *Software Inspection*. Addison-Wiley, Reading, 1993.
- [Glass et al. 1993] R. L. Glass, I. Vessey, and V. Ramesh, “Research in Software Engineering: An Analysis of the Literature,” *J. Information and Software Technology*, vol. 44, no. 8, pp. 491-506, 2002.
- [Gogolla et al. 2002] M. Gogolla, J. Bohling, and M. Richters, “Validation of UML and OCL Models by Automatic Snapshot Generation,” *Proc. 6th Int. Conf. on the Unified Modeling Language*, 2003.
- [Gomaa 2002] H. Gomaa, *Designing Software Product Lines with UML*. Addison Wiley Professional, 2004.
- [Gomaa 2000] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison Wiley, 2000.

- [Gomaa 1997] H. Gomaa, “Use Cases for Distributed Real-Time Software Architectures,” in *Proc. of the Joint Workshop on Parallel and Distributed Real-Time Systems*, 1997, pp. 34–42.
- [Good et al. 1989] M. Good, T. M. Spine, J. Whiteside, and P. George, “User-derived impact analysis as a tool for usability engineering. in *Proc. CHI’86 Human Factors in Computing Systems*, 1986, pp. 241–246.
- [Goodhue et al. 1959] D. L. Goodhue and R. L. Thompson, “Task-technology fit and individual performance,” *MIS Quart.*, vol. 19, no. 2, pp. 213–236, June 1995.
- [Gould et al. 1991] J. D. Gould, S. J. Boies, and C. Lewis, “Making usable, useful, productivity- enhancing computer applications,” *Comm. ACM*, vol. 34, pp. 74–85, 1991.
- [Gould et al. 1983] J. D. Gould, J. Conti, and T. Hovanyecz, “Composing letters with a simulated listening typewriter,” *Comm. ACM*, vol. 26, pp. 295–308, 1983.
- [Gould et al. 1985] J. D. Gould and C. Lewis, “Designing for usability: Key principles and what designers think,” *Comm. ACM*, vol. 28, pp. 300–311, 1985.
- [Harwood 1997] R. J. Harwood, “Use Case Formats: Requirements, Analysis, and Design,” *Journal of Object-Oriented Programming*, vol. 9, pp. 54-57, Jan. 1997.
- [Hood et al. 2007] C. Hood, S. Wiedeman , S. Fichtinger, and U. Pautz, *Requirements Management: The Interface Between Requirements*

Development and All Other Systems Engineering Processes. Springer, 2007.

[Höst et al. 2000] M. Höst, B. Regnell, and C. Wohlin, “Using Students as Subjects – A Comparative Study of Students and Professionals in Lead-Time Impact Assessment,” *Empirical Software Engineering*, vol. 5, pp. 210-214, Nov. 2000.

[Hess et al. 2005] M. R. Hess, J. D. Kromrey, J. M. Ferron, K. Y. Hogarty, and C. V. Hines, “Robust Inference in Meta-Analysis: An Empirical Comparison of Point and Interval Estimates Using the Standardized Mean Difference and Cliff’s Delta,” *Annual meeting of the American Educational Research Association*, pp. 36.

[Jaaksi 1998] A. Jaaksi, “Our Cases with Use Cases,” *Journal of Object-Oriented Programming*, vol. 10, pp. 58-64, Feb. 1998.

[Jacobson et al. 1995] I. Jacobson, M. Ericsson, and A. Jacobson, *The Object Advantage*. ACM Press, 1995.

[Jacobson et al. 1992] I. Jacobson, *Object-Oriented Software Engineering. A Use Case Driven Approach*. Addison-Wiley, 1992.

[Johansson 2004] A. Johansson, “Confusion in Writing Use Cases,” *Proc. of the 2nd Int’l Conf. on Information Tech. for Application*, 2004.

[Kaner et al. 2003] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing*. John Wiley & Sons, 2003.

- [Kitchenham 2004] B. Kitchenham, "Procedures for Performing Systematic Reviews," Tech. Rep. TR/SE0401, Keele University and Tech. Rep. 0400011T.1, National ICT Australia Ltd., 2004.
- [Kroll et al. 2003] P. Kroll and P. Kruchten, *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*. Addison-Wiley, 2003.
- [Kromrey et al. 2005] J. Kromrey, K. Hogarty, J. Ferron, C. Hines, and M. Hess, "Robustness in Meta-analysis: An Empirical Comparison of Point and Interval Estimates of Standardized Mean Differences and Cliff's Delta," *American Statistical Association 2005 Joint Statistical Meetings*, 2005, pp.7.
- [Kromrey et al. 1998] J. Kromrey and K. Hogarty, "Analysis Options For Testing Group Differences On Ordered Categorical Variables: An Empirical Investigation Of Type 1 Error Control And Statistical Power," *Multiple Linear Regression Viewpoints*, vol. 25, pp. 70–82, 1998.
- [Kruchten 1998] P. Kruchten, *The Rational Unified Process: An Introduction*, 2nd ed. Addison-Wiley Longman Inc., 1999.
- [Kulak et al. 2000] A. Kulak and E. Guiney, *Use Cases: Requirements in Context*. Addison-Wiley, 2000.
- [Larman 2001] C. Larman, *Applying UML Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2nd ed. Prentice Hall, 2001.
- [Leffingwell et al. 2000] D. Leffingwell and D. Widrig, *Managing Software Requirements: A Unified Approach*. Addison-Wiley, 2000.

- [Lehmann 1998] E. L. Lehmann, *Non-Parametrics: Statistical Methods Based On Ranks, Revised*. Pearson, 1998.
- [Lilly 1990] S. Lilly, “Use Case Pitfalls: Top 10 Problems from Real Projects Using Use Cases,” *Proc. of Technology of Object-Oriented Languages and Systems*, 1999.
- [MAPSTEDI 2008] “MAPSTEDI UC Model,” [Online]. Available: mapstedi.colorado.edu/documents/Mapstedi_High_Level_Use_Case_Model.pdf. [Accessed July 2008].
- [Mantei al. 1988] M. M. Mantei and T. J. Teorey, “Cost/benefit analysis for incorporating human factors in the software lifecycle,” *Comm. ACM*, vol. 31, pp. 428–439, 1988.
- [Mattingly al. 1988] L. Mattingly and H. Rao, “Writing Effective Use Cases and Introducing Collaboration Cases,” *Journal of Object-Oriented Programming*, vol. 11, pp. 77-84, Oct.1998.
- [McBeen 2007] P. McBeen, “Use Case Inspection List,” [Online]. Available: www.mcbreen.ab.ca/papers/QAUseCases.html. [Accessed Nov. 2007].
- [McCoy 2003] J. McCoy, “Requirements use case tool (RUT),” *Companion of the 18th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2003, pp. 104–105.
- [Medvidovic al. 2002] N. Medvidovic, D. Rosenblum, D. Redmiles, and J. Robbins, “Modeling software architectures in the Unified Modeling Language,” *ACM Transactions on Software Engineering and Methodology*, vol. 11, pp. 2-57, 2002.

- [Mian al. 2005] P. Mian, T. Conte, A. Natali, J. Biolchini, E. Mendes, and G. Travassos, "Lessons Learned on Applying Systematic Reviews to Software Engineering," *Proc. of the 3rd International Workshop 'Guidelines For Empirical Work' in the Workshop Series on Empirical Software Engineering*, 2005, pp 81-95.
- [Mugridge al. 2005] R. Mugridge and W. Cunningham, *Fit for Developing Software: Framework for Integrated Tests*. Prentice Hall, 2005.
- [Nebut al. 2006] C. Nebut, F. Fleurey, Y. Le Traon, and J. Jézéquel, "Automatic Test Generation: A Use Case Driven Approach," *IEEE Trans. on Soft. Eng.*, vol. 32, no. 3, pp. 140-155, March 2006.
- [OMG 2005] Object Management Group, "UML Superstructure Specification," *Object Management Group*, Dec. 2005, ver. 2.0 formal/05-07-04, 2005. [Online]. Available: <http://www.omg.org/docs/formal/05-07-04.pdf>. [Accessed: Dec. 2008].
- [QVT 2002] Object Management Group, "MOF 2.0 Query/Views/Transformations RFP," *Object Management Group*, Dec. 2002. [Online]. Available: <http://www.omg.org/cgi-bin/apps/doc?ad/05-03-02.pdf>. [Accessed: Dec. 2008].
- [Optimal Trace 2008] Compuware, *Optimal Trace*, ver. 4.1., [Online]. Available: www.compuware.com/optimaltrace.
- [Overgaard et al. 2005] F. Overgaard and K. Palmkvist, *Use Cases Patterns and Blueprints*. Addison-Wiley, 2005.

- [Paige et al. 2000] R. F. Paige, J. S. Ostroff, and P. J. Brooke, "Principles for modeling language design," *Information & Software Technology*, vol. 42, no. 10, pp. 665-675, 2000.
- [Regnell et al. 1995] B. Regnell, M. Andersson, and J. Bergstrand, "A Hierarchical Use Case Model with Graphical Representation," *Proc. of Second IEEE Int'l Symp. on Requirements Eng.*, 1995, pp. 270.
- [Ren et al. 2003] S. Ren, K. Rui, and G. Butler, "Refactoring the Scenario Specification: a Message Sequence Chart Approach," in *Proc. of 9th Object-Oriented Information Systems*, 2003, pp. 294-298.
- [Ren et al. 2004] S. Ren, G. Butler, K. Rui, J. Xu, W. Yu, and R. Luo, "A Prototype Tool for Use Case Refactoring," *Proc. of the 6th Int'l Conf. on Enterprise Information Systems*, 2004, pp. 173-178.
- [Rosenberg et al. 2007] D. Rosenberg and S. Kendall, "Top Ten Use Case Mistakes," [Online]. Available: <http://www.sdmagazine.com/documents/s=815/sdm0102c/>. [Accessed Nov. 2007].
- [Rosenberg et al. 1999] D. Rosenberg and K. Scott, *Use Case Driven Object Modeling with UML*. Addison-Wiley, 1999.
- [Rosenberg et al. 2005] D. Rosenberg, M. Stephens, and M. Collins-Cope, *Agile Development with ICONIX Process: People, Process, and Pragmatism*. Apress, 2005.

- [Rosson et al. 1987] M. B. Rosson, S. Maass, and W. A. Kellogg, “Designing for designers: An analysis of design practice in the real world,” *Proc. CHI + GI’87 Conf.*, 1987, pp. 137–141.
- [Rui et al. 2003] K. Rui and G. Butler, “Refactoring Use Case Models: The Metamodel,” in *Proc. of 25th Computer Science Conf.*, M. Oudshoorn ed., 2003, pp. 4-7.
- [Ryndina et al. 2004] O. Ryndina and P. Kritzinger, “Improving Requirements Specification: Verification of UC Models with Susan,” Tech. Rep. CS04-06-00, Department of Computer Science, University of Cape Town, 2004.
- [Ryser et al. 1999] J. Ryser and M. Glinz, “A Scenario-Based Approach to Validating and Testing Software Systems Using Statecharts,” *Proc. 12 Int’l Conf. Software and Systems Eng. and Their Applications*. 1999.
- [Sauvé et al. 2006] J. P. Sauvé, Osório L. A. Neto, and W. Cirne, “EasyAccept: a tool to easily create, run and drive development with automated acceptance tests,” *Proc. of the 2006 Int’l Workshop on Automation of Software Test*, pp. 111-117, 2006.
- [Schneider et al. 1998] G. Schneider and J. Winters, *Applying Use Cases – A Practical Guide*. Addison-Wiley, 1998.
- [Selenium 2008] “Selenium Reference Documentation,” [Online]. Available: <http://www.openqa.org/selenium-core/documentation.html>. [Accessed Jan. 21, 2008].

- [Shapiro et al. 1972] S. S. Shapiro and M. B. Wilk, “An Analysis of Variance Test for the Exponential Distribution,” *TechnoMetrics*, vol. 14, pp. 355-370, 1972.
- [Siegel et al. 1988] S. Siegel and N. J. Castellan Jr., *Non-parametric Statistics for the Behavioral Sciences*, 2nd ed. McGraw-Hill, 1988.
- [Sommerville 1996] I. Sommerville, *Software Engineering*, 5th ed. Addison-Wiley, 1996.
- [STEAM 2009] STEAM Laboratory website, University of Alberta, “Simple Structured Use Case Descriptions,” [Online]. Available: http://www.steam.ualberta.ca/main/research_areas/SSUCD.htm. [Accessed Jan. 2009].
- [STEAM 2009b] STEAM Laboratory website at the University of Alberta, “SUCD Formal Syntax,” [Online]. Available: http://www.steam.ualberta.ca/main/research_areas/Requirements_Capture.htm. [Accessed Jan. 2009].
- [STEAM 2009c] STEAM Laboratory website, University of Alberta, “Use Case Modeling Antipatterns,” [Online]. Available: http://www.steam.ualberta.ca/main/research_areas/Use%20Case%20Antipatterns%20Website.htm. [Accessed Jan. 2009].
- [SCM 2008] “Supply Chain Management UC Model,” [Online]. Available: [ws-i.org/SampleApplications/SupplyChainManagement/2002-11/SCMUseCases-0.18-WGD.pdf](http://www.ws-i.org/SampleApplications/SupplyChainManagement/2002-11/SCMUseCases-0.18-WGD.pdf). [Accessed July 2008].

- [TopTeam 2008] TechnoSolutions, *TopTeam Anaylst*, ver. 2.05., [Online].
Available: www.technosolutions.com.
- [Use Case Studio 2008] Rewritten Software, *Use Case Studio*, vers. 3.0., [Online].
Available: www.rewrittensoftware.com.
- [Warmer et al. 1998] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*. Addison-Wiley, 1998.
- [Wieggers 2003] K. Wieggers, *Software Requirements (Pro-Best Practices)*, 2nd ed. Microsoft Press, 2003.
- [Wirfs-Brock 1993] R. Wirfs-Brock, "Designing Scenarios: Making the Case for a Use Case Framework," *The Smalltalk Report*, vol. 3, no. 3, Nov.-Dec. 1993.
- [Wohlin et al. 1990] C. Wohlin and U. Korner, "Software Faults: Spreading, Detection and Costs," *Software Engineering Journal*, vol. 5, no. 1, pp. 33 – 42. 1990
- [Wohlin et al. 2000] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering - An Introduction*. Kluwer, 2000.
- [Woodall et al. 2006] P. Woodall and P. Brereton, "Conducting a Systematic Literature Review from the Perspective of a Ph.D. Researcher," *10th International Conference on Evaluation and Assessment in Software Engineering*, Keele University, UK. 2006.

[Xu] J. Xu, W. Yu, K. Rui, and G. Butler, "Use Case Refactoring: a Tool and a Case Study," in *Proc. 11th Asia Pacific Software Eng. Conf.*, 2004, pp. 484-491.

Appendix A

SUCD E-BNF

S ::= UseCaseDescription+ Actor+

Actor ::= Abstract? ActorName Implements? Specializes?

ActorName ::= CharactersAndOrDigits+

UseCaseDescription ::= NameSection
 BasicFlowSection?
 AlternativeFlowSection?
 Sub-flowsSection?
 ExtensionPointsSection?

NameSection ::= 'Use Case Name:'

Abstract?

UseCaseName

Implements?

Specializes?

Abstract ::= 'ABSTRACT'

Implements ::= 'IMPLEMENTS' UseCaseName

Specializes ::= 'SPECIALIZES' UseCaseName

BasicFlowSection ::= 'Basic Flow:'

'{BEGIN Use Case}'

Header*

'{END Use Case}'

Header ::= '{'BEGIN' HeaderName '}'

AfterStatement?

Contents*

ResumeStatement?

'{'END' HeaderName '}'

AlternativeFlowsSections ::= 'Alternative Flows:' **AF***

```

AF ::=  AtStatement
      IfStatement
      AFHeader

AFHeader ::=  '{BEGIN' HeaderName '}'
             Contents*
             ResumeStatement?
             '{END' HeaderName '}'

Sub-flowSection ::=  'Sub-flows: ' Sub-flow*

Sub-flow ::=  'SUB-FLOW' Sub-flowName
             Sub-flowHeader

Sub-flowHeader ::=  '{BEGIN' HeaderName '}'
                  Contents*
                  '{END' HeaderName '}'

Sub-flowName ::=  CharactersAndOrDigits+

ExtensionPointsSection ::=  'Extension Points: ' EP*

EP ::=  PREP | PUEP | PUEPDeclaration

PREP ::=  'PRIVATE EXTENSION POINT'
FlowStatement
AtStatement
         IfStatement
PREPHeader
EPHeader

PREPHeader ::=  '{BEGIN' HeaderName '}'
               Contents*
               ResumeStatement
               '{END' HeaderName '}'

PUEP ::=  'PUBLIC EXTENSION POINT BEHAVIOR'
FlowStatement
AtStatement
         IfStatement
         EPHeader

BaseUCName ::=  UseCaseName

HeaderInBaseUC ::=  HeaderName

EPHeader ::=  '{BEGIN' HeaderName '}'

```

Contents*

'{END' HeaderName '}'

ContinueStatement

PUEPDeclaration ::= 'PUBLIC EXTENSION POINT'

PublicExtensionPointName

ExtensionUCName ::= UseCaseName

HeaderInExtensionUC ::= HeaderName

Contents ::= Header | Statement

Statement ::= ('.' | Digit) (ActionStatement | PerformStatement | IncludeStatement)

ActionStatement ::= Actor '→' Action

Action ::= CharactersAndOrDigits+

FlowStatement ::= 'FLOW' FlowType

IncludeStatement ::= 'INCLUDE' UseCaseName

PerformStatement ::= 'PERFORM' Sub-flowName

ResumeStatement ::= 'RESUME' ('{'HeaderName'})+

AfterStatement ::= 'AFTER' ('{'HeaderName'})+

AtStatement ::= 'AT' {'HeaderName'} Statement*

IfStatement ::= 'IF' Condition

ContinueStatement ::= 'CONTINUE {' ReturnHeader '}'

UseCaseName ::= CharactersAndOrDigits+

HeaderName ::= CharactersAndOrDigits+

Condition ::= CharactersAndOrDigits+

FlowType ::= CharactersAndOrDigits+

ReturnHeader ::= CharactersAndOrDigits+

PublicExtensionPointName ::= CharactersAndOrDigits+

CharactersAndOrDigits ::= **Character** | **Digit**

Character ::= 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'

Digit ::= '0' | '1' | ... | '9'

Appendix B

Actor and UC Descriptions of the Library System Case Study

Actor:

Librarian

Brief Description:

This actor is an employee of the library

Actor:

Member

Brief Description:

This actor is a member of the Library who holds a membership cards

Use Case Name:

Authenticate Librarian

Brief Description:

This use case authenticates library staff to be able to perform administrative duties such as adding and removing books or enrolling members into the Library

Preconditions:

Basic Flow:

```
{BEGIN Use Case}
```

```
{BEGIN get librarian's username and password}
```

- Librarian -> Enters username
- Librarian -> Enters password
- SYSTEM-> Validates login information

```
{END get librarian's username and password}
```

{END Use Case}

Postconditions:

Login log is updated whether authentication was approved or not

Use Case Name:

Balance Overdue

Brief Description:

This use case is an extension UC that gets performed if a member attempts to borrow a book while there is a balance overdue on his/her account. The UC acts as a reminder to the Member to pay his overdue charges

Preconditions:

Extension Points:

PUBLIC EXTENSION POINT BEHAVIOR

EXTENDING {Borrow Book : Balance overdue}

FLOW Basic Flow

AT {bring book to borrow}

- Librarian -> Scans member's card

IF Member has an overdue balance

{BEGIN collect money}

- SYSTEM -> Notifies the Librarian that there is a balance overdue on that member's account

- Librarian -> Notifies the member that there is a balance overdue

{END collect money}

CONTINUE {authenticate librarian}

Special Requirements:

System must be online

Appendix C

SSUCD E-BNF

S ::= UseCaseDescription+ Actor+

Actor ::= Abstract? ActorName Implements? Specializes?

**UseCaseDescription ::= NameSection
 ExtendedSection?
 DescriptionSection?
 ExtensionPointsSection?**

**NameSection ::= 'Use Case Name:'
Abstract?
UseCaseName
Implements?
Specializes?**

Abstract ::= 'ABSTRACT'

Implements ::= 'IMPLEMENTS' UseCaseName

Specializes ::= 'SPECIALIZES' UseCaseName

**ExtendedSection ::= 'Extended Use Cases:'
 Extensions***

**Extensions ::= 'Base UC Name: ' UseCaseName
 ('Extension Point: ' EPName)?
 IfStatement?**

**DescriptionSection ::= 'Basic Flow:'
CharactersAndOrDigitsOrInclude***

**CharactersAndOrDigitsOrInclude ::= IncludeStatement |
CharactersAndOrDigits**

IncludeStatement ::= 'INCLUDE <' UseCaseName '>'

```
ExtensionPointsSection ::= 'Extension Points: '  
EPName*  
  
IfStatement ::= 'IF' Condition  
  
ActorName ::= CharactersAndOrDigits+  
  
UseCaseName ::= CharactersAndOrDigits+  
  
EPName ::= CharactersAndOrDigits+  
  
CharactersAndOrDigits ::= Character | Digit  
  
Character ::= 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'  
  
Digit ::= '0' | '1' | ... | '9'
```

Appendix D

Scoring UCs Developed in the Experiment

D.1. Scoring UCs from the Banking System developed in UNL

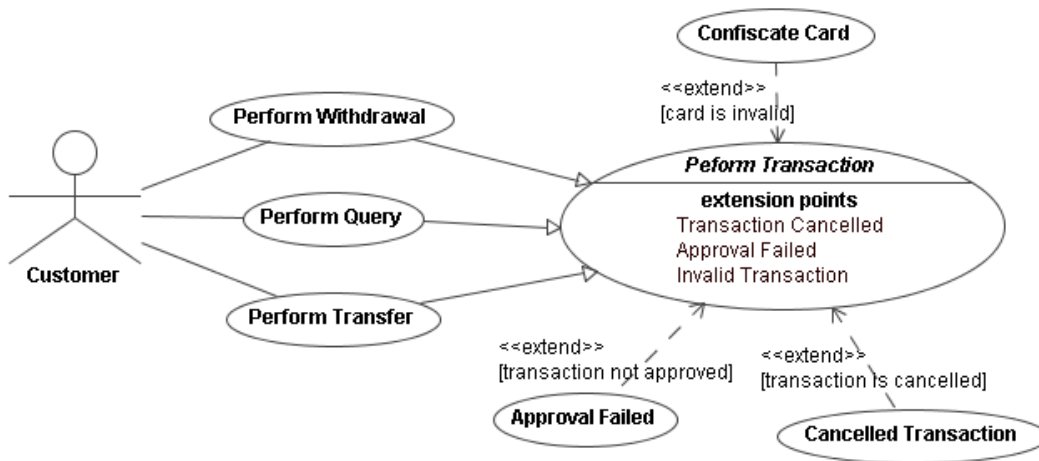


Figure D-1: Example Banking system UC diagram developed by a subject

Use Case Name: Perform Transaction

Preconditions: The customer must have an ATM card.

Basic Flow: The customer inserts their ATM card into the card reader and enters the correct PIN. The customer is given a selection of performing a withdrawal, query, or transfer. Once the customer chooses an option, the use case for the selected transaction type takes place. Once the child use case has completed, a receipt is printed and the card is ejected.

Alternative Flows:

- If the PIN is entered incorrectly three times or the card is lost or stolen, the Confiscate Card use case is carried out.
- If the customer cancels the transaction, the Cancelled Transaction use case takes place.
- If the transaction to be carried out by the child use case is not approved, the Approval Failed use case is carried out.

Use Case Name: Perform Withdrawal

Preconditions: Sufficient funds must be available in the account that is to have the money debited, the amount to be removed must not exceed the remaining daily limit, and there must be sufficient funds in the local cash dispenser.

Basic Flow: The customer specifies that amount of money to withdraw and the account to withdraw it from. The transaction is then approved and the requested amount of cash dispensed, and then control returns to the Perform Transaction use case.

Use Case Name: Approval Failed

Basic Flow: The customer is informed of the reason that the transaction was not approved, and any pending transactions are cancelled.

Scoring:

1. The “Perform Transaction” is depicted as an *abstract* UC in the diagram but this characteristic was not stated in the description. **(1 defect)**
2. The “Perform Transaction” is depicted to have two extension points both of which were not stated in the description. **(3 defects)**
3. The “Approval Failed” UC is depicted to extend the “Perform Transaction” but this fact was not stated in the “Approval Failed” UC. As an extension UC, it is the “Approval Failed” UC that is responsible for indicating when and under what clause will the exceptional it describes will be performed. Meanwhile, in an alternative flow contained in the “Perform Transaction” UC, it is mentioned that the “Approval Failed” UC is called upon to perform the required, which is a description of an *include* relationship between the UCs that was not depicted. These inconsistencies are a result of the same mistake and hence are scored as one defect. **(1 defect)**
4. The “Perform Withdrawal” UC states at the end of the Basic Flow that “control returns to the Perform Transaction use case” which is a description of a *include* relationship rather than an implementation relationship. **(1 defect)**

Inconsistency defects total = 6

D.2. Scoring UCs from the Airline Ticketing System developed in SSUCD

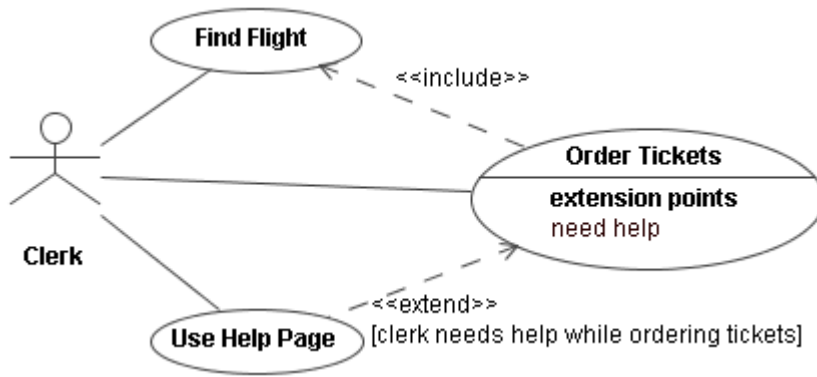


Figure D-2: Example Airline Ticketing system UC diagram developed by a subject

Use Case Name: Order Tickets

Associated Actors: Clerk

Description:

Basic Flow: Clerk enters the name of the traveler then enters the flight name which is INCLUDE <Find Flight> by the system based on the entered order information, and the books the tickets.

Sub Flows: Find Flight

Postconditions: The tickets have been ordered

Extension Points: need help

Use Case Name: Use Help Page

Associated Actors: Clerk

Description:

Basic Flow: The clerk invokes the help service and the order flight page is brought up.

Alternative Flows: If the Clerk invokes the help service outside of ordering tickets the main page is brought up.

Alternative Flows: If the Clerk invokes the help service outside of ordering tickets the main page is brought up.

Extended Use Cases:

Base UC Name: Order Tickets

Extension Point: need help

IF the clerk needs help while ordering a ticket

Scoring

1. For the “Order Tickets” UC:
 - The depicted *include* relationship with the “Find Flight” UC was correctly stated in the description.
 - The depicted extension point “need help” was correctly stated in the description.
 - The depicted association with the “Clerk” actor was correctly stated in the description.
 - A Sub-flow was stated which indicates that “Find Flight” is a sub routine described within the UC, which is incorrect since this behavior is described in the *inclusion* UC “Find Flight”. This results in a inconsistency within the UC description since the inclusion relationship was already stated. **(1 defect)**
2. For the “Use Help Page” UC:

- The depicted association with the “Clerk” actor was correctly stated in the description.
- The depicted *extend* relationship with the “Order Tickets” UC is correctly stated in the description.
- The depicted clause is correctly stated in the description.
- The description states the extension point (“need help”) at the “Order Tickets” UC where the additional behavior is inserted. This information should be depicted on the *extend* arrow but was not. **(1 defect)**

Inconsistency defects total = 2

Appendix E

Syntax for creating *ColumnFixtures* and *RowFixtures*

A *ColumnFixture* is most suitable for checking rules and calculations – basically input / output relationships in the absence of state information. For a given business logic, a series of input values are provided and the resulting outputs are checked accordingly. The fixture shown in Figure E-1 tests the return on investment calculation, which is used to test the behavior described by the “Calculate Investments” UC.

OneYearInvestment		
CurrentBalance	AccountType	ReturnOnInvestment()
10000	Savings	10500
5000	Savings	5250
1500	Chequeing	1515

Figure E-1: *ColumnFixture* example of calculating the return on one year investments. The name of the class under test is stated atop the fixture table, input values populate the left hand side columns, while output values populate the right hand side columns. The name of the input variable (or function) and the name of the output variable (or function) are stated atop their respective columns. It is assumed for the purposes of this example that the interest rate for savings accounts is 5% while the interest rate for a chequeing accounts is 1%.

RowFixtures are more suitable for checking sets of data. Rows in the test data are compared to the contents of objects under test. Test data is used to examine an object under test for any missing or surplus (unexpected) data sets. The test shown in Figure E-2 is also relative to the behavior described by the “Perform

Transactions” UC and it evaluates the transaction log after the transactions performed in the test presented by the *ActionFixture* shown in Figure E-3. FIT uses the fixtures created to generate code skeletons for later implementation.

TransactionLog		
AccountType	TransactionType	Balance
Savings	Deposit	200
Chequeing	Deposit	300
Chequeing	Withdraw	100

Figure E-2: RowFixture example of checking account activities. The name of each attribute in the test data row is stated atop its respective column, while the name of the class under test is stated atop the fixture table.

PerformTransactions		
Check	SavingsBalance	0
Check	ChequeingBalance	0
Enter	Amount	200
Press	Deposit	Amount
Enter	Amount	300
Press	Deposit	Amount
Enter	Amount	100
Press	Withdraw	Amount
Check	SavingsBalance	200
Check	ChequeingBalance	200

Figure E-3: ActionFixture example of performing transactions.

When fixtures run, the data returned by the software under test is compared against the values provided in the tables. Test results are indicated by color-coding the cells containing the expected data (cell containing output). For example, running the RowFixture shown in Figure E-2 will result in coloring the data fields of “Balance” column. The following is the list of color codes for test results is defined in [Mugridge al. 2005] and are briefly explained below:

- **Green:** The software returned an expected value.
- **Red:** The software returned an incorrect value.
- **Yellow:** The software caused an exception to be thrown.

- **Grey background:** The cell was ignored for some reason.
- **Grey text:** When cell is left intentionally blank by the user, FIT will fill in the answer from your software.

Appendix F

Robustness Analysis of the

“Generate Restaurant Map for City”

UCs

Basic Flow:

1. Get restaurants for current city with valid zoom setting
 - 1.1. The MapViewer interface is provided with the current city and the zoom setting by the “Generate City” UC or the User.
 - 1.2. MapViewer invokes “get restaurants for city” given the current city and the zoom setting
 - 1.3. “get restaurants for City” then passes these parameters to “map scale is OK?” to check the validity of the zoom setting.
 - 1.4. Given the validity of the zoom setting, “map scale is OK?” then passes on the two parameters to the RestaurantServer
 - 1.5. RestaurantServer invokes MapServer to get the restaurant layer
 - 1.6. MapServer returns the Restaurant layer to RestaurantServer
 - 1.7. RestaurantServer then return the queried restaurants to “map scale is OK?”

1.8. “map scale is OK?” then returns the restaurants to “get restaurants for city”

1.9. “get restaurants for city” then creates the RestaurantCollection

Inputs: Current City, Zoom setting

2. Get Map

2.1. The MapViewer interface is provided with the current city and the zoom setting by the “Generate City” UC.

2.2. MapViewer invokes “get map for City” and provides it with the two external parameters

2.3. “get map for City” passes those two parameters to the MapServer which return the required map data that can be used to build the map

2.4. “get map for City” uses the returned map data to build a Map

3. Add Restaurant Icons to Map

3.1. “add Restaurant icons to map” retrieves information about the Restaurants to be displayed as icons from the RestaurantCollection

3.2. “add Restaurant icons to map” retrieves information about the desired display filters from the DisplayFilter

3.3. “add Restaurant icons to map” add the qualifying Restaurants as icons by editing the Map entity object

Inputs: DisplayFilter

Outputs: Edited Map entity object containing Restaurant icons

Alternative Flow: Invalid zoom setting

1. Get Restaurants for Current city with Invalid Zoom Setting
 - 1.1. The MapViewer interface is provided with the current city and the zoom setting by the “Generate City” UC or the User.
 - 1.2. MapViewer invokes “get restaurants for city” given the current city and the zoom setting
 - 1.3. “get restaurants for city” then passes these parameters to “map scale is OK?” to check the validity of the zoom setting.
 - 1.4. “map scale is OK?” determines that the given zoom setting is invalid invokes the “Show Invalid Zoom Popup” control object.
 - 1.5. The “Show Invalid Zoom Popup” object then creates a the interface object “Invalid Zoom Popup” to display a message prompting the User change the zoom setting.

Inputs: Current City, Zoom setting

Outputs: Invalid Zoom Popup

Appendix G

Analyzing UC “Display Rollover Information”

The “Display Rollover Information” UC *extends* the “View Map” UC to provide additional information to the “User” about restaurants displayed in the map”. The additional information are presented in the form of a popup and is only presented when the “User” points to a restaurant icon or clicks on it with the mouse cursor.

The UC description is shown in Figure G-1.

Basic Flow:

When the user rolls the mouse cursor over a restaurant icon, the application displays a map tip window containing the restaurant’s name. If the user clicks on the restaurant icon, a popup is generated to display a variety of information about the restaurant. Within the popup, the restaurant information is hyperlinked so that the user may click them to retrieve even further information. This additional information is provided by the *View Detailed Restaurant Info* UC.

Alternative Flow: Clicked on coordinates with multiple restaurants

If more than one restaurant is found at click coordinates, a list containing the names of the clicked-on restaurant is created and displayed in a map tip window.

If the user clicks a restaurant name from the displayed list, the application generates a popup containing further information regarding the selected restaurant.

Alternative Flow: Clicked on coordinates with no restaurants

If no restaurants are found at click coordinates, then the click is ignored.

Figure G-1: Textual description of the *Display Rollover Information UC*

G.1. Examining the UC Description and Creating its HLATs (Phase 1)

A common precondition for all flows is that the map is displayed. For the *Basic Flow* to be performed successfully, the map must contain at least one restaurant icon. In order to perform the *Alternative Flow: Clicked on coordinates with multiple restaurants*, where the “User” clicks on coordinates with multiple restaurant icons, it is required that the map contain at least two restaurant icons located at the same coordinates. Finally, for the *Alternative Flow: Clicked on coordinates with no restaurants*, where a “User” clicks on nothing, it is not necessary for the map to contain any restaurant icons. The set of HLATs created are shown in Table G-1.

Table G-1: HLATs for the *Display Rollover Information UC*

Test ID	Description	Expected Results
DRI- <i>Basic Flow</i>	<i>Precondition:</i> Map displayed with at least one restaurant icon <i>Input:</i> mouse movement over restaurant icon	A <i>MapTipWindow</i> showing the name of the restaurant that is pointed to by the cursor

	<i>Input:</i> mouse click over a single restaurant icon	A Popup window containing various information about the selected restaurant
<i>DRI-Alternative Flow: Clicked on coordinates with multiple restaurants</i>	<i>Precondition:</i> Map displayed with at least two restaurant icon displayed at the same coordinates <i>Input:</i> mouse click at coordinates with multiple restaurant icons	A <i>MapTipWindow</i> showing the names of the restaurants that were clicked on
	<i>Input:</i> mouse click over a restaurant name from a list	A Popup window containing various information about the selected restaurant
<i>DRI-Alternative Flow: Clicked on coordinates with no restaurants</i>	<i>Precondition:</i> Map displayed <i>Input:</i> mouse click where no restaurant icons are displayed	Nothing

It is important to note that the output of the *Alternative Flow: Clicked on coordinates with no restaurants* indicates that nothing should happen in response to a mouse click. However, how is it possible to verify that nothing happened? From a Software Testing perspective, to verify that nothing has happened, certain data values need to be checked in order to determine that the system state has not changed. Therefore, an infinite number of tests to explore an infinite number of scenarios will be required to ensure that the system behaves correctly, which is infeasible and unpractical. Therefore, it is a judgment call as to how many tests should be created, and which specific scenarios they should address.

G.2. Robustness Analysis (Phase 2)

Tracing the steps in each flow indicate that `MapView` handles mouse inputs (see Appendix B). For the *Basic Flow* and the *Alternative Flow: Clicked on coordinates with multiple restaurants*, the interface object `MapTipWindow` implements the display of the map tip window. The map tip window is generated

in response to a mouse pointer moving over it (*Basic Flow*) or a mouse clicking at coordinates where multiple restaurant icons are present (*Alternative Flow: Clicked on coordinates with multiple restaurants*). Meanwhile, for both the *Basic Flow* and the *Alternative Flow: Clicked on coordinates with multiple restaurants*, the `RestaurantInfoPopup` object handles the popup window containing information about the selected restaurant. The results of performing robustness on the flows are shown in Tables G-2 – G-4, respectively.

Basic Flow:

1. Display Map Tip Window containing the name of the pointed to Restaurant icon
 - 1.1. The “User” rolls the mouse cursor over a Restaurant icon presented by the “MapView” interface.
 - 1.2. The “MapView” detects that the mouse cursor is over a Restaurant icon and invokes “Change cursor and highlight Restaurant icon” to change the cursor to “selection hand” and the Restaurant icon to “selected Restaurant”.
 - 1.3. “Change cursor and highlight Restaurant icon” then invokes “Get Restaurant name(s)” to get the name of the pointed to Restaurant.
 - 1.4. “Create the Tip Text” is then invoked to create tip text containing the Restaurant name.
 - 1.5. “Show the tip window” is then invoked to show the tip window.

1.6. The tip window is then visually displayed and handled by the “Map Tip Window” interface.

Inputs: Coordinates of mouse pointer that occurred over a Restaurant icon

Outputs: Map Tip Window containing the name of the Restaurant that had the mouse roll over it

2. Display a popup window containing various information about the clicked on Restaurant

2.1. The “MapView” interface is provided with the coordinates of a mouse click from the “User”.

2.2. The “MapView” detects that the mouse was clicked on a single Restaurant icon or name in a list.

2.3. The attributes of the clicked Restaurant is retrieved using “Get Restaurant Attributes”

2.4. “Get Restaurant Attributes” retrieves the required Restaurant information from the “Restaurant” entity that corresponds to the clicked Restaurant.

2.5. The Restaurant attributes are then used by “Show Restaurant Info Popup” to prepare them for display.

2.6. After the Restaurant attributes are prepared, they are displayed and controlled by the “Restaurant Info Popup” interface.

Inputs: Coordinates of mouse click that occurred over a Restaurant icon

Outputs: Popup window containing the information of the Restaurant that was selected

Table G-2: Results of performing robustness analysis on the *Basic Flow*

Flow Step	Input/Output	Element or Action	Representative Objects
1	Input	Coordinates of mouse pointer that occurred over a restaurant icon	MapView
1	Output	MapTipWindow containing the name of the selected restaurant	MapTipWindow, MapView
2	Input	Coordinates of mouse click that occurred over a restaurant icon	MapView
2	Output	Popup window containing the information of the restaurant that was selected	RestaurantInfoPopup, MapView

Alternative Flow: Clicked on coordinates with multiple restaurants

1. Generate Pop-up of Clicked Restaurants
 - 1.1. The “MapView” interface is provided with the coordinates of a mouse click from the “User”.
 - 1.2. The “MapView” detects that the click was on multiple Restaurants and invokes “Get list of Restaurants”.
 - 1.3. The list of Restaurants are retrieved and stored in a “MultipleRestaurantCluster” object.
 - 1.4. The “MultipleRestaurantCluster” object then invokes “Get Restaurant name(s)” to get the name of each Restaurant it contains.
 - 1.5. “Get Restaurant name(s)” is then invoked to retrieve the name of each Restaurant in “MultipleRestaurantCluster”.

1.6. “Create the Tip Text” is then invoked to create tip text containing the Restaurant names.

1.7. “Show the tip window” is then invoked to show the tip window.

1.8. The tip window is then visually displayed and handled by the “Map Tip Window” interface.

Inputs: Coordinates of a mouse click that occurred over multiple Restaurant icons

Outputs: Map Tip Window showing a list of Restaurants corresponding to the Restaurants that were selected by the mouse click

2. Display a popup window containing various information about the clicked on Restaurant → See Basic Flow

Inputs: Coordinates of a mouse click that occurred over a Restaurant name in a Map Tip Window

Outputs: Popup window containing the information of the Restaurant that was selected

Table G-3: Results of performing robustness analysis on the *Alternative Flow: Clicked on coordinates with multiple restaurants*

Flow Step	Input/Output	Element or Action	Representative Objects
1	Input	Coordinates of mouse click that occurred over multiple restaurant icons	MapView
1	Output	MapTipWindow containing the a list of the restaurant names	MapTipWindow, MapView
2	Input	Coordinates of mouse click that occurred over a restaurant name in the MapTipWindow	MapTipWindow, MapView
2	Output	Popup window containing the information of the restaurant that	RestaurantInfoPopup, MapView

		was selected	
--	--	--------------	--

Alternative Flow: Clicked on coordinates with no restaurants

1. Ignore Mouse Click

1.1. The “MapView” interface is provided with the coordinates of a mouse click from the “User”.

Inputs: Coordinates of mouse click that occurred that do not match a Restaurant icon coordinates

Outputs: None

2. Ignore mouse click.

Table G-4: Results of performing robustness analysis on the *Alternative Flow: Clicked on coordinates with no restaurants*

Flow Step	Input/Output	Element or Action	Representative Objects
1	Input	Coordinates of mouse click that occurred over nothing	MapView
2	Output	<i>None</i>	RestaurantInfoPopup, MapViewer

Section G.3. Creating Low Level Acceptance Tests (Phase 3)

The EAT corresponding to the *Basic Flow* consists of four fixtures (Figure G-2):

- An ActionFixture to simulate a mouse rollover on a single restaurant icon. The ActionFixture also checks that a tip window (MapTipWindow) is displayed;
- a RowFixture to check that the MapTipWindow is displaying the expected restaurant;

- an ActionFixture to simulate a mouse click on the restaurant icon. The ActionFixture checks that a pop-up window (RestaurantInfoPopup) is displayed; and
- a RowFixture to examine that the RestaurantInfoPopup is displaying information about the selected restaurant.

MapView			
Press	MapView.setMouseCoordinate	Y coordinate	X coordinate
Check	MapView.MapTipWindow.isDisplayed()	True	
MapView.MapTipWindow.ListedRestaurants			
restaurantName			
Expected restaurant			
MapView			
Press	MapView.mouseClicked	Y coordinate	X coordinate
Check	MapView.RestaurantInfoPopup.isDisplayed()	True	
MapView.RestaurantInfoPopup.Restaurants			
restaurantName	isValetParking	isSmokingAvailable	isLiveMusic
Expected restaurant	True or False	True or False	True or False

Figure G-2: EAT of the Basic Flow

The EAT corresponding to the *Alternative Flow: Clicked on coordinates with multiple restaurants* consists of four fixtures (Figure G-3):

- An ActionFixture to simulate a mouse click over multiple restaurant icons and that the MapTipWindow is displayed; while a subsequent RowFixture examines the MapTipWindow produced to check that the expected list of restaurant names is displayed; and
- A second ActionFixture to simulate a mouse click on a single restaurant name displayed by the MapTipWindow, which is followed by a RowFixture to check the details of the restaurant that was selected.

MapViewer			
Press	MapViewer.setMouseCoordinate	<i>Y coordinate</i>	<i>X coordinate</i>
Check	MapViewer.MapTipWindow. isDisplayed()	True	
MapViewer.MapTipWindow.ListedRestaurants			
restaurantName			
<i>Expected restaurant1</i>			
<i>Expected restaurant2</i>			
<i>Expected restaurant3</i>			
MapViewer			
Press	MapViewer.mouseClicked	<i>Y_2 coordinate</i>	<i>X_2 coordinate</i>
Check	MapViewer.RestaurantInfoPopup. isDisplayed()	True	
MapViewer.RestaurantInfoPopup.Restaurants			
restaurantName	isValetParking	isSmokingAvailable	isLiveMusic
<i>Expected restaurant1</i>	<i>True or False</i>	<i>True or False</i>	<i>True or False</i>
<i>Expected restaurant2</i>	<i>True or False</i>	<i>True or False</i>	<i>True or False</i>
<i>Expected restaurant3</i>	<i>True or False</i>	<i>True or False</i>	<i>True or False</i>

Figure G-3: EAT of the Alternative Flow: Clicked on coordinates with multiple restaurants

The EAT for the *Alternative Flow: Clicked on coordinates with no restaurants* (Figure G-4) consists of one ActionFixture that simulates a mouse click over an area with no restaurants and checks that RestaurantInfoPopup is not displayed.

MapViewer			
Press	MapViewer.mouseClicked	<i>Y coordinate</i>	<i>X coordinate</i>
Check	MapViewer.RestaurantInfoPopup. isDisplayed()	False	

Figure G-4: EAT of the Alternative Flow: Clicked on coordinates with no restaurants